Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2012/2013

PRACA DYPLOMOWA MAGISTERSKA

Łukasz Szymon Sowa

Niezawodność i bezpieczeństwo w programowaniu współbieżnym

Opiekun pracy:
dr inż. Tomasz Jordan Kruk

Ocena .....

.....
Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Kierunek: Informatyka

Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 1989.02.06

Data rozpoczęcia studiów: 2012.02.20

Życiorys

Urodziłem się 6 lutego 1989 roku w Warszawie. W 2008 roku ukończyłem Społeczne Liceum Ogólnokształcące nr 5 w Milanówku. W tym samym roku rozpocząłem studia stacjonarne pierwszego stopnia na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej na kierunku Informatyka. W dniu 7 lutego 2012 roku uzyskałem tytuł zawodowy inżyniera za wyróżnioną pracę dyplomową pt. „Nowoczesne mechanizmy bezpieczeństwa w systemie Linux" i ukończyłem studia pierwszego stopnia z oceną celującą. 20 lutego 2012 roku rozpocząłem studia drugiego stopnia.

.....
Podpis studenta

Egzamin dyplomowy

Złożył egzamin dyplomowy w dniu .....

z wynikiem .....

Ogólny wynik studiów: .....

Dodatkowe wnioski i uwagi Komisji: .....

.....

.....

Streszczenie

W pracy podjęto tematykę związku pomiędzy współbieżnością a niezawodnością i bezpieczeństwem systemów informatycznych. W ramach badań w tym obszarze dokonano przeglądu błędów i podatności w popularnych i powszechnie wykorzystywanych aplikacjach współbieżnych. Na podstawie dokładnego rozpoznania trudności związanych ze współbieżnością, przedstawiono nowoczesne techniki programowania, które mogą pomóc w rozwiązaniu najważniejszych problemów. Ze względu na zaobserwowane zapotrzebowanie, szczególną wagę przywiązano do narzędzi wspierających programowanie współbieżne. Istotną częścią pracy jest opis procesu projektowania i implementacji innowacyjnego narzędzia do testowania i uruchamiania programów współbieżnych, które powstało w ramach praktycznej części pracowni dyplomowej.

Słowa kluczowe: *niezawodność, bezpieczeństwo, błędy w programowaniu, podatności, programowanie współbieżne, programowanie równoległe, techniki programowania, przeploty, determinizm, narzędzia programistyczne, testowanie, uruchamianie.*

Reliability and Security in Concurrent Programming

The following thesis presents the relationship between concurrency, reliability and security of computer systems. It describes bugs and vulnerabilities in popular and commonly used concurrent applications. A set of modern programming techniques is presented as a remedy for the aforementioned problems. Due to the observed demand, particular attention is paid to programming tools for concurrent programming. A substantial part of this paper thoroughly describes the design and development of an innovative tool created to test and debug concurrent applications.

Keywords: *reliability, security, bugs, vulnerabilities, concurrent programming, parallel programming, programming techniques, interleavings, determinism, programming tools, testing, debugging.*

Spis treści

Wstęp

Programowanie współbieżne nigdy wcześniej w historii informatyki nie było tak popularnym i ważnym tematem jak dziś. Jeszcze kilkanaście lat temu zagadnienie to było kojarzone przede wszystkim z wielozadaniowymi systemami operacyjnymi i znajdowało się w kręgu zainteresowań osób zajmujących się systemami operacyjnymi i systemami czasu rzeczywistego. Sytuacja ta utrzymywała się aż do momentu, kiedy powszechnie znane, empiryczne prawo *Moore'a* [81], ze względu na ograniczenia fizyczne, przestało obowiązywać w klasycznym wydaniu. Od tego czasu, ze względu na stale rosnące zapotrzebowanie na moc obliczeniową, producenci jednostek centralnych oferują architektury wielordzeniowe i wieloprocesorowe. Układy wielordzeniowe w typowych urządzeniach codziennego użytku, takich jak telefony komórkowe bądź laptopy przestają być nieuzasadnionym technologicznie luksusem, a stają się wręcz standardem. W obliczu tych zmian, współbieżność przestała być rozwiązaniem nietypowym czy opcjonalnym. Aby sprostać konkurencji i nadążyć za rozwojem technologicznym programiści muszą tworzyć programy współbieżne, wykorzystujące moc obliczeniową układów z wieloma jednostkami obliczeniowymi.

Pomimo długiej historii i wielu badań, współbieżność jest wciąż uważana za zagadnienie niezwykle trudne, przede wszystkim ze względu na jej złożoność, lecz także z powodu dynamiki zmian zachodzących w tej dziedzinie na przestrzeni ostatnich lat. Wszystko to powoduje, że programy współbieżne często są zawodniejsze od programów sekwencyjnych [69].

Wśród błędów popełnianych przez programistów aplikacji współbieżnych znajdują się również te najbardziej niepożądane, czyli błędy bezpieczeństwa. Jest to spostrzeżenie szczególnie ważne, gdyż dzięki rozwojowi Internetu, komputery stały się najważniejszym środkiem wymiany informacji i są niezwykle istotnym elementem światowej gospodarki, co w ostatnich latach przyczynia się do rozwoju przestępczości komputerowej, która w znacznej mierze korzysta z tego, że oprogramowanie napisano niepoprawnie.

W obliczu powyższych spostrzeżeń, analiza niezawodności i bezpieczeństwa istniejących i dopiero powstających aplikacji współbieżnych jest jednym z kluczowych aspektów rozwoju współczesnej inżynierii oprogramowania. Niestety, tematyka ta nie została dotychczas dogłębnie zbadana — większość istniejących opracowań dotyczy wyłącznie programów sekwencyjnych, fundamentalnie różnych od programów współbieżnych. Z tego powodu w niniejszej pracy przeanalizowano zagadnienia niezawodności i bezpieczeństwa w programowaniu współbieżnym.

Celem pracy jest przedstawienie związku pomiędzy współbieżnością a niezawodnością i bezpieczeństwem systemów informatycznych. Aby zrealizować sformułowane zadanie, w pierwszej kolejności przeanalizowano błędy we współbieżności w popularnych, powszechnie wykorzystywanych aplikacjach tak, aby dogłębnie poznać przyczyny, okoliczności występowania, a także skutki typowych usterek.

Następnie w podobny sposób zbadano błędy bezpieczeństwa, koncentrując się dodatkowo na wpływie współbieżności na techniki wykrywania podatności i obrony przed atakami. Po dokładnym rozpoznaniu trudności, które sprawia współbieżność, skupiono się na nowoczesnych technikach programowania, które mogłyby pomóc w rozwiązaniu zidentyfikowanych problemów. Pośród różnych przeanalizowanych środków realizacji, począwszy od dobrych praktyk aż do wyrafinowanych języków programowania, rozpoznano zapotrzebowanie na nowe, skuteczne narzędzia wspierające programowanie współbieżne. W ramach pracy stworzono więc innowacyjne narzędzie do testowania i uruchamiania programów współbieżnych oparte na koncepcji deterministycznej weryfikacji przeplotów.

Układ rozdziałów niniejszej pracy w naturalny sposób odzwierciedla przebieg powyższego badania. W rozdziale 1 zaprezentowano analizę błędów w popularnych programach współbieżnych. W tej części pracy rozważono również, jakie są przyczyny powstawania usterek i jakie są ich bezpośrednie skutki. W kolejnym rozdziale przeprowadzono analogiczne badanie, koncentrując się tym razem jednak na błędach bezpieczeństwa wynikających z niewłaściwego wykorzystania współbieżności w różnych warstwach systemu. W rozdziale przeprowadzono również dyskusję na temat różnic pomiędzy błędami funkcjonalnymi a błędami bezpieczeństwa, a także omówiono wpływ współbieżności na techniki wykrywania podatności i obrony przed atakami. Rozdział 3 dotyczy w całości nowoczesnych metod programowania współbieżnego. W rozdziale tym opisano najlepsze praktyki, przedstawiono trendy w rozwoju technik i mechanizmów, a także zaprezentowano zaawansowane języki programowania oferujące wsparcie dla współbieżności. Ze względu na powolny rozwój narzędzi wspierających programowanie współbieżne, tę tematykę rozwinięto w rozdziale 4, opisując zarówno ogólne koncepcje, jak i konkretne, funkcjonujące rozwiązania.

W związku ze zidentyfikowanym w rozdziale 4 zapotrzebowaniem, na podstawie wiedzy pozyskanej podczas pozostałych badań przeprowadzonych w czasie pracy, zaprojektowano i zaimplementowano nowatorskie narzędzie do testowania i uruchamiania programów współbieżnych za pomocą deterministycznej weryfikacji przeplotów. Koncepcję tej metody, a także proces projektowania, implementacji i wnikliwych testów połączonych z badaniami skuteczności, zaprezentowano w rozdziale 5. Uzupełnieniem treści rozdziału jest przedstawienie przykładowego wdrożenia opisywanego rozwiązania, a także dyskusja nad perspektywami jego rozwoju.

1. Błędy w programowaniu współbieżnym

Ze względu na niedeterminizm, błędy w oprogramowaniu współbieżnym są uważane za jedne z trudniejszych do odnalezienia i zdiagnozowania [127]. Pomimo tego zdecydowana większość opracowań dotyczących niezawodności bazuje wyłącznie na badaniach programów sekwencyjnych [69]. Istniejące prace na temat błędów w programach współbieżnych dotyczą zwykle konkretnych przypadków lub opierają się na uproszczonych przykładach, stworzonych na potrzeby badania. Poza nielicznymi wyjątkami [69, 122], nie istnieją prace, w których analiza błędów zostałaby przeprowadzona na podstawie różnych, istniejących i funkcjonujących programów współbieżnych. Badanie błędów w rzeczywistych, rozwijanych i wykorzystywanych programach współbieżnych pozwala sformułować uogólnione wnioski, które mogą mieć zastosowanie także do innych projektów. Przeprowadzona w ten sposób analiza otwiera zatem drogę do dalszych, pogłębionych eksploracji.

W związku z powyższym w niniejszym rozdziale skupiono się na przedstawieniu typowych błędów występujących w istniejących aplikacjach współbieżnych. Analiza ma na celu nakreślenie i uwypuklenie istniejących problemów, co będzie stanowiło podstawę do dalszych rozważań. W pierwszej części rozdziału przedstawiono podstawę teoretyczną — różne klasyfikacje błędów w programach współbieżnych. W kolejnym podrozdziale dokonano przeglądu konkretnych błędów o zróżnicowanej naturze w istniejących aplikacjach z dostępnym kodem źródłowym. Następnie przeprowadzono analizę przyczyn i skutków występujących błędów. Kluczowe wnioski z badania zebrano w krótkim podsumowaniu.

1.1. Typy błędów

Pogrupowanie błędów występujących w oprogramowaniu jest kluczowym elementem ich analizy. Odpowiednia systematyka nie tylko ułatwia opis, lecz także pozwala, na podstawie wyodrębnionych wspólnych cech wewnątrz grupy, zaproponować uniwersalne metody wykrywania bądź też korekcji błędów. Klasyfikacja stosowana do błędów w programach sekwencyjnych [8] okazuje się niezbyt użyteczna dla programów współbieżnych [28], gdyż błędy z pozoru podobne mają zasadniczo różne przyczyny i okoliczności występowania, a jedynie wspólny skutek.

Z tego powodu, dla programów współbieżnych w literaturze zaproponowano wiele różnych metod grupowania błędów. Autorzy koncentrują się zwykle na przyczynach (pośrednich i bezpośrednich), skutkach lub wzorcach działań prowadzących do usterek. Poniżej przedstawiono najpopularniejsze sposoby systematyzacji błędów prezentujące różne ujęcia zagadnienia.

1.1.1. Podział ze względu na skutki

Pierwszy z podziałów [121] dokonuje rozróżnienia na podstawie skutków błędu w programie współbieżnym. Te skutki to:

— **Zakleszczenie** — zadania zostają permanentnie zablokowane.
— **Nadpisanie informacji** — istotne informacje zostają nadpisane przez zadanie (np. przez współbieżne zapisy danych).
— **Utrata spójności danych** — informacje zapisywane przez różne zadania niszczą ich logiczną bądź fizyczną strukturę (np. przez zapisy niezserializowane w ściśle określony sposób).
— **Degradacja wydajności** — niezbalansowany dostęp do dzielonych zasobów powoduje problemy z wydajnością.

Taka klasyfikacja jest wyjątkowo użyteczna przy opracowywaniu metod łagodzenia i przeciwdziałania konsekwencjom błędów. Należy przy tym zwrócić uwagę, że twórcy powyższej systematyki skupili się wyłącznie na skutkach *bezpośrednich*, pominąwszy całkowicie kwestię skutków pośrednich (takich jak np. naruszenie bezpieczeństwa systemu). Ponadto opisywanie błędów w ten sposób jest możliwe w zasadzie wyłącznie po wyizolowaniu i przebadaniu danego problemu, ponieważ bardzo trudno jest wnioskować o następstwach błędu wyłącznie na podstawie kodu źródłowego czy analizy działającego programu. Z tego powodu powyższa klasyfikacja wydaje się nieużyteczna przy tworzeniu narzędzi wspomagających wykrywanie ewentualnych błędów.

1.1.2. Podział ze względu na wzorzec

Kolejna prezentowana klasyfikacja [28] dzieli błędy ze względu na wzorzec (ang. *bug pattern*), który prowadzi do ich wystąpienia:
— **Niewłaściwe założenia dotyczące ochrony przed współbieżnością**
  — **Naruszenie niepodzielności** — ten typ błędu występuje w następstwie mylnego założenia, że pewna operacja (albo grupa operacji) wykona się niepodzielnie (np. operacja inkrementacji na platformie *JVM*).
  — **Dwuetapowy dostęp** — błąd tego typu występuje, gdy cała sekwencja operacji nie znajduje się w sekcji krytycznej przez co może zostać przerwana w trakcie wykonania (np. niechronione sprawdzenie pewnego warunku na wskaźniku do danych, a następnie wykorzystanie rezultatów tej operacji, które mogą się zdezaktualizować).
  — **Brak lub użycie niewłaściwego zamka** — błąd występuje, gdy zadania przy dostępie do zasobu dzielonego wykorzystują niewłaściwe zamki (lub nie wykorzystują ich wcale).
  — **Niewłaściwa inicjalizacja danych** — ten typ błędy występuje w programach napisanych w językach programowania wykorzystujących maszyny wirtualne (np. *Java*), gdzie przy współbieżnym dostępie zachodzi kopiowanie z pamięci wątku, do pamięci głównej (ew. innego wątku). W takim przypadku może zostać zwrócona poprawna (niezerowa) referencja do obiektu, którego pola nie zostały jeszcze skopiowane. Błąd ten ujawnia się wyjątkowo często przy zastosowaniu niepoprawnego wzorca *double–checked locking* [38].
— **Niewłaściwe założenia dotyczące przeplotu zadań**
  — **Usypianie zadania** — błąd ten popełniany jest często przez niedoświadczonych bądź nierozważnych programistów i polega na usypianiu zadania na pewien z góry określony czas w celu powstrzymania zadania przed przedwczesnym wykonaniem pewnych operacji. Błąd ten często obniża wydajność i może powodować nieoczekiwane awarie systemu w odmiennych od testowego środowiskach.

  - **Zgubienie sygnału wznowienia** — ten typ błędu występuje w przypadku, gdy zostaje zgubiony sygnał wznowienia zadania z powodu przedwczesnej emisji (np. wywołanie `notify()` w pierwszym zadaniu przed `wait()` w zadaniu drugim).
- **Zakleszczenie**
  - **Zawieszenie w sekcji krytycznej** — błąd tego typu występuje w momencie, gdy zadanie znajdujące się w sekcji krytycznej wykonuje operację blokującą, która może nigdy się nie zakończyć, prowadząc tym samym do zagłodzenia innych zadań próbujących uzyskać dostęp do tej samej sekcji krytycznej programu. Błędy tego typu są najczęstszym powodem zatrzymania pracy systemów informatycznych.
  - **Osierocony wątek** — ten typ błędu znany jest z architektur *master–slave*, gdzie jedno zadanie nadzoruje pracę drugiego przez pewną strukturę współdzieloną (np. kolejkę). Jeśli wątek nadzorujący zakończy swoje działanie w sposób nieoczekiwany, wątek podrzędny najczęściej pozostanie na zawsze zablokowany.

Powyższa klasyfikacja jest użyteczna przy tworzeniu narzędzi służących do wykrywania błędów we współbieżności, ponieważ wskazuje gotowe wzorce prowadzące do powstania danego błędu. Została ona opracowana na podstawie analizy błędów w programach napisanych w języku *Java*, co powoduje, że niektóre z opisanych kategorii usterek mogą nie występować w innych językach programowania. Można zatem powiedzieć, że ten podział jest nieco zbyt drobnoziarnisty, aby stosować go do każdego rodzaju oprogramowania.

1.1.3. Podział ze względu na przyczynę

Ostatnia z opisywanych systematyk [69] jest uproszczoną wersją podziału według wzorca w taki sposób, aby miała zastosowanie do różnych rodzajów oprogramowania tworzonego na różne platformy:

- **Błędy z zakleszczeniem** — to wszystkie błędy, które prowadzą do zakleszczenia zadań.
- **Błędy bez zakleszczenia**
  - **Naruszenie niepodzielności** — to typ błędu, który występuje w następstwie mylnego założenia o niepodzielności operacji (pojedynczej bądź całej grupy).
  - **Naruszenie kolejności** — błąd tego typu występuje, gdy założenie o dokładnej kolejności wykonania operacji współbieżnych okazuje się fałszywe.
  - **Inne** — to kategoria zbiorcza zawierająca wszystkie błędy nie pasujące do żadnej z powyższych grup.

Powyższe grupowanie wyróżnia trzy zasadnicze kategorie oraz dodatkową kategorię zbiorczą (*Inne*) dla przypadków, które nie pasują do żadnej pozostałej. Istnieje zatem obawa, że ta klasyfikacja jest zbyt uproszczona (wiele przypadków trafia do kategorii *Inne*), a tym samym nieużyteczna przy analizie. Autorzy tej systematyki przeprowadzili jednak badanie [69], w którym sprawdzili procentowy rozkład błędów w poszczególnych grupach, z którego wynikło, że błędy zakleszczenia stanowią ok. 26.7% wszystkich błędów we współbieżności, błędy naruszenia niepodzielności ok. 48.6%, błędy naruszenia kolejności ok. 22.9%, a pozostałe błędy to jedynie ok. 1.8%. Widać zatem, że stosując powyższą klasyfikację, odsetek pominiętych błędów nie jest duży. Z powodu zadowalającej prostoty i odpowiedniego uogólnienia przytoczona klasyfikacja będzie używana do opisu błędów w niniejszym badaniu.

Niewątpliwą zaletą powyższego grupowania jest także zorientowanie na przyczyny powstawania błędów, co pozwoli na wyciągnięcie odpowiednich wniosków i zaproponowanie środków zaradczych.

1.2. Analiza przykładowych błędów

Aby dobrze zrozumieć naturę błędów w programach współbieżnych — ich przyczyny, przebieg i skutki — przeanalizowano rzeczywiste błędy w popularnych i powszechnie używanych aplikacjach. Do badania wybrano: jądro systemu *Linux*, system zarządzania relacyjnymi bazami danych *MySQL*, serwer *HTTP Apache*, pakiet aplikacji internetowych *Mozilla* (*Firefox* oraz *Thunderbird*) oraz pakiet biurowy *Apache OpenOffice*. Są to dojrzałe projekty (od 10 do 21 lat rozwoju), z rozbudowanym kodem źródłowym (od 1 do 16 milionów linii kodu) oraz dobrze zarządzanymi bazami błędów (ang. *bug trackers*). Projekty te ze względu na zróżnicowane przeznaczenie wykorzystują współbieżność do różnych celów (wielozadaniowość, jednoczesna obsługa wielu zapytań, responsywny interfejs użytkownika itd.), co zwiększa użyteczność i kompletność dokonanego przeglądu. W badaniu wykorzystano projekty typu *open source* ze względu na łatwy dostęp do baz błędów, kodu źródłowego, a także komentarzy społeczności.

Poniżej zaprezentowano i skomentowano niektóre z przeanalizowanych błędów, kategoryzując je przy tym ze względu na przyczynę występowania. Omówienie zakończono wnioskami z przeprowadzonego badania. Poniższe opracowanie błędów sporządzono na podstawie analizy [69] oraz baz błędów [6, 7, 62, 82, 95].

1.2.1. Zakleszczenie

Pierwszym omówionym błędem zakleszczenia (zilustrowanym na rysunku 1.1) jest błąd w sterowniku do systemu plików *XFS* w jądrze systemu operacyjnego *Linux*. Błąd ten polega na rekurencyjnym wywołaniu funkcji zwalniającej zakresy `xfs_rtfree_extent()`, która zamyka zamek dostępu do odpowiedniej struktury bitmapowej. Funkcja ta mogła zostać wywołana dla tego samego zakresu wielokrotnie w pętli procedury `xfs_bunmapi()` usuwającej mapowanie. Wielokrotne zamknięcie tego samego zamka prowadziło do zakleszczenia.

Tego typu błąd, pomimo krytycznego znaczenia dla stabilności systemu, mógł zostać z łatwością przeoczony, ponieważ występuje wyłącznie w jednej, nietypowej i bardzo złożonej ścieżce wykonania (ze względu na czytelność kodu pominięto to na rysunku). Najprostszym sposobem naprawienia tej usterki mogłoby być zastosowanie zamków rekurencyjnych, jednak w wielu środowiskach (w tym w społeczności programistów jądra *Linux* [119]) używanie konstrukcji tego typu uważane jest za złą praktykę [108]. Ostatecznie przeniesiono kod zamykania zamków do funkcji `xfs_bunmapi()` tak, aby blokowanie występowało tylko jeden raz. W kodzie `xfs_rtfree_extent()` pozostawiono natomiast dla pewności asercje, które zapewniają, że przed zwolnieniem zakresu odpowiedni zamek został zamknięty.

Kolejnym błędem (przedstawiony na rysunku 1.2) jest zakleszczenie na skutek niewłaściwej obsługi procesów potomnych w pakiecie oprogramowania *Mozilla*. Bezpośrednią przyczyną błędu w tym wypadku nie są zatem niewłaściwie zastosowane zamki, a blokujące wywołania systemowe. Kod funkcji `safe_pclose()` mający za zadanie zakończenie procesów potomnych we właściwy sposób, w trzech różnych przypadkach może doprowadzić do powstania procesu zombie. Po pierwsze

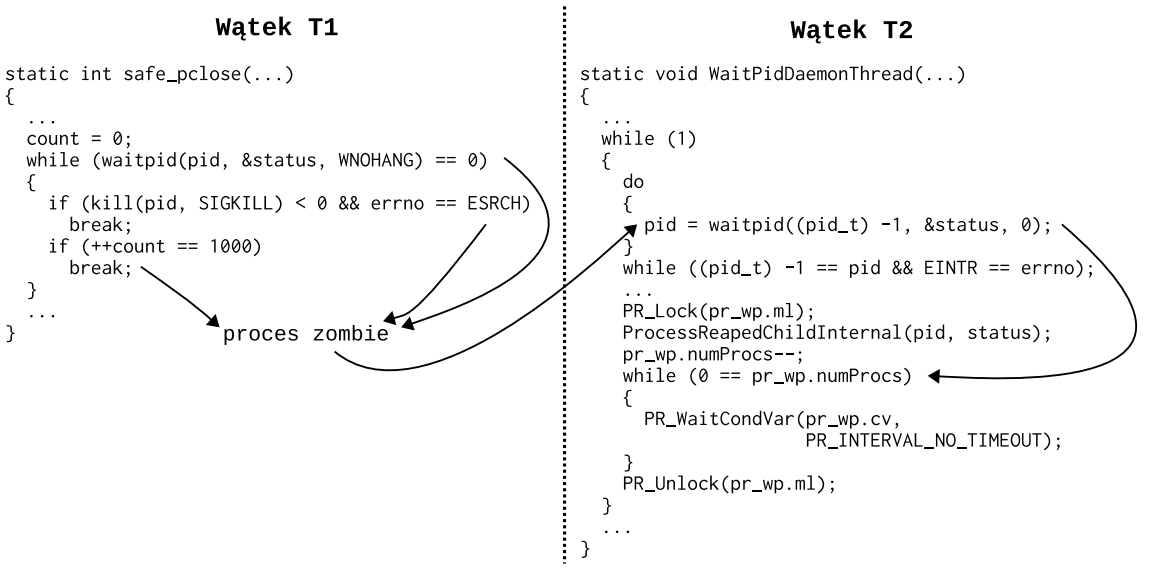
Wątek T1

```
int xfs_bunmapi(...)
{
  ...
  while (...)
  {
    ...
    error = xfs_bmap_del_extent(ip, tp, &lastx, flist, cur, &del, &tmp_logflags, whichfork);
    ...
  }
  ...
}

STATIC int xfs_bmap_del_extent(...)
{
  ...
  error = xfs_rtfree_extent(tp, bno, (xfs_extlen_t)len);
  ...
}

int xfs_rtfree_extent(...)
{
  ...
  /*
   * Synchronize by locking the bitmap inode.
   */
  xfs_ilock(mp->m_rbmip, XFS_ILOCK_EXCL);
  xfs_trans_ijoin_ref(tp, mp->m_rbmip, XFS_ILOCK_EXCL);
  ...
}
```

Rysunek 1.1: Zakleszczenie na skutek powtórnego (rekurencyjnego) zamknięcia zamków w jądrze systemu operacyjnego *Linux* w sterowniku do systemu plików *XFS*.

błędnie założono, że wywołanie `kill()` nie może zwrócić błędu `ESRCH` dla procesu zombie — specyfikacja *POSIX* faktycznie zawiera takie zalecenia, jednak wiele systemów nie implementuje tego w taki sposób. Po drugie funkcja `waitpid()`, tak jak wiele powolnych wywołań systemowych, może zwrócić błąd `EINTR` na skutek przerwania np. sygnałem i taka sytuacja nie została obsłużona. Po trzecie wprowadzono licznik, który, po z góry określonej liczbie obiegów pętli, przerywa oczekiwanie nie zwracając procesu potomnego. Powstały w jeden z wyżej opisanych sposobów proces zombie może zostać przechwycony w funkcji `WaitPidDaemonThread()`, która stworzona została do innych celów. Każdy dodatkowy proces potomny przechwycony wewnątrz `WaitPidDaemonThread()` spowoduje zmniejszenie wartości wewnętrznego licznika procesów, szybko doprowadzając do zakleszczenia.

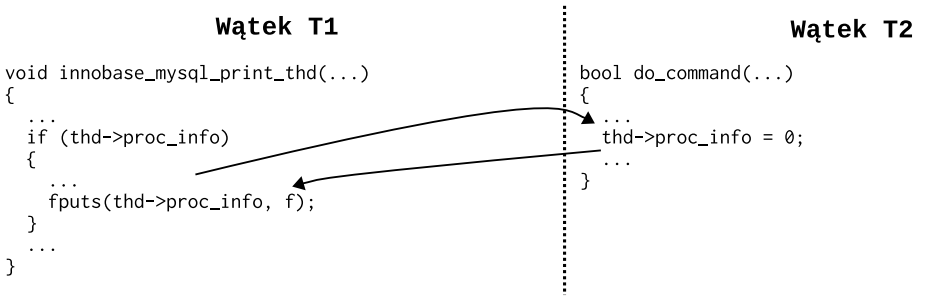
W pierwszych dwóch przypadkach błędy doprowadzające do powstania procesu zombie wynikają przede wszystkim z nieznajomości złożonych zasad funkcjonowania sygnałów w systemach z rodziny *UNIX*, a także z powodu niezgodności implementacji systemów operacyjnych ze standardem *POSIX*. Trzeci powód jest skutkiem nierozwagi i braku doświadczenia programisty tworzącego omawiany kod, ponieważ jest to przykład opisanego wcześniej antywzorca *usypianie zadania* — czekanie określony czas jest wrażliwe na wiele czynników takich jak platforma sprzętowa czy tryb pracy planisty, a więc może działać tylko w ściśle określonych warunkach. Poprawa opisanych błędów wymagała napisania właściwej obsługi sygnałów, a także usunięcia licznika przerywającego czekanie.



Rysunek 1.2: Zakleszczenie w pakiecie oprogramowania *Mozilla* występujące z powodu utracenia informacji o procesie potomnym.

1.2.2. Naruszenie niepodzielności

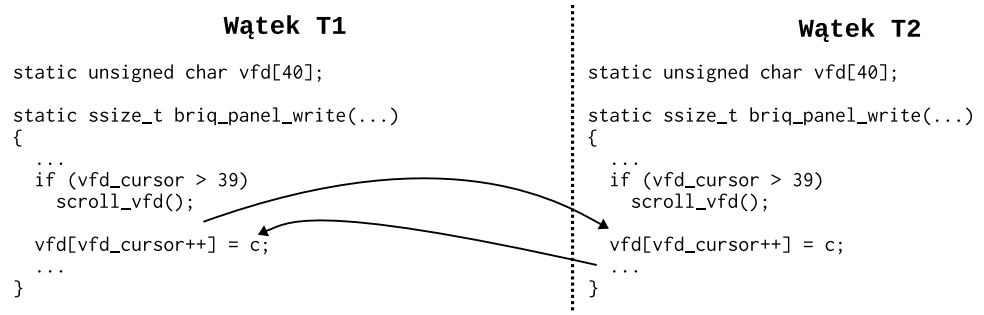
Pierwszy z opisanych błędów naruszenia niepodzielności występował w systemie zarządzania bazami danych *MySQL*. Jeśli podczas działania serwera wystąpił przeplot wątków przedstawiony na rysunku 1.3, to wątek *T1* odwoływał się do wyzerowanego wskaźnika, powodując tym samym krytyczny błąd i restart systemu. Zwyczajne sprawdzenie warunku zastosowane przez programistę nie jest wystarczające — musi się odbyć wraz z użyciem wyniku w sposób niepodzielny. Dostęp do `thd->proc_info` powinien być zatem chroniony odpowiednim zamkiem.



Rysunek 1.3: Naruszenie niepodzielności w systemie zarządzania bazami danych *MySQL* występujące w wyniku możliwości przeplotu operacji wykorzystania i zerowania wskaźnika.

Kolejny błąd (przedstawiony na rysunku 1.4) z tej kategorii znajdował się w kodzie sterownika wyświetlacza minikomputera *briQ* w jądrze *Linux*. Odwołania do zmiennej kursora `vfd_cursor` wydają się z pozoru bezpieczne, bo występują w obrębie tej samej funkcji, jednak nic nie stoi na przeszkodzie, aby wiele zadań jednocześnie wykonywało zapis danych do tablicy wyświetlacza. W takiej sytuacji może

zdarzyć się, że pomimo sprawdzonego wcześniej warunku dostępnego miejsca, bufor został w trakcie zapełniony przez inne zadanie (przykładowy błędny przebieg zaznaczono na rysunku strzałkami, zakłada się, że wątek *T1* zapisuje znak na pozycji 39) — zostanie nadpisana pamięć znajdująca się bezpośrednio za buforem. Błąd ten sklasyfikowano jako potencjalnie zagrażający bezpieczeństwu systemu, ponieważ teoretycznie istnieje możliwość zapisania dowolnych danych w dowolnym miejscu pamięci jądra. Usunięcie błędu jest bardzo proste — należy zastosować zamek wzajemnego wykluczania zapisów do bufora.

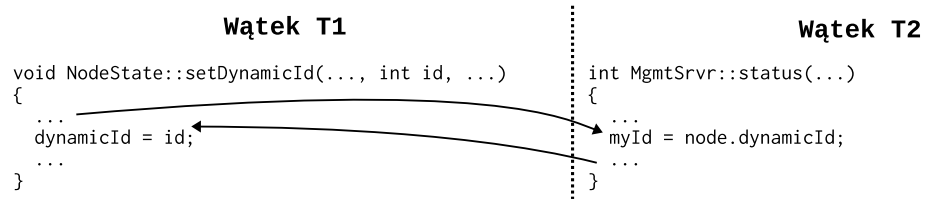


Rysunek 1.4: Naruszenie niepodzielności w jądrze systemu *Linux* występujące z powodu braku synchronizacji w dostępie do kursora, który jest zasobem dzielonym.

Obie opisane powyżej usterki, należące do rodziny *TOCTTOU* (ang. *time of check to time of use*), są przyczyną wielu poważnych błędów bezpieczeństwa, w szczególności w systemach operacyjnych. Dokładny opis problemu przedstawiono w rozdziale 2.

1.2.3. Naruszenie kolejności

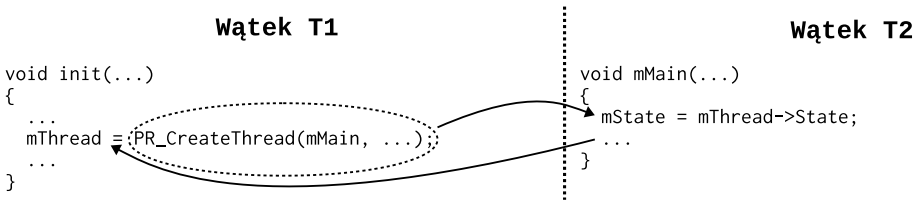
Pierwszy błąd naruszenia kolejności (przedstawiony na rysunku 1.5) występował w projekcie *MySQL*. Opisywana usterka polegała na użyciu zmiennej w wątku *T2*, zanim została poprawnie zainicjowana w wątku *T1*, co jest przykładem najbardziej elementarnego błędu naruszenia kolejności.



Rysunek 1.5: Naruszenie kolejności w systemie zarządzania bazami danych *MySQL* występujące w wyniku możliwego użycia zmiennej przed inicjalizacją.

Poprawienie tej usterki we właściwy sposób nie jest zadaniem łatwym, ponieważ użycie bezpośrednich mechanizmów synchronizacji pomiędzy `NodeState::setDynamicId()`, a `MgmtSrvr::status()`, czyli między metodami niezależnych typów, naruszałoby zasadę hermetyzacji programowania obiektowego [32]. Właściwe podejście wymaga synchronizacji funkcji tworzących obiekty i wykorzystujących ich metody lub zastosowanie mechaniki monitora [112].

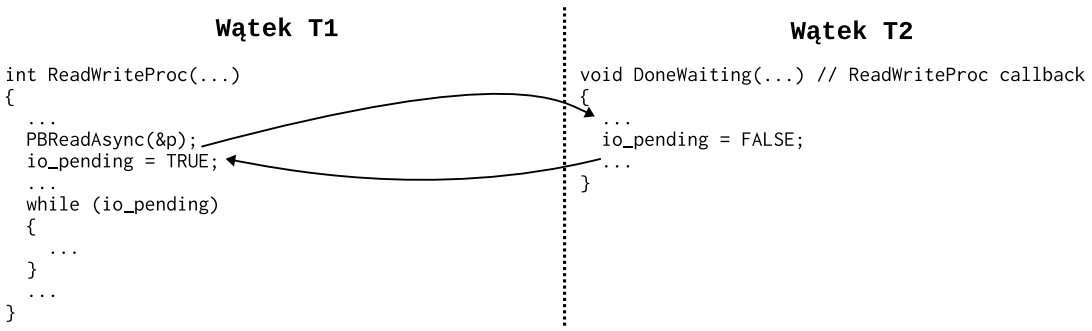
Kolejny błąd (przestawiony na rysunku 1.6) również dotyczy użycia przed właściwą inicjalizacją, tym razem w projekcie oprogramowania *Mozilla*. W tym przypadku interesujące jest to, że wyścig występuje pomiędzy wykonaniem pierwszej instrukcji wątku *T2*, a przypisaniem wartości zwracanej z funkcji ten wątek tworzącej. Istnieje zatem bardzo wąskie okno czasowe, w którym błąd może wystąpić. Ponadto na niektórych platformach programowych usterka może w ogóle się nie ujawniać — stanie się tak w przypadku, gdy utworzony wątek rozpocznie swoje działanie dopiero po zakończeniu (zablokowaniu) wątku tworzącego.



Rysunek 1.6: Naruszenie kolejności w pakiecie oprogramowania *Mozilla* wynikające z błędnego założenia, że funkcja tworząca wątek zwróci wynik przed rzeczywistym uruchomieniem wątku.

Identyfikacje problemu utrudnia fakt, że przypisanie i wywołanie funkcji tworzącej wątek `PR_CreateThread()` znajdują się w pojedynczej klauzuli języka *C*, co może powodować fałszywe przekonanie, że operacja ta jest pojedynczą instrukcją, która wykona się niepodzielnie. Naprawa, po uprzednim zrozumieniu skąd bierze się błąd, jest bardzo prosta i polega na dodaniu mechanizmu synchronizacji do zmiennej `mState`.

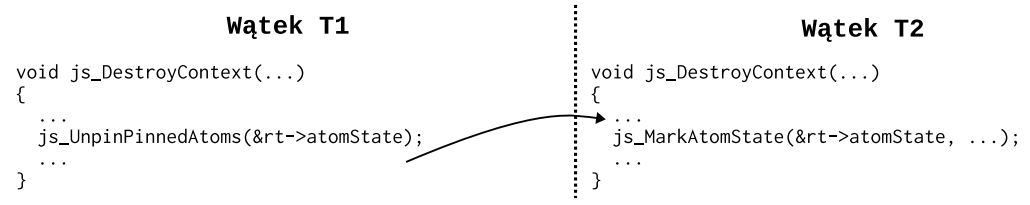
Następny błąd (przedstawiony na rysunku 1.7), także w oprogramowaniu *Mozilla*, jest nieco podobny do wyżej opisanego. Tym razem wyścig występuje między funkcją wywołującą asynchroniczny odczyt, a wywołaniem zwrotnym (ang. *callback*) tej operacji wejścia/wyjścia. Jeżeli operacja asynchronicznego odczytu wykona się dostatecznie szybko, to jej wywołanie zwrotne `DoneWaiting()` ustawi zmienną `io_pending` na wartość `FALSE`, zanim funkcja `ReadWriteProc()` przypisze tej zmiennej wartość `TRUE`. W ten sposób informacja o wykonaniu operacji wejścia/wyjścia zostanie utracona i wątek *T1* pozostanie w pętli oczekującej na zakończenie.



Rysunek 1.7: Naruszenie kolejności w pakiecie oprogramowania *Mozilla* wynikające z błędnego założenia, że wywołanie zwrotne nie wykona się przed instrukcją następującą bezpośrednio po funkcji, która to wywołanie stworzyła.

W tym przypadku również bardzo łatwo o przeoczenie, ponieważ można z bardzo dużym prawdopodobieństwem zakładać, że operacja wejścia/wyjścia będzie trwała istotnie dłużej niż przypisywanie zmiennej wartości. Niemniej jednak w specyficznych przypadkach (np. błędu odczytu) planista zatrzyma wykonanie `ReadWriteProc()` w wątku *T1* pozwalając dokończyć wywołanie zwrotne `DoneWaiting()` w wątku *T2*. Wystarczającą strategią naprawy błędu w tym wypadku jest zamiana kolejności klauzul, tj. najpierw przypisanie zmiennej `io_pending` wartości `TRUE`, a dopiero następnie wykonanie operacji asynchronicznego odczytu. W ten sposób mogłoby dojść do innego błędu np. gdyby pewien wątek odczytał zmienną, nim rzeczywiście operacja się zacznie. Jednak w tym przypadku jest to rozwiązanie w zupełności wystarczające i poprawne.

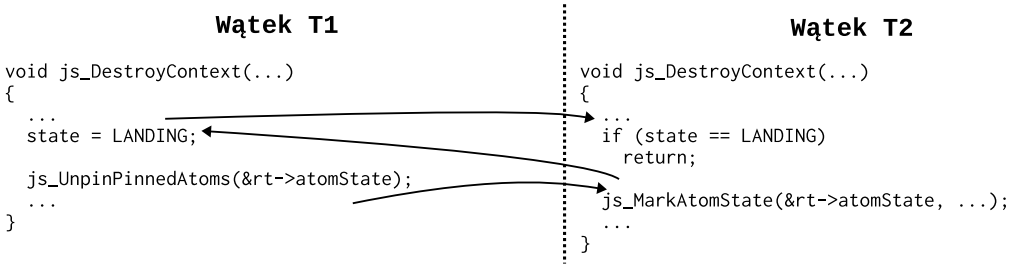
Ostatnim błędem (przedstawionym na rysunkach 1.8, 1.9, 1.10) w kategorii błędów naruszenia kolejności jest usterka w projekcie *Mozilla*. W celu zwolnienia pamięci wiele wątków wykonuje funkcję `js_DestroyContext()`, która operuje na zasobach współdzielonych. Ostatni wątek wykonujący tę procedurę powinien zwolnić pamięć za pomocą podprogramu `js_UnpinPinnedAtoms()`. Zdarzało się jednak, że ostatni wątek był nieco szybszy od wątku przedostatniego i funkcja zwalniająca pamięć `js_UnpinPinnedAtoms()` była wywoływana przed procedurą `js_MarkAtomState()`, która próbowała skorzystać z tej pamięci, doprowadzając do awarii oprogramowania. Proste sprawdzenie warunku, czy dany wątek jest wątkiem ostatnim, okazało się niewystarczające.



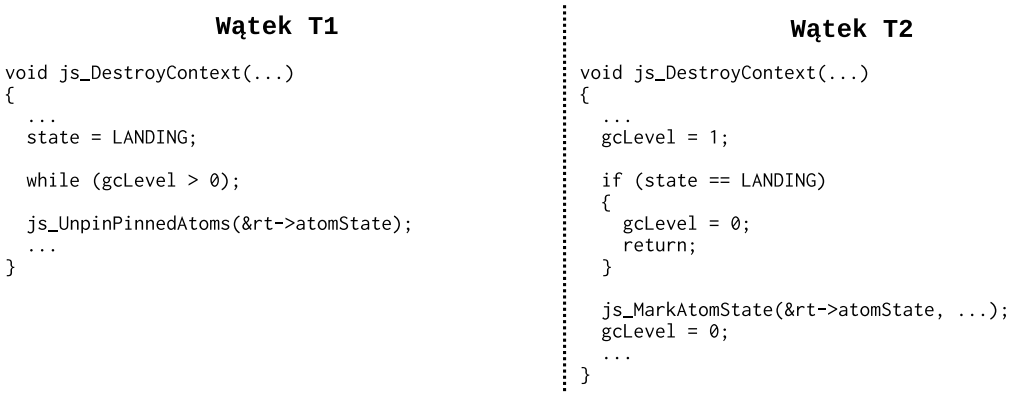
Rysunek 1.8: Naruszenie kolejności w pakiecie oprogramowania *Mozilla* wynikające z niezastosowania mechanizmu wymuszenia odpowiedniej relacji następstwa między wykonaniami funkcji.

Interesujący jest sposób, w jaki próbowano tę usterkę naprawić (zob. rysunek 1.9). Aby wyeliminować wyżej opisaną sytuację dodano zmienną wykluczającą `state`, która w razie potrzeby miała przerywać wykonanie funkcji `js_DestroyContext()` w wątku *T2*. Zmienne wykluczające (zwane też czasem blokowaniem zmiennych) są znanym i popularnym w literaturze wzorcem niewłaściwej synchronizacji [112] ze względu na możliwość pojawienia się wyścigu w wąskim oknie czasowym. Co ciekawe, to niewłaściwe rozwiązanie zostało najpierw zaakceptowane, a ponownie podjęto próbę naprawy znacznie później, dopiero po formalnym przedstawieniu przypadku, w którym ten kod nie zadziała poprawnie.

Do pierwszej, przedstawionej wyżej łaty, zaproponowano kolejną (zob. rysunek 1.10), która miała definitywnie rozwiązać zarówno pierwotny błąd, jak i błąd w stworzonej wcześniej łacie. Z kodu nie usunięto zmiennej wykluczającej `state`, a wprowadzono dodatkową, która wstrzymuje w pętli aktywnego oczekiwania wątek *T1* do czasu, aż wątek *T2* zakończy przetwarzanie. Po tej poprawce awarie oprogramowania przestały występować, jednak zastosowanie aktywnego oczekiwania jest zdecydowanie kontrowersyjnym rozwiązaniem, które zmniejsza wydajność aplikacji. Można zatem powiedzieć, że w ten sposób poważny problem zastąpiono innym,



Rysunek 1.9: Naruszenie kolejności w pakiecie oprogramowania *Mozilla*. Błędna łata do usterki przedstawionej na rysunku 1.8 wykorzystująca niepoprawne zmienne wykluczające.



Rysunek 1.10: Naruszenie kolejności w pakiecie oprogramowania *Mozilla*. Kolejna łata do usterki przedstawionej na rysunku 1.8 — poprawna, lecz wykorzystująca aktywne oczekiwanie.

mniej zauważalnym. Takie rozwiązanie zdecydowanie nie jest satysfakcjonujące — lepszym wyjściem mógłby się okazać semafor zliczający lub pewien rodzaj bariery.

1.2.4. Inne

Ostatni z prezentowanych błędów (przedstawiony na rysunku 1.11) występował w systemie zarządzania bazami danych *MySQL* i nie zalicza się do żadnej z omówionych wcześniej kategorii. Aby rozwiązać problem zakleszczeń, programiści projektu zdecydowali się śledzić w wątku monitorującym czasy oczekiwania danego zapytania na zwolnienie blokady. W przypadku, gdy czas ten przekraczał pewną z góry ustaloną wartość `fatal_timeout`, wątek monitorujący interpretował to jako zakleszczenie i serwer był uruchamiany ponownie. Pierwotna wersja miała ustaloną zbyt małą wartość `fatal_timeout`, co powodowało, że przy dużym obciążeniu i 2048 wątkach roboczych serwer restartował się pomimo braku zakleszczenia — z powodu zbyt długiego czasu oczekiwania.

Problem ten poprawiono, ograniczając liczbę wątków roboczych, co można uznać za rozwiązanie niskiej jakości, ponieważ dalej obarczone jest pewnymi odgórnymi założeniami, co do których nie ma pewności. Lepszym rozwiązaniem mogłoby być np. zastosowanie bardziej wyrafinowanego mechanizmu wykrywania zakleszczeń.

Wątek T1...Tn

```
void buf_flush_try_page()
{
  ...
  rw_lock(&lock);
  ...
  rw_unlock(&lock);
  ...
}
```

Wątek S1

```
void error_monitor_thread()
{
  ...
  if (lock_wait_time[i] > fatal_timeout)
    assert(0, "Crash, it seems to be hung");
  ...
}
```

Rysunek 1.11: Błąd doprowadzający do restartu systemu do zarządzania bazami danych *MySQL*. Pod dużym obciążeniem wątek monitorujący interpretował zaistniałą sytuację jako zatrzymanie serwera. Kod przykładu został istotnie uproszczony w celu zachowania przejrzystości.

1.2.5. Wnioski

Poniżej zebrano i skomentowano najważniejsze wnioski płynące z analizy błędów w programach współbieżnych. Należy przy tym pamiętać, że analiza opierała się wyłącznie na projektach typu *open source*, co może powodować, że zaprezentowane konkluzje nie mają zastosowania do oprogramowania zamkniętego. Można jednak z dużym prawdopodobieństwem przypuszczać, że zjawiska występujące w oprogramowaniu z otwartym źródłem mają zastosowanie także w oprogramowaniu własnościowym [101, 105].

Dla uzupełnienia, poza autorskimi wnioskami jakościowymi, przytoczono i opisano wnioski ilościowe z badania [69]. Cytowane postulaty zostaną wykorzystane w dalszych rozważaniach.

Jakościowe

Błędy w programach współbieżnych to realny problem. Jak pokazano na powyższych przykładach, z programowaniem współbieżnym wiąże się ryzyko powstawania błędów specyficznych dla tego paradygmatu. Co istotne, błędy te mają zupełnie inną naturę niż błędy w programach sekwencyjnych — występują tu zupełnie inne, niedeterministyczne, zależne od platformy zjawiska, inaczej się je naprawia, a także inny jest ich opis formalny. Błędy we współbieżności wymagają więc odmiennego i specjalnego traktowania, szczególnie w okresie intensywnego rozwoju dziedziny.

Niektóre skutki błędów w programach współbieżnych są podobne do skutków błędów znanych z programów sekwencyjnych. Błędy w programach współbieżnych doprowadzają do zakleszczeń bądź awarii systemów, mogą także być źródłem błędów bezpieczeństwa — podobnie jak w programach sekwencyjnych. Problemy częściej występujące w programach współbieżnych to natomiast obniżenie wydajności (np. na skutek aktywnego oczekiwania) oraz utrata bądź nadpisanie informacji (np. na skutek wyścigu). Skutki błędów w programach współbieżnych są więc zatem co najmniej tak samo poważne, jak skutki błędów w programach sekwencyjnych.

Manifestacja błędu współbieżnego bardzo często zależy od platformy programowej i sprzętowej. Wpływ na to, czy błąd się ujawnia, ma wiele czynników: architektura sprzętowa, liczba fizycznych i logicznych rdzeni, system operacyjny, tryb pracy planisty, stan środowiska programowego, obciążenie itd. Wiąże się to z koniecznością wyjątkowo solidnej weryfikacji kodu współbieżnego, a także

obszernych testów uwzględniających różne, wspomniane wyżej uwarunkowania. Pisanie poprawnego, przenośnego i uniwersalnego kodu współbieżnego wiąże się zatem z wnikliwą oceną poczynionych założeń.

Wystąpienie błędu współbieżnego często uwarunkowane jest przez zjawiska zewnętrzne wobec wykonującego się programu. Oprócz platformy wykonującej dany program, istotny jest również kontekst, czyli inne procesy współpracujące z rozważanym oprogramowaniem. Niektóre błędy mogą występować tylko w przypadku potencjalnie niekorzystnych zjawisk zewnętrznych np. jednoczesnej zmiany wspólnego pliku czy wadliwej komunikacji międzyprocesowej lub sieciowej. Wyizolowanie błędów tego typu jest bardzo trudne ze względu na skomplikowanie zachodzących zdarzeń i najczęściej wymagane jest do tego wsparcie systemu operacyjnego.

Niektóre błędy w programach współbieżnych wynikają z innych zjawisk niż niewłaściwe użycie zamków. Jak pokazano na przykładach, źródłem niewłaściwej synchronizacji procesów mogą być blokujące wywołania systemowe czy po prostu niewłaściwa kolejność instrukcji. Analizując programy współbieżne, nie należy więc koncentrować się wyłącznie na fragmentach związanych z zakładaniem zamków — należy analizować szerszy kontekst, tzn. cały kod wielowątkowy.

Stosowanie konkretnego paradygmatu programowania może mieć wpływ na bezpieczeństwo kodu wielowątkowego. Dogłębna analiza projektów pisanych w językach *C* oraz *C++* pokazała, że niektóre błędy występujące w programach napisanych w językach obiektowych są dużo trudniejsze do zidentyfikowania i poprawienia niż błędy w programach strukturalnych. Powodem takiego stanu rzeczy jest założenie o wzajemnej izolacji obiektów w programowaniu obiektowym, co utrudnia właściwą synchronizację w momencie, gdy obiekty te muszą ze sobą współpracować w sposób współbieżny. Ponadto można spotkać się z opiniami, jakoby niektóre paradygmaty programowania (jak np. programowanie funkcyjne) istotnie ułatwiały właściwą organizację i pisanie bezpiecznego kodu wielowątkowego (temat ten został podjęty w podrozdziale 3.4.2).

Konstrukcje semantyczne używanych platform programowych bywają niewystarczające do właściwego zrealizowania zamierzeń programisty. Jak pokazują przeanalizowane przykłady, często zdarza się, że wykorzystywane metody programowania nie oferują wystarczającej abstrakcji nad platformą sprzętowo–programową, którą obsługują. Programista często ma problemy z określeniem np. które operacje wykonają się w sposób niepodzielny, jakie skutki będzie miało dane wywołanie systemowe na innej niż testowa architekturze sprzętowej lub czy rozważane systemy operacyjne we właściwy sposób implementują deklarowany standard. Konieczność posiadania rozległej wiedzy i doświadczenia dotyczących programowania współbieżnego jest istotną barierą na drodze do poprawnego programu wielowątkowego. Wyżej wspomniane czynniki bardzo często przeszkadzają programiście w koncentrowaniu się na istocie problemu (czyli dostarczaniu funkcjonalności), zmuszając do opracowywania metod radzenia sobie z ograniczeniami środowiska.

Podobnym problemem są również bardzo ograniczone konstrukcje semantyczne wykorzystywanych narzędzi — w przedstawionym wcześniej (zob. rysunek 1.8, 1.9, 1.10), naprawianym kilkukrotnie błędzie w pakiecie oprogramowania *Mozilla* programista dobrze wiedział, jakie są jego zamierzenia, nie wiedział jednak, jak zrealizować je za pomocą wykorzystywanego języka i platformy programowej. Widać zatem, że dostarczane mechanizmy synchronizacji są ubogie i tym samym często

zmuszają do wykorzystywania prymitywów takich jak np. semafory, które nie zawsze pozwalają na łatwą realizację zamierzeń programisty.

Właściwe zidentyfikowanie problemu przy znanych wyłącznie skutkach jest bardzo trudne, a często wręcz niemożliwe. Analiza baz problemów pokazuje, że raporty zgłaszanych błędów bardzo często zawierają tylko skutek usterki np. kod błędu czy okoliczności awarii, a brakuje jakichkolwiek dodatkowych informacji jak choćby zrzutu pamięci. Identyfikacja przyczyny w takim wypadku jest zadaniem praktycznie niemożliwym i takie zgłoszenia są bardzo często zamykane z powodu braku możliwości reprodukcji. Zdarza się także, że błąd ujawnia się w sposób całkowicie niedeterministyczny i występuje na tyle rzadko, że programiści zniechęcają się do jego naprawy. Błędy w programach współbieżnych są zatem często całkowicie pomijane z powodu trudności, jakie sprawia diagnostyka. Potrzebne są więc nowe, lepsze metody wspierające testowanie i czynności diagnostyczne.

Naprawienie błędu już po wyizolowaniu i zrozumieniu przyczyny często sprawia problemy. Usterki w programach współbieżnych często są nietrywialne i zdarza się, że brakuje koncepcji, jak taki błąd naprawić. W niektórych sytuacjach usunięcie błędu wiąże się ze znaczną reorganizacją kodu, co jest bardzo niechętnie wykonywane przez programistę. Obie powyższe sytuacje powodują, że proponowane poprawki przybierają wiele potencjalnych postaci (często także niepoprawnych), zanim błąd zostanie ostatecznie naprawiony.

Praktycznie żadne narzędzia wspomagające wykrywanie i analizę błędów w programach współbieżnych nie są powszechnie wykorzystywane. W żadnym z przeanalizowanych raportów programiści nie wspominali o użyciu specyficznych dla dziedziny narzędzi wspomagających poszukiwanie czy naprawianie błędów w programach współbieżnych. W przypadku błędów w fragmentach sekwencyjnych programiści z powodzeniem wykorzystują programy takie jak *gdb* [31] czy *Valgrind* [118], jednak w przypadku błędów współbieżnych okazywały się one bezużyteczne. Sugeruje to, że nie istnieją wysokiej jakości narzędzia przypisane programowaniu współbieżnemu lub nie istnieje zwyczaj wykorzystywania takich rozwiązań (temat ten został podjęty w rozdziale 4).

Błędy często wynikają z niewiedzy programisty na temat działania programu, a także środowiska programowego i sprzętowego. Programiści nierzadko stosują optymistyczne założenia co do tworzonego przez siebie kodu, przez co jest on niedbale napisany. Na ogół brakuje im również dogłębnego zrozumienia, jak działają złożone mechanizmy takie jak np. wywołania zwrotne lub konkretne metody synchronizacji. Taki stan rzeczy jest spowodowany najprawdopodobniej niewystarczającą edukacją programistów lub niedogodnością istniejących technik programowania.

Błędy w programach współbieżnych częściej ujawniają się przy dużym obciążeniu. Przy dużej liczbie wątków, przestrzeń możliwych przeplotów jest intensywniej testowana, a więc błędne przebiegi są częściej wybierane.

Ilościowe

Poniższe postulaty zaczerpnięto w całości z [69].

97% błędów bez zakleszczeń, to błędy naruszenia kolejności lub naruszenia niepodzielności. Z tego powodu narzędzia wykrywające błędy w programach współbieżnych powinny być zorientowane na wykrywanie błędów tego typu.

32% błędów bez zakleszczeń, to błędy naruszenia kolejności, którym nie poświecono wiele uwagi w poprzednich badaniach. Brakuje narzędzi do wykrywania błędów naruszenia kolejności. Często zdarza się, że sekcje krytyczne są chronione odpowiednimi zamkami, jednak to nie gwarantuje właściwej kolejności ich wykonania, a jedynie brak niepożądanych przeplotów.

96% błędów ujawnia się w sposób deterministyczny przy konkretnym, częściowym przeplocie dokładnie dwóch wątków. Testowanie działania wątków parami może być zatem skuteczną techniką wykrywania błędów w programach współbieżnych. Cechuje się ona znacznie niższą złożonością przy wciąż wysokiej skuteczności.

22% błędów z zakleszczeniem występuje przy rekurencyjnym założeniu zamka przez dokładnie jeden wątek. Proste narzędzia detekcji zakleszczeń operujące tylko na jednym wątku mogą zatem okazać się bardzo skuteczne i pozwalają na łatwe wyeliminowanie błędów tego typu.

66% błędów bez zakleszczenia dotyczy dostępu do wyłącznie jednej zmiennej. Skoncentrowanie się na analizie dostępów do wyłącznie jednej zmiennej jest dobrym uproszczeniem analizy programów współbieżnych. Wiele aktualnych rozwiązań używa właśnie tej techniki.

34% błędów bez zakleszczenia dotyczy dostępu do wielu zmiennych. Nieznane są narzędzia analizujące skorelowane dostępy do wielu zmiennych — potrzebne są nowe mechanizmy tego typu.

97% błędów z zakleszczeniem występuje z powodu cyklicznej zależności w dostępie do najwyżej dwóch zasobów współdzielonych. Testowanie parami sekwencji zajęcie/zwolnienie zasobu na wyłącznie dwóch zasobach pozwala wykryć zdecydowaną większość błędów z zakleszczeniem.

92% błędów ujawnia się w sposób deterministyczny przy konkretnym, częściowym przeplocie co najwyżej czterokrotnego dostępu do pamięci. Testowanie częściowych przeplotów pomiędzy niewielkimi grupami dostępów do pamięci pozwala zredukować złożoność z wykładniczej do wielomianowej, przy zachowaniu bardzo wysokiej skuteczności.

73% błędów bez zakleszczenia należy poprawić poprzez zastosowanie techniki innej niż dodanie bądź zmienienie zamków. Nie istnieje złoty środek na poprawienie błędów we współbieżności. Narzędzia do wykrywania i diagnozy błędów powinny analizować szerszy kontekst niż wyłącznie zamki.

61% błędów z zakleszczeniem jest poprawianych przez usunięcie zajmowania jednego zamka w jednym wątku. Taka strategia jest prosta, jednak bardzo często prowadzi do innych błędów współbieżnych. Należy zachować szczególną ostrożność przy tego typu poprawkach.

30% błędów jest poprawianych za pierwszym razem w niewłaściwy sposób. Znaczna część błędów poprawiana jest w kilku łatach, ponieważ pierwsza po pewnym czasie okazuje się niewystarczająca. Często pierwotne podejście jedynie zmniejsza prawdopodobieństwo wystąpienia błędu, a nie eliminuje go całkowicie. Zdarza się też, że kolejne łaty wprowadzają nowe błędy. Programiści powinni przywiązywać większą uwagę do poprawek w kodzie.

39% błędów można poprawić przez zastosowanie pamięci transakcyjnej (ang. *transactional memory*). Pamięć transakcyjna jest zatem skutecznym narzędziem synchronizacji zadań, szczególnie takich, w których sekcje krytyczne są niewielkie i nie zawierają operacji wejścia/wyjścia. Należy jednak zauważyć, że pamięć transakcyjna jest wciąż bardzo rzadko wykorzystywana.

42% błędów można poprawić przez zastosowanie pamięci transakcyjnej z pewnymi dodatkowymi usprawnieniami. Pamięć transakcyjna rozszerzona o pewne konstrukcje semantyczne (np. `watch/retry`, `retry/orElse`) pozwala poprawić łącznie 81% błędów, co jest bardzo dobrym wynikiem.

19% błędów nie można poprawić przez zastosowanie wzorca pamięci transakcyjnej. Są to przede wszystkim błędy o rozbudowanych sekcjach krytycznych, zawierających operacje wejścia/wyjścia. Również nie wszystkie błędy naruszenia kolejności mogą być poprawione przez zastosowanie pamięci transakcyjnej.

32% błędów powoduje całkowite wyłączenie systemu, 35% zatrzymanie. Zatem ponad połowa błędów powoduje doprowadzenie aplikacji do stanu uniemożliwiającego jej użycie. Kontrola współbieżności jest więc istotnym elementem wiarygodności systemów.

1.3. Przyczyny i skutki błędów

Dla kompletności analizy, na zakończenie warto zastanowić się, jaka jest geneza błędów w programach współbieżnych oraz to, jakie są ich skutki. Obie kwestie były już częściowo poruszane przy okazji badania konkretnych błędów, jednak poniższy opis stanowi całościowe spojrzenie na zagadnienia i podkreśla tym samym istotę błędów w programach współbieżnych.

1.3.1. Przyczyny społeczne

Duże znaczenie dla poprawności programów współbieżnych ma czynnik ludzki. Pomimo że nasz świat jest współbieżny, to ludzki mózg zdecydowanie lepiej radzi sobie z tym, co sekwencyjne. Człowiek ma naturalną tendencję do liniowego porządkowania zadań i nie radzi sobie z przetwarzaniem informacji pochodzących z różnych kanałów jednocześnie czy też wykonywaniem wielu czynności na raz [107]. Współbieżna organizacja programów nie jest więc zgodna z naturalnym sposobem postrzegania świata [124].

Kolejnym istotnym czynnikiem są braki w edukacji informatycznej. Temat współbieżności, tak jak wspomniano we wstępie, był przez długi czas zapomniany w kręgach akademickich. W momencie, gdy dość niespodziewanie wróciła konieczność zajmowania się współbieżnością, okazało się, że brakuje dobrych naukowych i przemysłowych opracowań, sprawdzonych i przemyślanych wzorców projektowych, a także, w znacznym stopniu, algorytmów przystosowanych do programowania wielowątkowego. Nadal, podczas rozważania problemów tak podstawowych jak choćby sortowanie tablicy liczb, zwykle całkowicie pomija się zagadnienie zrównoleglenia wykonania algorytmu, co w obecnej sytuacji wydaje się znaczącym niedopatrzeniem. Można zatem w najbliższym czasie spodziewać się istotnych zmian w programach edukacji nowych programistów.

1.3.2. Przyczyny technologiczne

Poza problemami wynikającymi z ludzkiej natury, kłopoty sprawia również obecny stan technologii. Zdecydowana większość istniejących narzędzi została stworzona dla programowania sekwencyjnego i ich użyteczność jest istotnie ograniczona w programowaniu współbieżnym [69]. Niemal wszystkie rozwiązania zaprojektowane z myślą o programowaniu wielowątkowym zostały opracowane przez

środowiska akademickie, jednak bardzo często mechanizmy te nie wyszły poza fazę koncepcyjną. Faktycznie istniejące narzędzia nie zyskały natomiast znaczącej popularności w przemyśle. Tematyka narzędzi została obszernie opisana w rozdziale 4.

W programowaniu współbieżnym często nie pomagają również języki programowania i odpowiadające im biblioteki, które najczęściej nie były projektowane z zamysłem wykorzystania w programowaniu wielowątkowym. Tak jak pokazano wcześniej, środki wyrazu platform programistycznych bywają niewystarczające do zrealizowania z pozoru prostych zamierzeń programisty. Sytuacja ta zaczęła się od niedawna zmieniać i należy spodziewać się dynamicznego rozwoju języków programowania i całych środowisk programowych tak, aby nadążały one za technologicznym rozwojem wielordzeniowych architektur.

Ostatnim problemem technologicznym są skomplikowane zależności architekturowe, zarówno w warstwie sprzętowej jak i warstwie programowej, które nie zawsze są odpowiednio ukrywane przez abstrakcję w postaci modelu programowego. Programista nie zawsze wie, które operacje wykonają się w sposób niepodzielny, kiedy może zdarzyć się zmiana kolejności wykonania instrukcji z pozoru sekwencyjnych, czy zdarzają się stany chwilowej niespójności pamięci, jak zadania zostaną uszeregowane przez planistę, czy zastosowanie pewnej konstrukcji synchronizacyjnej doprowadzi do zjawiska inwersji priorytetów i wiele innych. Problemy specyficzne dla sprzętu i środowiska programowego są inne dla każdej platformy i jest to zjawisko wysoce niepożądane, gdyż wymaga oddzielnej specjalizacji dla dowolnej, wykorzystywanej kombinacji. Sytuacja ta może ulec dalszemu pogorszeniu, gdyż najlepiej znana i opisana architektura sprzętowa *x86* powoli traci wieloletnią hegemonię, a na jej miejsce wchodzą inne, mniej znane platformy.

1.3.3. Skutki

Skutki błędów w programowaniu współbieżnym są podobne do tych znanych z programów sekwencyjnych. Najczęstszym następstwem błędu w programie współbieżnym jest destabilizacja lub całkowite zatrzymanie działania systemu lub aplikacji [69]. Co istotne, podobnie jak w przypadku programowania sekwencyjnego, błędy we współbieżności mogą skutkować poważnymi błędami bezpieczeństwa. Ze względu na ważność tematyki, a także bardzo niewielką liczbę istniejących opracowań, błędy bezpieczeństwa w programowaniu współbieżnym zostały obszernie przeanalizowane w rozdziale 2.

Specyfika błędów w programach współbieżnych wiąże się również z charakterystycznymi skutkami, które nie występują tak często w programowaniu sekwencyjnym. Typowym następstwem jest zniszczenie informacji (np. z powodu przypadkowego nadpisania danych) lub utrata spójności danych (np. w wyniku niezserializowanych zapisów). Skutki tego typu są zwykle oceniane jako dużo bardziej szkodliwe niż typowe, powodujące zatrzymanie działania systemu, ponieważ przywrócenie bądź odtworzenie odpowiedniej struktury informacji jest często bardzo kosztowne i czasochłonne. Innym, typowym dla programowania współbieżnego skutkiem błędu, jest degradacja wydajności. Zastosowanie niewłaściwej synchronizacji w dostępie do zasobów współdzielonych może doprowadzić do powstania wąskiego gardła i istotnie spowolnić działanie całego systemu. Zjawisko to występuje bardzo często w rozwiązaniach stosujących blokowanie pesymistyczne lub nadmiarowe.

1.4. Podsumowanie

Jak pokazano w niniejszym rozdziale, programowanie współbieżne jest niezwykle trudne, a tym samym podatne na błędy. Usterki w programach współbieżnych są co najmniej tak samo poważne, jak w tradycyjnych programach sekwencyjnych. Ich natura jest jednak zupełnie odmienna, stąd dla błędów w programach współbieżnych stworzono oddzielną klasyfikację i ich badanie stanowi niezależną od badania błędów w programowaniu sekwencyjnym dziedzinę nowoczesnej inżynierii oprogramowania.

Analiza konkretnych błędów w znanych i powszechnie używanych programach pozwala zidentyfikować i zrozumieć najpoważniejsze problemy występujące w programowaniu współbieżnym: zakleszczenia, naruszenia kolejności oraz naruszenia niepodzielności. Źródłem błędów tych typów jest przede wszystkim niedbałość i niewiedza programistów dotycząca platformy sprzętowo–programowej, dla której tworzą oprogramowanie. Istniejące błędy w programach współbieżnych są natomiast bardzo często trudne do naprawienia, ich identyfikacja jest zaś zadaniem niezwykle trudnym ze względu na niedeterminizm zjawisk zachodzących w aplikacjach tego typu.

Narzędzia dla programowania współbieżnego wciąż są zdecydowanie mniej dojrzałe, a tym samym rzadziej używane niż sekwencyjne odpowiedniki. Istnieje duże zapotrzebowanie na lepsze — skuteczniejsze i łatwiejsze w użyciu rozwiązania w tej dziedzinie. Zaawansowane, nowoczesne platformy programistyczne i języki programowania dostosowane do programowania współbieżnego już istnieją, rzadko są jednak wdrażane w środowiska produkcyjne, głównie ze względu na mentalność programistów.

Przyczynami błędów w programach współbieżnych są problemy zarówno po stronie programistów, jak i dostępnych technologii. Konieczna jest więc intensywna i postępowa edukacja w zakresie programowania wielowątkowego, wspomagana przez nowoczesne i ekspresyjne technologie pozwalające swobodnie tworzyć poprawne programy i w razie konieczności łatwo rozpoznawać pojawiające się błędy.

Skutki błędów w programach współbieżnych nieco przypominają te znane z programowania sekwencyjnego. Nawet niepozorne usterki mogą w efekcie doprowadzić do awarii całego systemu. Błędy w programowaniu współbieżnym częściej niż w programowaniu sekwencyjnym powodują utratę lub zniszczenie struktury danych, a także znaczące problemy z wydajnością. Niezwykle istotną, lecz bardzo słabo zbadaną tematyką, są błędy bezpieczeństwa wynikające z niepoprawnego kodu współbieżnego. Z tego względu zagadnienie to zostało opisane w kolejnym rozdziale.

2. Błędy bezpieczeństwa w programowaniu współbieżnym

Błędy w oprogramowaniu, obok metod socjotechnicznych, są podstawowym źródłem naruszeń bezpieczeństwa systemów [93]. Pomimo wieloletnich badań nad podatnościami w oprogramowaniu, ich liczba nie wydaje się zmniejszać, jednak można zaobserwować wzrost jakości mechanizmów zabezpieczających [5]. Podobnie jak w przypadku usterek opisanych w rozdziale 1, w zasadzie wszystkie dotychczasowe badania poświęcono analizie błędów sekwencyjnych prowadzących do naruszenia bezpieczeństwa, zupełnie ignorując przy tym zagadnienie wpływu współbieżności na powstawanie zagrożeń. Pomimo, że o korelacji współbieżności i bezpieczeństwa pisano już w latach siedemdziesiątych XX wieku [1, 12], to faktyczną uwagę na problem zwróciła dopiero publikacja Roberta Watsona z 2007 roku [121] dotycząca błędów bezpieczeństwa w systemach operacyjnych wynikających właśnie ze współbieżnego wykonania kodu. Wciąż brakuje jednak wyczerpujących opracowań opisujących temat z szerszej perspektywy — większość prac powstałych w ostatnim czasie analizuje dogłębnie pojedyncze przypadki, nie przedstawiając przy tym uogólnionych wniosków, które mogłyby przysłużyć się zwiększeniu bezpieczeństwa programów współbieżnych.

W obliczu powyższych spostrzeżeń, a także wniosków uzyskanych w badaniu opisanym w rozdziale 1, w tej części pracy skoncentrowano się na analizie błędów w programowaniu współbieżnym, które prowadzą do powstania błędów bezpieczeństwa. W szczególności podjęto próbę odpowiedzenia na następujące pytania:

— Czy błędy bezpieczeństwa w programowaniu współbieżnym są realnym problemem?
— Co powoduje, że z pozoru zwykły błąd staje się podatnością?
— Jakie czynniki wpływają na stopień trudności wykorzystania podatności?
— Jak współbieżność wpływa na skuteczność istniejących technik wykrywania podatności i obrony przed atakami?

Aby odpowiedzieć na pierwsze z postawionych pytań, na początku niniejszego rozdziału krótko wprowadzono teorię błędów bezpieczeństwa w programach współbieżnych, a następnie, podobnie jak w rozdziale 1 przeanalizowano błędy w istniejących aplikacjach współbieżnych. W kolejnym podrozdziale przedyskutowano problem rozpoznawania i wykorzystywania podatności, żeby uzyskać odpowiedź na trzecie i czwarte pytanie. Odpowiedzi na ostatnie pytanie udzielono w przedostatnim podrozdziale, który podejmuje kwestię technik wykrywania podatności i obrony przed atakami — zarówno istniejących, jak i tych, które mogą powstać w przyszłości. Rozdział kończy się krótkim podsumowaniem zawierającym najważniejsze wnioski z przeprowadzonego badania.

2.1. Charakterystyka błędów bezpieczeństwa w programach współbieżnych

Technika badania błędów bezpieczeństwa jest istotnie różna od analizy np. błędów funkcjonalnych. W przypadku bezpieczeństwa konieczne jest badanie znacznie szerszego kontekstu występowania błędu, ponieważ oprócz kodu, może on być spowodowany błędnie zdefiniowanymi procesami biznesowymi lub zwykłymi słabościami ludzkimi. Oprócz technologii, istotne jest zatem, w jakim środowisku rozważany system działa.

W bezpieczeństwie analiza przyczynowa ma zwykle na celu zdefiniowanie nowych środków zaradczych, które w przyszłości mogą powstrzymać powstawanie podobnych błędów. Można jednak stwierdzić, że w tej dziedzinie od samych przyczyn dużo istotniejsze są skutki, wymagające opracowania metod obsłużenia sytuacji wyjątkowych, które nierzadko pochłaniają znaczną część zasobów ludzkich i finansowych. Jest to spowodowane tym, że to właśnie konsekwencje błędów bezpieczeństwa mają istotny wpływ na powodzenie przedsięwzięcia informatycznego. Kompletna analiza przyczynowo–skutkowa pozwala zatem na zdefiniowanie środków przeciwdziałania czynnikom sprawczym, a także określenie sposobów łagodzenia efektów, co w rezultacie pozwala określić kluczową rzecz, jaką jest ryzyko. W idealnym przypadku system informatyczny powinien być wolny od wszelkich błędów bezpieczeństwa. W praktyce jednak nierzadko wykonuje się kalkulacje oparte na prawdopodobieństwie przeprowadzenia ataku oraz kosztach zabezpieczeń i skutków i na tej podstawie określa się dopiero, czy zasadne jest wdrożenie rozważanych środków bezpieczeństwa.

Zasady opisane powyżej dotyczą także błędów bezpieczeństwa w programach współbieżnych. Analiza ma na celu poznanie okoliczności w jakich dany błąd występuje i jakie jest tego realne prawdopodobieństwo. Następnie należy określić, jakie są pośrednie i bezpośrednie przyczyny błędu, a także jakie skutki pociąga za sobą skuteczny atak na rozważaną podatność.

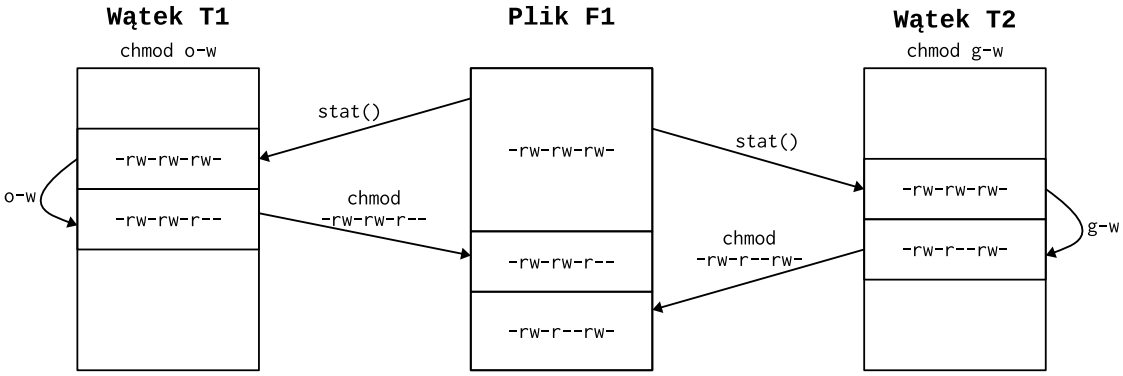
2.1.1. Klasyfikacja błędów bezpieczeństwa we współbieżności

Dla ułatwienia opisu błędów bezpieczeństwa w programowaniu współbieżnym, podobnie jak dla błędów bezpieczeństwa w programowaniu sekwencyjnym, zaproponowano ich różne klasyfikacje. Tak jak wspomniano w podrozdziale 1.1 odpowiednie pogrupowanie błędów umożliwia łatwiejszy ich opis, a także ułatwia zrozumienie natury błędów danego typu.

Pierwsza proponowana klasyfikacja [122] została zaczerpnięta wprost z terminologii dotyczącej ataków na oprogramowanie [5] i dzieli błędy na:

— **Pasywne** — błędy w oprogramowaniu współbieżnym, w wyniku których w sposób spontaniczny dochodzi do ujawnienia informacji lub nadania zbyt wysokich uprawnień. Przykład takiego błędu przedstawia rysunek 2.1.

— **Aktywne** — błędy w oprogramowaniu współbieżnym, które umożliwiają przeprowadzenie ataku, prowadzącego do pozyskania informacji, uprawnień bądź umożliwiającego spowodowanie awarii systemu. Przykład takiego błędu przedstawia rysunek 2.2.



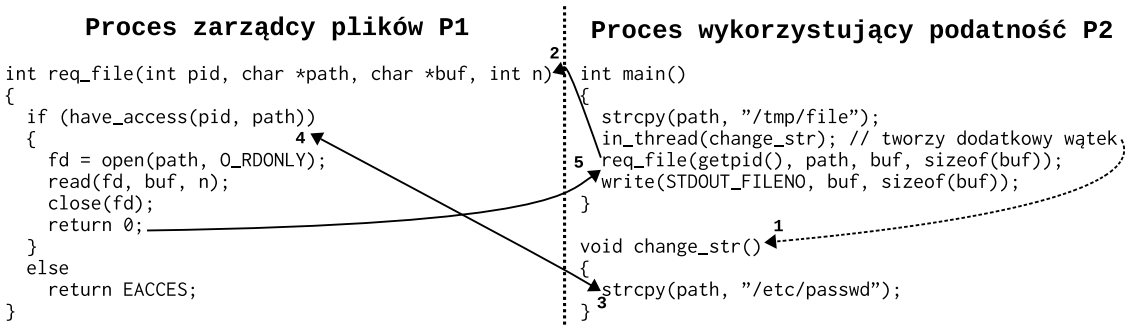
Rysunek 2.1: Przykład pasywnego błędu bezpieczeństwa w programie współbieżnym. Dwa niezależne wątki mają za zadanie usunąć uprawnienia do zapisu do pliku — jeden dla aktualnej grupy, drugi dla pozostałych użytkowników. Oba wątki niezależnie odczytują za pomocą wywołania `stat()` aktualne uprawnienia, modyfikują je, a następnie zapisują. Wywołania `stat()` i `chmod()` nie mogą jednak zostać użyte w sposób atomowy, co powoduje, że wszystkie wywołania kończą się sukcesem, ale tylko działania wątku *T2* zostają uwzględnione. Finalnie błąd ten powoduje, że plik posiada niewłaściwe, zbyt pobłażliwe uprawnienia — pozostali użytkownicy mają prawa zapisu. Opracowano na podstawie: [122].

W tej klasyfikacji zwraca się zatem uwagę na to, czy dany błąd wymaga aktywnego działania i wysiłku ze strony atakującego na drodze do osiągnięcia celu. W ogólności błędy pasywne są uważane za błędy poważniejsze, gdyż nawet niedoświadczona osoba jest w stanie taki błąd wykorzystać.

Kolejna proponowana w literaturze [124] klasyfikacja dzieli błędy bezpieczeństwa w programach współbieżnych ze względu na warstwę systemu, w której występuje:

— **Pamięć** — błędy bezpieczeństwa w oprogramowaniu współbieżnym, które spowodowane są naruszeniem spójności pamięci.
— **System plików** — błędy bezpieczeństwa w oprogramowaniu współbieżnym, które wynikają z nieatomowych operacji w systemie plików.
— **Fizyczna interakcja** — błędy bezpieczeństwa w oprogramowaniu współbieżnym, które ujawniają się przy bezpośredniej interakcji z systemem za pomocą interfejsu użytkownika.

Autorzy niniejszej klasyfikacji oszacowali ponadto, że błędy dotyczące pamięci stanowią ok. 65%, błędy spowodowane niewłaściwą obsługą systemu plików ok. 28%, a błędy wynikające z niewłaściwej obsługi interfejsu użytkownika ok. 7%. W badaniu [124] podkreśla się również, że przez lata błędy bezpieczeństwa w programach współbieżnych były kojarzone wyłącznie z błędami typu *time-of-check-to-time-of-use, TOCTOU* (zob. rysunek 2.1 oraz 2.2) w systemach plików [97, 114, 115, 123] i z tego powodu większość istniejących mechanizmów bezpieczeństwa koncentruje się właśnie na tym problemie. Oszacowane na podstawie analizy wielu błędów rozkłady procentowe pokazują natomiast, że rozważenie błędów dotyczących pamięci jest co najmniej tak samo ważne, jak błędów wynikających z nieatomowych operacji na plikach.



Rysunek 2.2: Przykład aktywnego błędu bezpieczeństwa w programie współbieżnym. Proces atakującego próbuje początkowo uzyskać dostęp do pliku `/tmp/file`, jednak nim złoży żądanie tworzy dodatkowy wątek `change_str()` (1), który ma za zadanie we właściwym momencie podmienić przesłaną ścieżkę. Po przesłaniu żądania dostępu do wspomnianego wcześniej pliku (2), atakujący czeka, aż proces zarządcy sprawdzi uprawnienia dostępu po to, aby natychmiast (3) podmienić ścieżkę dostępu na ścieżkę chronionego pliku `/etc/passwd`. Gdy proces nadzorcy kontynuuje odczyt pliku (4), w zmiennej `path` znajduje się już ścieżka do pliku chronionego i po jego odczycie następuje zwrócenie wyniku atakującemu (5). W ataku tym kluczowe jest podmienienie ścieżki w odpowiednim momencie (po sprawdzeniu dostępu) i aby było to możliwe, zmienna `path` musi znajdować się w pamięci współdzielonej.

Niniejsza klasyfikacja może się wydawać niekompletna lub zbyt gruboziarnista, jednak trudno dla tak złożonego tematu jak bezpieczeństwo programów współbieżnych zaproponować grupowanie błędów, które będzie jednocześnie uniwersalne i czytelne. Dodatkowo warto wspomnieć, że w literaturze nie zaproponowano klasyfikacji innych niż przedstawione powyżej. Z tego powodu w dalszej części badania używane będą przedstawione wyżej techniki opisu i klasyfikacji.

2.2. Analiza przykładowych błędów bezpieczeństwa

Aby móc skutecznie bronić się przed atakami na błędy w programach współbieżnych, należy dobrze zrozumieć ich naturę. Z tego powodu przeanalizowano rzeczywiste błędy w popularnych i powszechnie używanych aplikacjach. Badanie oparto na bazie błędów bezpieczeństwa *CVE* (ang. *Common Vulnerabilities and Exposures*) [80], która stanowi zestandaryzowany sposób publikowania informacji o tychże błędach. Baza ta była przeszukiwana pod kątem standardowych słów kluczowych dotyczących programowania współbieżnego. W toku badania korzystano również z prac [122, 124].

Poniżej zaprezentowano i skomentowano niektóre z przeanalizowanych błędów, kategoryzując je przy tym ze względu na warstwę występowania. Zaprezentowane błędy wybrano tak, aby oprogramowanie, którego dotyczą, stanowiło różne elementy systemu informatycznego — od systemu operacyjnego do zwykłej aplikacji użytkowej.

2.2.1. Błędy w warstwie pamięci

Pierwszym prezentowanym typem podatności są błędy w warstwie pamięci operacyjnej. Zgodnie z badaniem [124] są one najliczniejszą grupą stanowiącą 65% wszystkich błędów bezpieczeństwa w programowaniu współbieżnym.

Linux uselib() CVE–2004–1235

Pierwszym demonstrowanym błędem (zilustrowanym na rysunku 2.3) jest błąd w kodzie ładującym formaty binarne w jądrze systemu *Linux*, który pozwalał lokalnym użytkownikom uzyskać uprawnienia superużytkownika [96].

```
static int load_elf_library(struct file *file)
{
  down_write(&current->mm->mmap_sem);
  error = do_mmap(file, ...);
  up_write(&current->mm->mmap_sem);
  ...
  if (bss > len)
    do_brk(len, bss - len);
  ...
}

unsigned long do_brk(unsigned long addr, unsigned long len)
{
  ...
  vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
  ...
  vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
  ...
  vma_link(mm, vma, prev, rb_link, rb_parent);
  ...
}
```

Rysunek 2.3: Błąd CVE–2004–1235 w jądrze *Linuksa* pozwalający na uzyskanie uprawnień superużytkownika, wynikający z braku synchronizacji wywołania funkcji `do_brk()`.

Aby załadować plik formatu `ELF` do pamięci, proces wykonywał wywołanie systemowe `uselib()`, które wykorzystywało procedurę `load_elf_binary()`. W pierwszej fazie funkcja ta prawidłowo wykonywała podprogram `do_mmap()` modyfikujący mapę pamięci w sekcji krytycznej, jednak przy procedurze `do_brk()` tego samego typu nie zastosowano żadnej synchronizacji. Wywołanie `do_brk()` współbieżnie, pod warunkiem, że funkcja `kmem_cache_alloc()` doprowadzi do uśpienia jednego wątku, może spowodować, że stworzony deskryptor `VMA` zostanie umieszczony w złym miejscu w mapie pamięci procesu, co w zupełności wystarcza, aby uzyskać uprawnienia superużytkownika. Dokładny opis odpowiedniego sposobu wykorzystania podatności wraz z przykładowym kodem można odnaleźć w [96].

Bezpośrednią przyczyną wystąpienia błędu jest niezastosowanie mechanizmu wzajemnego wykluczania przy wywołaniu `do_brk()`, co było najprawdopodobniej zwykłym przeoczeniem programisty. Skutek tego błędu jest najpoważniejszy z możliwych w przypadku systemu operacyjnego — szansa na przejęcie przez użytkownika lokalnego całkowitej kontroli nad maszyną. Problematyczną kwestią przy wykorzystaniu tej podatności jest zmuszenie `kmem_cache_alloc()` do uśpienia wątku wcześniejszego tak, aby późniejszy wygrał wyścig doprowadzając do uszkodzenia

mapy pamięci. Okazało się, że jest to jednak stosunkowo łatwe — wystarczy próbować załadować plik `ELF` z dostatecznie dużą sekcją `DATA`.

glibc setuid RHBA–2009:1634–1

Kolejny opisywany błąd (zilustrowany na rysunku 2.4) dotyczył biblioteki standardowej języka *C glibc* i stwarzał możliwość wycieku uprawnień.

Specyfikacja *POSIX* wymaga, aby przy wywołaniu systemowym `setuid()` zmieniającym identyfikator właściciela procesu, wszystkie wątki otrzymały ten sam identyfikator. Wywołanie `setuid()` musi być więc powtórzone tyle razy, ile wątków zawiera aktualny proces, ponieważ w systemie *Linux* każdy wątek posiada przypisany własny identyfikator właściciela. W bibliotece *glibc* zostało to zaimplementowane przy pomocy funkcji `__nptl_setxid()`, która iteruje przez listę wątków i sygnalizuje im zmianę identyfikatora użytkownika.

```
int __nptl_setxid(struct xid_command *cmdp)
{
  lll_lock(stack_cache_lock);
  list_for_each (runp, &stack_used)
  {
    struct pthread *t = list_entry(runp, struct pthread, list);
    if (t == self)
      continue;
    setxid_signal_thread(cmdp, t);
  }
  lll_unlock(stack_cache_lock);
}

static int allocate_stack(...)
{
  ...
  lll_lock(stack_cache_lock);
  list_add(&pd->list, &stack_used);
  lll_unlock(stack_cache_lock);
  ...
}
```

Rysunek 2.4: Błąd RHBA–2009:1634–1 w bibliotece *glibc* powodujący wyciek uprawnień w przypadku, gdy przy wywołaniu `setuid()` był tworzony nowy wątek.

Jeśli w trakcie wykonywania `__nptl_setxid()` tworzony był nowy wątek (funkcja `allocate_stack()`), to otrzymywał on stary identyfikator właściciela, który nie podlegał aktualizacji, ponieważ z powodu synchronizacji nie był on w odpowiednim momencie włączany do listy istniejących wątków. Jeżeli taka sytuacja miała miejsce w aplikacji uruchamianej z wysokimi uprawnieniami, które następnie były obniżane (co jest standardową techniką w przypadku demonów), to powodowało to pasywny błąd wycieku uprawnień. Atakujący, który przejąłby kontrolę nad wątkiem z podwyższonymi uprawnieniami, posiadałby zatem dużo większe możliwości niż te, które teoretycznie przysługiwałyby mu z kodu aplikacji.

OpenSSH signal handler CVE–2006–5051

Wątki nie są jedynym źródłem współbieżności w aplikacjach — innym, równie problematycznym, mogą być sygnały. Pomimo powszechnego poglądu na temat

trudności z właściwą obsługą sygnałów standardu *POSIX* [64], kwestie z tym związane są bardzo często całkowicie ignorowane przez programistów. Pierwszym źródłem nieporozumień jest asynchroniczna natura sygnałów — programiści mają problemy ze zrozumieniem, kiedy, w jakich okolicznościach i do którego wątku sygnał może zostać dostarczony. Następną kłopotliwą kwestią jest to, jakie operacje mogą być wykonywane w procedurze obsługi sygnału. Kolejne wersje standardu *POSIX* publikowane przez *The Open Group* [91] dokładnie definiują, które funkcje mogą być bezpiecznie wywoływane podczas obsługi sygnału (ang. *async–signal–safe functions*), a wynik działania pozostałych procedur uważa się za niezdefiniowany. Programiści nierzadko ignorują jednak te zalecenia, co często prowadzi do błędów w programach.

Źródłem poważnego błędu bezpieczeństwa (zilustrowany na rysunku 2.5) w krytycznie istotnym oprogramowaniu, jakim jest *OpenSSH*, było właśnie wywołanie funkcji `free()`, która nie jest zaliczana do grupy *async–signal–safe functions*, ponieważ wykonuje ona operacje na danych współdzielonych. Skuteczny atak na ten błąd polegał na wielokrotnym spowodowaniu zgłoszenia sygnału `SIGALARM` w krótkim czasie, co powodowało wielokrotne wywołanie funkcji `free()` zwalniającej ten sam obszar pamięci. Takie działanie doprowadzało do sytuacji określanej jako *double free*, która skutkuje błędem przepełnienia bufora (ang. *buffer overflow*) [92] lub awarią aplikacji. Przy okazji omawiania tego błędu warto zwrócić uwagę na komentarz jaki znalazł się w kodzie procedury obsługi sygnału — programista piszący ten fragment najprawdopodobniej był świadomy przynajmniej części problemów, jakie może spowodować tworzony przez niego kod, jednak nie wiedział, jak go naprawić. Ostatecznie z omawianej funkcji usunięto wywołanie podprogramu zwalniającego zasoby i zastąpiono go zwykłym zamknięciem aplikacji poprzez `_exit(SIGALARM)`.

```
void grace_alarm_handler(int sig)
{
  /* XXX no idea how fix this signal handler */

  if (use_privsep && pmonitor != NULL && pmonitor->m_pid > 0)
    kill(pmonitor->m_pid, SIGALRM);

  cleanup_exit(SIGALRM);
}

int main(int ac, char **av)
{
  ...
  signal(SIGALRM, grace_alarm_handler);
  ...
}
```

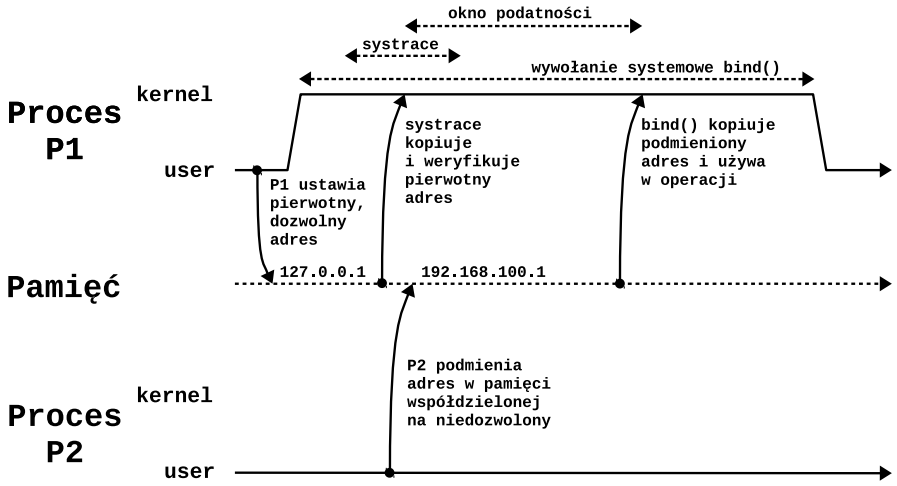
```
{
  ...
  free(...);
  ...
}
```

Rysunek 2.5: Błąd CVE–2006–5051 w *OpenSSH* umożliwiający ataku typu *DoS* oraz wykonanie kodu.

Opisywany błąd bezpieczeństwa w *OpenSSH* oceniono jako krytyczny i przyznano mu ocenę istotności 9.3/10 [23]. Warto nadmienić, że podobne błędy odnaleziono w innych popularnych aplikacjach takich jak *sendmail* [126] oraz *traceroute* [27]. Jest to zatem kolejny przykład tego, że błędy we współbieżności nie ograniczają się tylko i wyłącznie do niewłaściwego zarządzania wątkami.

Systrace bypass

Podatności występują również w oprogramowaniu, które ma zwiększać bezpieczeństwo — kolejny prezentowany błąd (zilustrowany na rysunku 2.6) dotyczy mechanizmu filtrowania wywołań systemowych *systrace* [104]. Podstawowym zadaniem *systrace* jest umożliwienie twórcy aplikacji zdefiniowania zasad dostępu do wywołań systemowych na podstawie parametrów tak, aby atakujący, w razie przejęcia kontroli nad aplikacją, miał ograniczone do minimum możliwości wyrządzenia szkód. Podczas wykonywania wywołania systemowego *systrace* musi zatem sprawdzić wszystkie przekazane przez użytkownika parametry i na tej podstawie podjąć decyzję, czy wywołanie należy kontynuować. Przy projektowaniu omawianego mechanizmu podjęto jednak wyjątkowo niefortunną decyzję, aby przekazywany przez użytkownika argument był kopiowany do przestrzeni jądra dwukrotnie: najpierw w celu weryfikacji polityki przez *systrace*, a następnie już przy samym wywołaniu systemowym. Pomiędzy tymi dwiema operacjami kopiowania pozostaje okno czasowe, w którym dodatkowy proces czy wątek może podmienić zawartość pamięci. W ten sposób inne argumenty są weryfikowane, a na innych faktycznie następuje wykonanie operacji systemowej — dochodzi do aktywnego błędu bezpieczeństwa pozwalającego ominąć omawiany mechanizm.



Rysunek 2.6: Błąd bezpieczeństwa w *systrace* pozwalający na ominięcie weryfikacji parametrów wywołania systemowego. Opracowano na podstawie: [121].

Przedstawiony błąd jest klasycznym błędem naruszenia niepodzielności *TOCTOU*, występujący tym razem w pamięci, a nie w systemie plików. Dokładniejsza analiza pokazała, że nie tylko *systrace* posiada wyżej opisaną podatność. Dotyczy ona także mechanizmów takich jak *GSWTK* oraz *CerbNG* [121]. Odkrycie to na długi czas zahamowało badania i rozwój technik filtrowania wywołań systemowych.

Moonlight CVE–2011–0990

Ostatnią prezentowaną podatnością w warstwie pamięci jest błąd (zilustrowany na rysunku 2.7) we wtyczce *Moonlight* będącej portem *Silverlight* dla platformy *Mono*. W celu przyspieszenia działania metody `Array.Copy()` kopiującej zakres z tablicy źródłowej do docelowej, funkcja realizująca ją najpierw sprawdza zgodność typów elementów, a następnie wykonuje szybkie kopiowanie za pomocą `memcpy()`

zamiast używać powolnych instrukcji maszyny wirtualnej *CLR* (ang. *Common Language Runtime*).

```
bool FastCopy(MonoArray *src, MonoArray *dst, int length)
{
  ...
  for (i = 0; i < length; ++i)
    if (!safe_cast(type_of(src[i]), type_of(dst[i])))
      return FALSE;

  for (i = 0; i < length; ++i)
    memcpy(dst[i], src[i], size_of(objPtr));

  return TRUE;
}
```

Rysunek 2.7: Błąd bezpieczeństwa w *Moonlight* pozwalający na zmodyfikowanie wewnętrznej struktury wtyczki, ucieknięcie z *sandboksa*, wykonanie ataku typu przepełnienie bufora lub *DoS*. Kod uproszczono w celu zwiększenia czytelności. Opracowano na podstawie: [124].

Ze względu na podzielność operacji sprawdzenia typu i kopiowania, atakujący może zaraz po wykonaniu pierwszej czynności podmienić wartość wskazywaną przez `src` na taką, która ma niezgodny z `dst` typ. W konsekwencji możliwe jest zmodyfikowanie wewnętrznej struktury wtyczki, opuszczenie bezpiecznego środowiska wykonania poprzez modyfikację menadżera bezpieczeństwa, spowodowanie awarii, a być może nawet wykonanie wrogiego kodu.

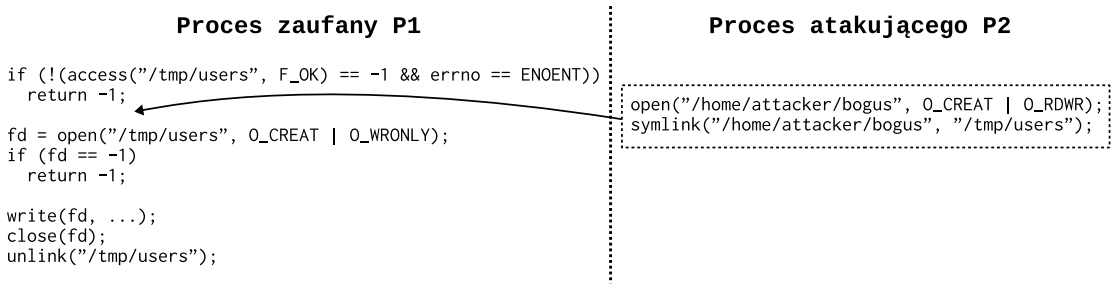
2.2.2. Błędy w warstwie systemu plików

Kolejny omawiany typ błędów, to klasyczne, znane od dawna, błędy *TOCTOU* występujące w systemie plików. Zgodnie z badaniem [124], stanowią one 28% wszystkich błędów bezpieczeństwa w programowaniu współbieżnym.

Symlink race

Najpopularniejszymi współbieżnymi błędami bezpieczeństwa w warstwie plików są podatności określane wspólnym mianem *symlink race*. Usterki tego typu pozwalają uzyskać atakującemu dostęp do zawartości poufnych plików tymczasowych, stworzonych (najczęściej w katalogu `/tmp`) w niewłaściwy sposób. W wypadku tego błędu omówiono ogólny przebieg wykorzystania podatności (zilustrowany na rysunku 2.8).

Zaufany proces (działający z prawami superużytkownika) w pierwszym kroku sprawdza istnienie pliku tymczasowego `/tmp/users`, który ma za zadanie przechowywać poufne dane. Po pomyślnym zweryfikowaniu braku pliku w systemie plików, proces przystępuje do operacji stworzenia i otworzenia takiego pliku. Nim to się stanie atakujący tworzy dowiązanie symboliczne o takiej samej ścieżce dostępu jak plik, który zaraz zostanie stworzony, a prowadzące do odpowiedniego miejsca w swoim folderze domowym. Po wykonaniu tych operacji wywołanie `open()` w procesie zaufanym nie stworzy nowego pliku, a jedynie otworzy plik podstawiony przez atakującego. Wszystkie dane, które zostaną następnie zapisane do tego pliku, będą możliwe do odczytu przez atakującego. Co więcej, wywołanie `unlink()` z końca programu zaufanego nie usunie właściwego pliku, a jedynie dowiązanie symboliczne.



Rysunek 2.8: Uogólniony przebieg błędu typu *symlink race*, w którym atakujący uzyskuje możliwość dostępu do zawartości poufnych plików tymczasowych.

Atak, który przebiega w ten sposób, pozwala na uzyskanie dostępu do danych poufnych, jednak nie jest to jedyne niebezpieczeństwo, jakie występuje w przypadku tego błędu. W innych wariantach, w których atakujący uruchamia proces z ustawionym bitem `setuid`, intruz może stworzyć dowiązanie symboliczne do istniejącego, istotnego pliku systemowego (np. pliku w `/etc/cron.d/`) i jeśli jest w stanie kontrolować dane, które są do tego pliku zapisywane, to może zmodyfikować konfigurację systemu.

Zapytanie bazy błędów *CVE* [80] o frazę */tmp* zwraca 280 wyników[1], co doskonale obrazuje skalę błędów bezpieczeństwa bezpośrednio powiązanych z plikami tymczasowymi. Warto nadmienić, że na przestrzeni lat zaproponowano wiele schematów postępowania w przypadku tworzenia poufnych plików tymczasowych, które mimo wszystko bardzo często są ignorowane przez programistów. Przede wszystkim standard *POSIX* oferuje funkcję `mkstemp()`, która pozwala w bezpieczny (tj. wolny od opisywanego błędu) stworzyć unikalny plik tymczasowy, z gwarancją wyłącznej własności. Inne, acz mniej wygodne podejścia, polegają miedzy innymi na wykorzystywaniu wywołania `open()` z parametrami `O_EXCL | O_CREAT`, co zapewnia, że plik pod podaną ścieżką nie istniał przed faktycznym stworzeniem.

Częste przypadki błędów typu *symlink race*, pomimo istniejących poprawnych schematów, doprowadziły do tego, że po 16 latach od wykrycia problemu [18] zdecydowano się na wprowadzenie do jądra *Linuksa* łaty [17], która uniemożliwia stworzenia pliku tymczasowego w niebezpieczny sposób. W tym celu łata zezwala na podążanie za dowiązaniami tylko, gdy:

— dowiązanie znajduje się poza katalogiem oznaczonym jako *sticky* z globalnymi prawami zapisu,
— podążający za dowiązaniem i właściciel dowiązania to ten sam użytkownik,
— właściciel katalogu zawierającego jest taki sam jak właściciel dowiązania.

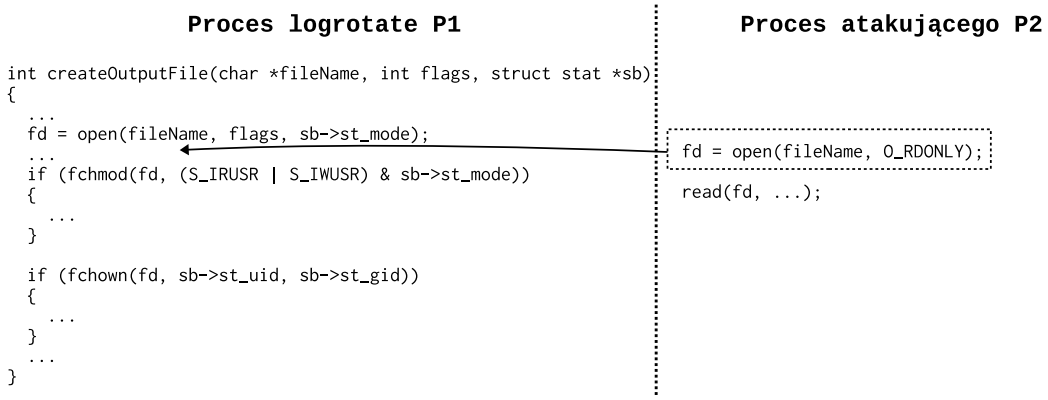
Warto zauważyć, że rozwiązanie to narusza standard *POSIX* i stwarza możliwość niewłaściwego działania niektórych aplikacji, jednak ani argument podążania za wadliwą specyfikacją, ani argument zgodności wstecznej z potencjalnie niebezpiecznymi aplikacjami nie był wystarczający, aby powstrzymać wdrożenie łaty do głównej linii *Linuksa* [17].

logrotate CVE–2011–1098

Niektóre z błędów bezpieczeństwa występujące w warstwie systemu plików są jeszcze prostsze niż te związane z dowiązaniami symbolicznymi. Kolejny błąd

[1] Stan na dzień 27.03.2013.

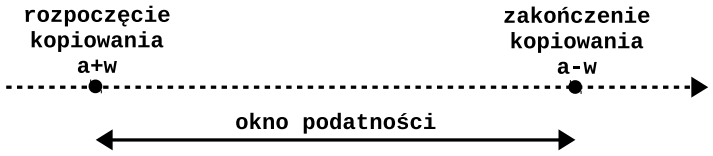
(zilustrowany na rysunku 2.9) występował w oprogramowaniu administracyjnym *logrotate* i mógł doprowadzać do wycieku poufnych informacji.



Rysunek 2.9: Błąd bezpieczeństwa w *logrotate* powodujący wyciek poufnych informacji.

Wadliwy kod tworzący plik wyjściowy w pewnych warunkach mógł pozostawiać globalne prawa odczytu. Jeśli atakującemu udawało się otworzyć ten plik, nim *logrotate* zmniejszyło uprawnienia, to możliwe było odczytanie jego zawartości. Warto zwrócić uwagę na to, że intruz po udanym otwarciu mógł odczytywać zawartość dowolnie długo, nawet jeśli prawa odczytu i zapisu zmieniły się w trakcie — w systemach uniksopodobnych weryfikacja dostępu następuje wyłącznie w momencie pozyskania deskryptora. Poprawnym rozwiązaniem omawianego problemu także i w tym wypadku było zastosowanie funkcji `mkstemp()`, która pozwalała na stworzenie pliku tymczasowego z maską uprawnień 0000, dla którego następnie ustalany był właściciel i prawa dostępu, a także zmieniana była nazwa.

Core Foundation CVE-2008-0054



Rysunek 2.10: Błąd bezpieczeństwa w *Core Foundation* pozwalający na wykonanie ataku *DoS* lub podwyższenia uprawnień.

Nieco podobny błąd bezpieczeństwa do omówionej podatności w *logrotate* wystąpił w *API Core Foundation* systemu *Mac OS X* (zilustrowany na rysunku 2.10). Przy rekurencyjnym kopiowaniu plików, docelowe katalogi tworzone były jako globalnie modyfikowalne, a dopiero po zakończeniu operacji ustawiane były poprawne prawa dostępu. W czasie pomiędzy rozpoczęciem a zakończeniem kopiowania, powstawało więc okno czasowe, w którym inni użytkownicy mogli manipulować zawartością katalogów, co umożliwiało ataki typu *DoS* lub podwyższenia uprawnień.

2.2.3. Błędy w warstwie fizycznej interakcji

Ostatnim omawianym typem błędów są podatności występujące na styku fizycznej interakcji użytkownika z systemem współbieżnym. Zgodnie z badaniem [124] stanowią one jedynie 7% wszystkich błędów bezpieczeństwa w programowaniu współbieżnym.

Lockscreen bypass

Najbardziej znanym błędem w kategorii podatności w warstwie fizycznej interakcji są wszelkie sposoby pominięcia startowego ekranu blokowania (ang. *lock screen*) w urządzeniach mobilnych. Pierwszy taki błąd [124] został odkryty w systemie operacyjnym *iOS* i pozwalał pominąć ekran wpisywania kodu odblokowującego urządzenie poprzez wykonanie w określonym tempie odpowiedniej sekwencji operacji, która składa się m.in. z wybrania połączeń alarmowych, wpisania błędnego numeru i szybkiego naciskania przycisku blokowania ekranu. Naprawienie tego błędu okazało się dosyć trudne, ponieważ pojawiał się on kilkukrotnie w różnych wersjach systemu *iOS*. Podobne błędy odnaleziono także w innych platformach mobilnych, w tym w telefonach firmy *Samsung* z systemem *Android* [84].

Request race

Błędy bezpieczeństwa występują również w aplikacjach webowych, w których współbieżność występuje pod postacią jednoczesnych zapytań kierowanych do serwera. Bardzo często zdarza się, że zapytania te operują na stanie współdzielonym, co może w prosty sposób doprowadzać do wyścigów.

Interesujący błąd (zilustrowany na rysunku 2.11), wynikający z niewłaściwej obsługi wielu zapytań, występował w przeglądarkowej grze *Forumwarz* [120] i pozwalał graczom w sposób niezgodny z zasadami gry zwielokrotniać zdobywane punkty.

```
Player.transaction do
  # Find the player's current goal
  goal = player.goal

  # Make sure we don't reward goals that have been already been completed
  unless goal.completed?
    goal.update_column :completed, true
    player.increment!(:score, goal.score)
  end
end
```

Rysunek 2.11: Błąd w przeglądarkowej grze *Forumwarz* pozwalający graczom na zwielokrotnianie swoich punktów w sposób niezgodny z zasadami gry.

W przypadku, gdy gracz wysłał w bardzo krótkim czasie dużą liczbę zapytań związanych z podjęciem nagrody, mogło dochodzić do sytuacji, gdy była ona wielokrotnie dopisywana (`player.increment!(:score, goal.score)`) do sumy punktów, ponieważ warunek `goal.completed?` pozostawał niespełniony. Programista wadliwego kodu wspomniał również, że z oczywistych powodów błąd nie występował przy testach na jednowątkowej, lokalnej maszynie, co było znacznym utrudnieniem w odnalezieniu błędu. Dodatkowo warto przytoczyć słowa autora tekstu [120] opisującego błąd w jego kodzie:

> After I discovered the cause of the exploit, I was a little worried. I'll be the first to admit I am green behind the ears when it comes to multithreading and concurrency. I was worried that I'd have to resort to semaphores or some kind of other exotic programming construct.

Pokazuje to, że pomimo bogatego doświadczenia w tworzeniu aplikacji webowych, programista nie posiadał podstawowego wykształcenia informatycznego, ani dostatecznej wiedzy na temat współbieżności. Co więcej, autor wykazuje pewną niechęć do stosowania jakichkolwiek metod synchronizacji, nazywając prymityw, jakim jest semafor, „egzotyczną konstrukcją programistyczną". Doskonale wspiera to przedstawioną w rozdziale 1 tezę, że wiele błędów w programach współbieżnych występuje z powodu niedostatecznej wiedzy i edukacji programistów.

Ostatecznie błąd został poprawiony bez zastosowania jawnej synchronizacji w kodzie — wykorzystano wsparcie blokowania ze strony bazy danych. Rozwiązanie to zostało przedstawione jako skuteczne i polecane przez autora [120], można jednak powiedzieć, że zmniejsza ono czytelność kodu.

Na problem błędów bezpieczeństwa w aplikacjach webowych zwrócono szczególną uwagę podczas konferencji *Black Hat USA 2008* [106]. Słusznie zauważono, że najpopularniejsze zbiory bibliotek (w tym *Ruby on Rails* używane w rozważanym wcześniej przykładzie) służące do tworzenia aplikacji internetowych zwykle pozostawiają zarządzanie współbieżnością programiście. Istotnym spostrzeżeniem jest również to, że testowanie różnych stanów w aplikacjach webowych jest bardzo trudne, gdyż nie ma na to uniwersalnego sposobu, a najpopularniejsze testy obciążeniowe oferują tylko metryki wydajnościowe.

2.3. Od błędu w aplikacji współbieżnej do podatności

Zaprezentowane w rozdziale 1 usterki, w przeciwieństwie do tych z niniejszego rozdziału, to w większości błędy, które nie zagrażają bezpośrednio bezpieczeństwu systemu. Warto zatem zastanowić się, co powoduje, że z pozoru zwyczajny błąd we współbieżności prowadzi do powstania istotnej podatności w oprogramowaniu. Analiza taka pozwala na określenie, którym błędom należy przyglądać się dokładniej i nadawać im wyższy priorytet, a które mogą zostać oznaczone jako pomniejsze, do naprawy w późniejszym czasie.

Dodatkowym zagadnieniem, które warto rozważyć przy okazji wspomnianej analizy, są czynniki, które powodują, że konkretna podatność zostaje uznana za poważną. Jak pokazano przy okazji badania błędów bezpieczeństwa w różnych aplikacjach, nie wszystkie z nich były uznawane za krytyczne — wiele identyfikuje się jedynie jako pomniejsze zagrożenie.

2.3.1. Błąd funkcjonalny a błąd bezpieczeństwa

W przypadku programów sekwencyjnych rozróżnienie błędu funkcjonalnego od błędu bezpieczeństwa jest zwykle jasne. Podatnością określa się zwyczajowo każdy błąd w oprogramowaniu, który umożliwia pewnej osobie (atakującemu) wyrządzić istotną dla rozważanego systemu szkodę [5]. W kwestii błędów w programach współbieżnych trudno jednak o jednogłośną opinię. W literaturze spotyka się intuicyjnie naturalne określenie, że błąd w programie współbieżnym, który narusza politykę bezpieczeństwa, jest podatnością [121]. Z drugiej strony popularne jest też

stwierdzenie, że każde naruszenie specyfikacji lub oczekiwań użytkownika względem systemu należy zaklasyfikować jako błąd bezpieczeństwa [122].

W zasadzie każde opracowanie dotyczące błędów bezpieczeństwa w programach współbieżnych podkreśla z kolei, że wnioskowanie na temat współbieżności jak i bezpieczeństwa jest w zasadzie tym samym procesem i nie należy rozważać tych pojęć rozdzielnie [121, 122, 124]. Niespodziewana współbieżność, niewłaściwa synchronizacja wynikająca ze złego zaplanowania dekompozycji, błędy implementacyjne lub nieznajomość środowiska uruchomieniowego mogą skutkować podatnością w systemie. Podsumowując, nie istnieje uniwersalny sposób na ocenienie, czy dany błąd w oprogramowaniu współbieżnym jest jednocześnie błędem bezpieczeństwa — każdorazowo wymaga to dogłębnej analizy eksperckiej. Z tego też powodu nie należy ignorować praktycznie żadnych błędów w programach współbieżnych.

Niemniej jednak bardzo często wskazuje się dwa elementy współbieżnych systemów informatycznych, które są szczególnie często źródłem błędów bezpieczeństwa: interfejs programistyczny oraz składnice danych [122, 124]. W pierwszym przypadku błędy pojawiają się z powodu wyścigów występujących w interpretacji argumentów (jak w omawianej podatności w *systrace*) lub z powodu nieatomowego użycia sprawdzenia i dostępu do pewnych zasobów (jak w omawianej klasie błędów typu *symlink race*). W przypadku składnic danych podatności najczęściej wynikają z przestarzałych lub niespójnych metadanych dotyczących bezpieczeństwa lub z powodu niespójności pomiędzy danymi a metadanymi (jak w przypadku omówionego błędu w *Moonlight*).

2.3.2. Trudności w wykorzystywaniu podatności

Niezależnie od tego, czy błąd współbieżny umożliwia wykonanie uderzenia np. typy *DoS* czy zdalnego wykonania kodu, to na jego istotność wpływa tzw. okno podatności, które jest głównym czynnikiem stanowiącym o trudności powodzenia ataku. Okno podatności, to zgodnie z intuicją, odcinek czasu, w którym możliwe jest pomyślne wykorzystanie podatności. Naturalnie zatem, im okno podatności jest szersze, tym wykonanie skutecznego ataku prostsze, a istotność samego błędu większa.

Szerokość okna podatności zależy przede wszystkim od warstwy, w której błąd występuje. Typowo, w przypadku błędów w warstwie pamięci okno podatności mierzone jest w nanosekundach, w przypadku błędów w warstwie systemu plików w milisekundach, a w przypadku błędów w warstwie fizycznej interakcji w sekundach.

Kolejnym istotnym zagadnieniem związanym z oknem podatności jest możliwość powtarzania prób wykorzystania błędu. Jeśli okno podatności jest bardzo wąskie, ale intruz ma możliwość wykonania wielu prób w krótkim czasie, to szanse na skuteczny atak znacznie wzrastają. Z drugiej strony często zdarza się, że atakujący ma tylko jedną szansę w ataku na szersze okno np. milisekundowe (jest tak przykładowo w przypadku uruchamiania systemu i podatności typu *symlink race*) i jeśli mu się nie uda, musi stosunkowo długo czekać na kolejną możliwość. Przy okazji omawiania wielokrotnych, powtórnych ataków warto wspomnieć, że tego typu działania mogą zostać w łatwy sposób wykryte przez detektory zachowań anomalnych.

Z tego powodu atakujący znacznie częściej wykorzystują wyrafinowane techniki, które za pomocą odpowiednio spreparowanych akcji pozwalają na rozszerzenie okna podatności. Metody te polegają zwykle na dobraniu odpowiednich danych

wejściowych, które wymagają wydłużonego czasu przetwarzania. W przypadku omówionych wcześniej błędów rozszerzanie okna podatności może wyglądać w następujący sposób:

— **uselib()** — w przypadku gdy alokowana przez funkcję `kmem_cache_alloc()` pamięć jest dostępna natychmiastowo, to okno podatności jest wąskie; wprowadzenie systemu w stan niskiej pamięci może natomiast zmusić do wymiatania, rozszerzając okno podatności z czasu pamięciowego do czasu dyskowego,
— **systrace bypass** — w przypadku próby wykorzystania błędu ominięcia *systrace* na maszynie jednoprocesorowej atakujący musi spowodować wywłaszczenie procesu (aby dodatkowy wątek mógł podmienić argumenty), co może udać się poprzez wymuszenie błędu strony (ang. *page fault*); technika ta służy także rozszerzaniu okna w przypadku maszyny wieloprocesorowej,
— **Moonlight** — rozszerzenie okna podatności w błędzie w *Moonlight* sprowadza się do żądania skopiowania dostatecznie dużej tablicy tak, aby zwiększyć liczbę iteracji pętli kopiującej,
— **symlink race** — naturalnym sposobem na rozszerzenie okna podatności w przypadku błędów związanych z dowiązaniami symbolicznymi jest używanie długich ścieżek plików, które nie są aktualnie przechowywane w pamięci *cache resolvera*,
— **request race** — wykorzystanie błędów związanych z współbieżnymi zapytaniami do serwera może być ułatwione, gdy atak przeprowadzany jest w sytuacji dużego obciążenia spowodowanego działalnością innych użytkowników.

Widać zatem, że nawet w przypadku, gdy wykorzystanie podatności wydaje się praktycznie niemożliwe ze względu na uwarunkowania czasowe, atakujący bardzo często znajdują sposoby, które powodują, że szanse na skuteczny atak stają się realne.

2.4. Wpływ współbieżności na techniki wykrywania podatności i obrony przed atakami

Naturalnym skutkiem rozwoju cyberprzestępczości jest powstawanie nowych i ulepszanie istniejących technik wykrywania podatności i obrony przed atakami. Zdecydowana większość używanych współcześnie mechanizmów została jednak zaprojektowana dla programów sekwencyjnych. Zasadne jest zatem rozważenie, jak współbieżność wpływa na skuteczność wspomnianych metod i jak ewentualnie mogą zostać one dostosowane do obecnych wymagań. Niniejszy podrozdział jest jedynie szkicem problemu, a temat narzędzi wspierających programowanie współbieżne został szerzej omówiony w rozdziale 4.

2.4.1. Techniki wykrywania podatności

Techniki wykrywania podatności mają za zadanie rozpoznać ewentualne problemy w kodzie nim zostanie on wdrożony do środowiska produkcyjnego. Istniejące narzędzia tego typu praktycznie ograniczają się do statycznej i dynamicznej analizy kodu ze wspomaganiem debuggera. Można zdecydowanie stwierdzić, że niestety nie osiągnęły one jeszcze stopnia rozwoju narzędzi sekwencyjnych [122, 124] — dynamiczne analizatory oferują badanie wyłącznie pojedynczego, losowego przeplotu (ang. *interleaving*) ze względu na eksplozję kombinatoryczną możliwych stanów, natomiast statyczne analizatory bardzo często zgłaszają wiele tzw. *false–positives*,

zaciemniając faktyczne błędy. Debuggery w zasadzie nie istnieją w wersjach dostosowanych do programowania współbieżnego, a istniejące rozwiązania sekwencyjne nierzadko bardziej szkodzą, niż pomagają [122].

Istnieje zatem zapotrzebowanie na nowe, skuteczniejsze narzędzia wykrywania podatności. Zgodnie ze wcześniejszymi spostrzeżeniami, metody detekcji powinny być nastawione na wyszukiwanie błędów w interfejsie programistycznym — na styku warstw systemu. Kolejnym pomysłem jest zorientowanie narzędzi na elementy przechowujące wrażliwe dane — identyfikatory użytkownika, wskaźniki na funkcje, metadane typu czy mapy pamięci. Ostatnią propozycją jest wysuwanie na pierwszy plan wykrytych przez analizatory statyczne potencjalnych błędów o najszerszym oknie podatności ocenianym w sposób heurystyczny — pozwoliłoby to na łatwiejsze odnalezienie istotnych fragmentów programu pośród wspomnianych *false–positives*.

2.4.2. Techniki obrony przed atakami

Techniki obrony przed atakami zakładają, że w programie istnieją pewne nieznane programistom podatności i po prostu należy się bronić przed wtargnięciem intruza. Większość istniejących metod obrony została zaprojektowana z myślą o programach sekwencyjnych zatem istotne jest, aby przeanalizować, czy są one skuteczne również dla programów współbieżnych. Mechanizmy obrony typowo dzieli się na [124]:
— śledzenie metadanych,
— kontrolę programową,
— kontrolę sprzętową,
— wykrywanie anomalii,
— randomizację.
Niniejszy podział wykorzystano w poniższych rozważaniach.

Śledzenie metadanych

Śledzenie metadanych obejmuje techniki pozwalające wnioskować o poprawnym stanie wykonywanego programu na podstawie dodatkowych danych, takich jak zakresy w tablicach lub śledzenie skażenia (ang. *taint–tracking*). Jeśli w programie współbieżnym wykorzystującym tę metodę występują wyścigi na danych, to istnieje duża szansa, że wykorzystywane metadane bezpieczeństwa są niespójne. Z tego powodu technika ta uznawana jest za nieskuteczną w programach współbieżnych i nie istnieją proste metody poprawienia jej skuteczności [122, 124].

Kontrola programowa

Kontrola programowa to technika nieco podobna do śledzenia metadanych. Opiera się na wykonywaniu dodatkowych sprawdzeń pewnych warunków, które mają zwiększać bezpieczeństwo programu jak np. kontrola wartości wskaźników czy sprawdzanie typów danych. Jeśli metody te nie są odpowiednio przygotowane dla programowania współbieżnego, to istnieje duże prawdopodobieństwo, że są one podane na ataki *TOCTOU* opisywane wcześniej. Techniki kontroli programowej muszą być zatem odpowiednio dostosowane przed skutecznym użyciem.

Kontrola sprzętowa

Kontrola sprzętowa to odpowiednik kontroli programowej zaimplementowany w sprzęcie. Najczęściej zestaw możliwych operacji jest bardzo ubogi (np. ochrona przed wykonaniem kodu poprzez oznaczanie segmentów pamięci jako niewykonywalne (ang. *non–executable*)), jednak cechuje się on skutecznością i wydajnością. Uważa się, że wszelkie metody sprzętowe mają równie duże zastosowanie i skuteczność w programach współbieżnych jak w sekwencyjnych [124].

Wykrywanie anomalii

Mechanizmy wykrywania anomalii działają zwykle na zasadzie maszynowego uczenia się typowego zachowania systemu i ujawniania wszelkich zachowań niezgodnych z wyuczoną polityką. Problematyczną kwestią w uczeniu się działania programów współbieżnych jest ich bardzo duża złożoność wynikająca wprost z wielowątkowości i eksplozji kombinatorycznej. Z tego powodu stworzone przez detektory anomalii zasady mogą być zbyt skomplikowane, a tym samym obfite w szumy lub z drugiej strony zbyt uproszczone, by działać skutecznie. Brakuje zatem odpowiednich modeli, które pozwoliłyby w wystarczająco dokładny, a zarazem czytelny sposób przedstawiać i interpretować działanie systemów współbieżnych.

Randomizacja

Techniki randomizacji polegają na dodaniu pewnej losowości do środowiska uruchomieniowego programu. Randomizacja najczęściej przybiera postać losowego układu pamięci (ang. *address space layout randomization, ASLR*), który utrudnia atakującemu interpretację mapy pamięci procesu, zmniejszając tym samym szansę na przeprowadzanie skutecznych ataków takich, jak choćby wstrzykiwanie kodu. Ze względu na trudność oceny skuteczności, metody wprowadzania losowości są kontrowersyjnym [5], aczkolwiek powszechnie używanym mechanizmem obrony przed atakami. Przyjmuje się, że stosowanie tych technik w programach współbieżnych przynosi dokładnie takie same korzyści, jak w programach sekwencyjnych, gdyż żadne założenia dotyczące metod randomizacji nie są przez współbieżność osłabiane.

2.5. Podsumowanie

W niniejszym rozdziale przedstawiono i szczegółowo omówiono kilkanaście przykładowych błędów w programach współbieżnych, które bezpośrednio skutkują powstaniem podatności. Wspomniana w rozdziale skala różnych problemów tego typu pokazuje, że współbieżność jest bezpośrednio związana z bezpieczeństwem. Błędy bezpieczeństwa w programowaniu współbieżnym to zatem realny problem, który jest warty głębszych rozważań.

Nie istnieje za to łatwa odpowiedź na pytanie, co powoduje, że zwykły błąd w programie współbieżnym staje się podatnością. Nie ma uniwersalnych metod, które pozwalają rozróżnić błędy funkcjonalne od błędów bezpieczeństwa — zwykle jest to w stanie ocenić jedynie ekspert. Wnioskowanie na temat współbieżności jest jednak zawsze silnie powiązane z wnioskowaniem na temat bezpieczeństwa.

Pomimo trudności z określeniem genezy błędów bezpieczeństwa zauważono, że większość podatności występuje w warstwie interfejsu programistycznego oraz w elementach systemu przechowujących dane i z tego powodu warto szczególnie

przyglądać się właśnie tym warstwom oprogramowania. Ponadto, jak pokazała analiza przykładowych błędów, zasadnicze znaczenie w trudności wykorzystania podatności ma okno czasowe, w którym owa podatność występuje. Należy także zwracać szczególną uwagę na nieoczywiste sposoby rozszerzania okna podatności, które mogą być wykorzystywane w celu ułatwienia ataków.

Trudnej sytuacji bezpieczeństwa programów współbieżnych nie polepszają istniejące techniki wykrywania podatności i ochrony przed atakami. Funkcjonujące mechanizmy detekcji błędów bezpieczeństwa w zasadzie ograniczają się do niedojrzałych metod statycznej i dynamicznej analizy kodu. W przypadku dynamicznej analizy kodu testowanie jest zwykle zbyt ubogie, aby było użyteczne, a w przypadku statycznej analizy otrzymywane wyniki zawierają zbyt wiele fałszywych alarmów, aby pokazywać rzeczywiste błędy. Spośród technik obrony przed atakami trzy z pięciu omówionych (śledzenie metadanych, kontrola programowa, wykrywanie anomalii) są istotnie osłabione przez fakt występowania współbieżności w systemie. Jedynie kontrola sprzętowa i randomizacja zachowują identyczną skuteczność zarówno w programach współbieżnych jak i sekwencyjnych. Techniki wykrywania podatności, jak również obrony przed atakami, muszą zostać znacznie zmodernizowane lub wręcz stworzone na nowo, nim staną się istotną pomocą w tworzeniu bezpiecznych programów współbieżnych.

W początkowych rozdziałach niniejszej pracy, poprzez pokazanie i omówienie przykładowych błędów, zarysowano problem, jakim jest tworzenie niezawodnych i bezpiecznych programów współbieżnych. Warto zatem zastanowić się, jakie kroki należy podjąć, aby kod współbieżny, który z całą pewnością będzie dominował w następnych latach w inżynierii oprogramowania, istotnie zwiększył swoją jakość skutkując poprawnymi programami. Kwestię różnych sposobów osiągnięcia tego celu rozważono w kolejnym rozdziale.

3. Nowoczesne techniki programowania współbieżnego

W rozdziałach 1 i 2 opisano najpoważniejsze problemy jakie związane są z tworzeniem niezawodnych i bezpiecznych programów współbieżnych. Ogrom różnorodnych przeszkód, które napotykają programiści rozwiązań współbieżnych może być przytłaczający i zniechęcający. Obserwując rozwój jednostek wieloprocesorowych może się niestety wydawać, że od programowania współbieżnego nie ma ucieczki. Co więcej, bardzo możliwe, że świat obliczeń komputerowych stoi przed kolejną rewolucją — tym razem wielkoskalowej równoległości (ang. *massively parallel*). Firmy takie jak *Adapteva* [3] już w najbliższym czasie zamierzają zaoferować niedrogie komputery o znacznej liczbie procesorów. W najbliższym czasie można np. spodziewać się architektury hybrydowej *Parallella* złożonej z 64 procesorów o łącznej mocy ponad 45 GHz wraz z dwurdzeniowym procesorem sterującym *ARM A9*, kosztującej kilkaset dolarów [56]. Należy zatem jak najszybciej opracować i wdrożyć postępowe techniki programowania współbieżnego tak, aby powstające i rozwijane systemy wykorzystywały w pełni dostępny sprzęt, zmniejszając tym samym deficyt mocy obliczeniowej.

Zagadnienie nowoczesnych metod tworzenia programów współbieżnych podjęto w niniejszym rozdziale. W pierwszej jego części opisano proponowane przez autorytety w tematyce współbieżności najlepsze praktyki, które pozwalają uniknąć podstawowych błędów i problemów. W drugim podrozdziale wprowadzono tematykę narzędzi wspierających tworzenie programów współbieżnych, jednak dokładnie temat ten rozwinięto w rozdziale 4. Kolejna sekcja opisuje innowacyjne techniki i mechanizmy programowania współbieżnego funkcjonujące w najczęściej wykorzystywanych platformach programistycznych. W przedostatnim podrozdziale scharakteryzowano trendy w rozwoju nowoczesnych języków programowania, które jawnie wspierają współbieżność i równoległość. Rozdział kończy się krótkim podsumowaniem dokonanego przeglądu. Organizacja materiału w rozdziale odzwierciedla narastającą trudność wdrożenia kolejnych sposobów tworzenia oprogramowania współbieżnego — od ogólnych zaleceń poprzez narzędzia i techniki aż do wyrafinowanych paradygmatów wymagających od programisty zupełnie odmiennego podejścia do tworzenia oprogramowania.

3.1. Najlepsze praktyki

Inżyniera oprogramowania jest młodą dziedziną, szczególnie jeśli zestawia się ją z innymi dyscyplinami inżynierskimi takimi jak choćby budownictwo. Z tego też powodu dla wytwarzania oprogramowania nie opracowano jeszcze kompletnego i niezawodnego zbioru zasad, według których należy postępować na drodze do udanego projektu. Przyjmuje się zatem, że wszelkie zalecenia formułowane przez

autorytety w dziedzinie inżynierii oprogramowania mają charakter zdroworozsądkowych reguł i wzorców opracowanych na podstawie doświadczenia w tworzeniu konkretnych systemów, a co za tym idzie — nie dają one gwarancji pomyślnego zastosowania w każdym projekcie. Nierzadko zdarza się też, że porady wygłaszane przez różne osoby są ze sobą sprzeczne, co pokazuje ich nienaukowy charakter. Wszelkie najlepsze praktyki należy zatem rozważać w konkretnych okolicznościach i środowisku, a także z należytą krytyką.

Pomimo że programowanie współbieżne dopiero niedawno stało się jednym z najistotniejszych zagadnień współczesnej inżynierii oprogramowania, to już pojawiło się stosunkowo dużo zaleceń na temat właściwego tworzenia aplikacji tego typu. Najważniejsze z zasad dotyczące projektowania, implementacji i testowania programów współbieżnych zaprezentowano poniżej. Do przeglądu wybrano praktyki możliwe do zastosowania w dowolnej technologii i na dowolnej platformie programistycznej.

3.1.1. Przegląd praktyk

Opisanie wszystkich porad, jakie zostały wygłoszone w kwestii programowania współbieżnego, jest zadaniem niemożliwym. Z tego powodu dokonano podstawowego przeglądu, mającego na celu pokazanie typowych wskazówek udzielanych programistom. Ponadto warto zaznaczyć, że zaprezentowane praktyki abstrahują od konkretnych środowisk programowych i sprzętowych — dla konkretnych platform i środowisk uruchomieniowych formułuje się zwykle oddzielne, bardziej szczegółowe zalecenia.

Unikanie współbieżności

Wiele opracowań [39, 111, 122] dotyczących programowania współbieżności zaznacza, że jeśli to możliwe i wystarczające (pod względem wydajności i skalowalności), to należy unikać współbieżności. Brak współbieżnych, powiązanych operacji oznacza oczywiście brak problemów, które zostały opisane wcześniej. Niestety, jak wielokrotnie wspominano, programów całkowicie sekwencyjnych będzie coraz mniej, a więc całkowite ucieknięcie od współbieżności wydaje się bardzo trudne.

Preferowanie determinizmu i silnych modeli spójności

W pełni deterministyczne zachowania systemu są bardziej pożądane od działań (nawet poprawnych) uwarunkowanych czasowo w sposób losowy [111, 122]. Pomimo że ta porada wydaje się być całkiem oczywista, nie zawsze jest stosowana w praktyce. Przewidywalne systemy współbieżne są łatwiejsze w zrozumieniu, a tym samym w weryfikacji i ewentualnym testowym uruchamianiu z podłączonym debuggerem. Wytworzenie takiego rozwiązania jest często trudniejsze i zwykle jest ono wolniejsze, a więc i droższe od oprogramowania, w którym występuje pewna losowość — zawsze należy jednak brać pod uwagę koszty testowania i późniejszego utrzymania.

Bezpośrednio z preferencją determinizmu wiąże się faworyzowanie silnych modeli spójności. Modele te także są łatwiejsze w zrozumieniu i wykorzystaniu, jednak w systemach silnie rozproszonych mogą być trudne lub wręcz niemożliwe do zrealizowania [111, 122]. Zastosowanie silnego modelu spójności danych powoduje, że programiści mogą podchodzić do systemu rozproszonego podobnie jak do standardowego programu z deterministyczną pamięcią współdzieloną.

Kooperacja jest lepsza niż konkurencja

Zalecaną metodą dostępu do zasobów współdzielonych (np. struktur danych) jest kooperacja [77]. Kooperacja zakłada, że możliwe jest jednoczesne wykorzystywanie zasobu przez wiele zadań, przy czym nie jest powiedziane, w jaki sposób przebiega zarządzanie dostępem. Konkurencja natomiast wymaga jawnego żądania i oczekiwania na udostępnienie zasobu, zwykle na wyłączność. Kooperatywne zasoby są łatwiejsze w wykorzystaniu (nie da się używając ich zapomnieć o odpowiednim zajęciu), mogą być szybsze (często nie blokują innych zadań w oczekiwaniu na udostępnienie), a ponadto pozwalają unikać zakleszczeń.

Oczywiście nie wszystkie zasoby mogą być udostępniane w sposób swobodny, pozbawiony synchronizacji. Budowa kooperatywnego zasobu nie wymaga jednak całkowitego pozbycia się synchronizacji, a jedynie zrealizowania dostępnych operacji w taki sposób, aby nie wymagały one jawnej kontroli dostępu (np. poprzez zastosowanie planisty lub wzorca monitora). Jest to trudniejsze w implementacji, ale ponowne wykorzystywanie raz napisanego w ten sposób kodu jest zwykle łatwe. Więcej o kooperatywnych strukturach danych znajduje się w podrozdziale 3.3.4.

Unikanie globalnego stanu współdzielonego

Zmienne globalne są bardzo często traktowane jako antywzorzec projektowania oprogramowania [32]. Powodują one naruszenie izolacji i luźnego powiązania poszczególnych modułów. Niemniej jednak bardzo często jest to najprostszy środek realizacji założeń projektowych.

W przypadku programów współbieżnych współdzielony stan jest jeszcze bardziej kłopotliwy w utrzymaniu, ponieważ wymaga konsekwentnej strategii współdzielenia dostępu. Może to potencjalnie powodować wiele problemów, które mogą wynikać z pominięcia obowiązkowego blokowania dostępu lub zakleszczeń w oczekiwaniu na dostęp do wielu zasobów globalnych. Co więcej, w programowaniu współbieżnym współdzielony stan rozumie się znacznie szerzej niż jedynie jako zmienne globalne — stanem współdzielonym są praktycznie wszystkie dane znajdujące się na stercie, ponieważ potencjalnie może mieć do nich dostęp wiele wątków na raz (np. dostęp do pól klasy poprzez wiele współbieżnych metod jednego obiektu). Z tego powodu należy dbać o jak najsilniejsze metody izolacji dostępu do danych na stercie i ograniczanie decyzji, które prowadzą do powstawania zmiennych współdzielonych.

Całkowite wyeliminowanie stanu współdzielonego wydaje się bardzo trudne, jak pokazują jednak badania nad językami funkcyjnymi (zob. podrozdział 3.4.2), w wielu zastosowaniach jest to realne i pozwala osiągać wymierne korzyści.

Separacja obliczeń i skutków ubocznych

Bezpośrednio z unikaniem stanu współdzielonego powiązana jest separacja obliczeń i skutków ubocznych. Skutki uboczne to wszelkie działania, które powodują zmiany poza wykonującym się programem, np. wywołania systemowe czy operacje na pamięci współdzielonej z innych procesem. Potocznie skutki uboczne nazywa się interakcją ze światem zewnętrznym procesu. Obliczenia to natomiast wszystkie operacje na strukturach danych znajdujących się w lokalnej pamięci procesu. Istotną przewagą obliczeń nad skutkami ubocznymi jest to, że są idempotentne — mogą być wiele razy powtarzane bez konsekwencji zewnętrznych, a więc mogą być

realizowane w transakcjach w schemacie *commit–rollback*. Obliczenia doskonale więc nadają się do zrównoleglania, nawet gdy wymagana jest synchronizacja.

Dużo trudniejszą kwestią jest zrównoleglanie skutków ubocznych, ponieważ zwykle nie są one możliwe do wycofania (np. wysłanie pakietu sieciowego), a więc wymagają silnej synchronizacji. Tworząc programy w sposób, który rozdziela obliczenia od skutków ubocznych wyznacza się w naturalny sposób linie podziału tego, co powinno być wykonywane współbieżnie od części czysto sekwencyjnych.

Silną koncepcję rozdzielenia obliczeń i skutków ubocznych prezentują znów języki funkcyjne (zob. podrozdział 3.4.2). Cecha ta w połączeniu z unikaniem globalnego stanu współdzielonego powoduje, że są one bardzo dobrym narzędziem do programowania współbieżnego.

Wykorzystywanie dobrze określonych protokołów

Stosowanie jawnego blokowania za pomocą zamków i innych metod synchronizacji oferuje wysoką wydajność i pełną kontrolę nad wykonaniem programu. Odbywa się to jednak kosztem czytelności architektury systemu, co może w prosty sposób prowadzić do powstawania błędów. Z tego powodu w wielu zastosowaniach, w których najwyższa wydajność nie jest najważniejsza, należy skłaniać się ku metodom skierowanym na protokół (ang. *protocol–centric*) [111, 122] takim jak np. przekazywanie wiadomości (ang. *message–passing*) (zob. podrozdział 3.3.2). Metody te, pomimo tego, że są istotnie wolniejsze [112], ze względu na swoją klarowność znacznie ułatwiają próbne uruchamianie (ang. *debugging*) i wnioskowanie na temat działania.

Elegancka architektura nie zawsze sprzyja bezpieczeństwu

W innych opracowaniach podkreśla się, że stosowanie architektur promujących luźne powiązanie komponentów może prowadzić do powstawania błędów bezpieczeństwa [121, 122]. Przykład takiego błędu, wynikającego właśnie z obranej organizacji programu, stanowi podatność typu *TOCTOU* w mechanizmie *systrace* opisana w punkcie 2.2.1. Nie istnieje zatem uniwersalne (niezależne od systemu) podejście zapewniające wszystkie pożądane cechy.

Dokumentowanie strategii synchronizacji

Niezwykle istotnym elementem właściwie wytworzonego systemu współbieżnego jest dokumentacja strategii synchronizacji [122]. Dokument taki służy nie tylko ułatwieniu komunikacji między programistami, ale dodatkowo pozwala uwypuklić pewne problemy, a także ułatwia weryfikację i testowanie powstającego kodu. Zalecenie to może wydawać się oczywiste, bo zgodnie ze sztuką inżynierii oprogramowania każdy system powinien być odpowiednio udokumentowany. W praktyce wiadomo jednak, że nie zawsze tak się dzieje. W wielu przypadkach uważa się, że dostęp do kodu programu jest w zupełności wystarczającym źródłem wiedzy o budowie i działaniu, jednak w przypadku synchronizacji zasada ta praktycznie nigdy nie ma zastosowania.

Doskonałym przykładem tego zjawiska jest kod jądra *Linuksa*, którego dokumentacja jest zwykle bardzo uboga i często sprowadza się jedynie do komentarzy w kodzie źródłowym. Inaczej jest w przypadku kodu, który wymaga synchronizacji — praktycznie zawsze jest bogato skomentowany, a nierzadko opisany w dodatkowym dokumencie. Ponadto tworzy się całe oddzielne opracowania, artykuły,

a nawet rozdziały w książkach [66, 74, 102] dedykowane współbieżnym operacjom w jądrze.

Niezależnie od wybranych metod współdziałania poszczególnych zadań, synchronizacja jest trudna do zrozumienia i zawsze wymaga nieco więcej uwagi niż pozostałe zagadnienia związane z wytwarzaniem systemu.

Unikanie zależnościach czasowych

Jeśli wystąpienie pewnego błędu uwarunkowane jest zajściem określonych, mało prawdopodobnych zależności czasowych, to z dużą pewnością można zakładać, że błąd ten kiedyś w końcu wystąpi. Nie można dopuszczać do tego, aby rzeczy istotne, takie jak niezawodność systemu, zależały od szczęścia i wątłych założeń (zgodnie z zasadą *things which matter most should never be at mercy of things which matter least* (ang.) [87]).

Przykłady bardzo wąskich okien czasowych błędu pokazane zostały zarówno w rozdziale 1 jak i w rozdziale 2 i za każdym razem okazywało się, że pewne okoliczności powodowały ujawnienie się usterki. Co więcej, bardzo często okazuje się, że na jednej platformie uruchomieniowej wystąpienie danego błędu jest faktycznie niemożliwe, a na innej błąd ten jest bardzo powszechny. Wszystkie zależności czasowe, które mogą prowadzić do naruszenia specyfikacji należy zatem traktować jako niepożądane, niezależnie od tego jak nieprawdopodobne jest ich wystąpienie.

Algorytmy i wzorce projektowe

Dla programowania współbieżnego, podobnie jak dla programowania obiektowego i strukturalnego, stworzono algorytmy [11] i wzorce projektowe [32], które reprezentują ustalony i sprawdzony sposób rozwiązywania pewnych problemów. W przypadku współbieżności zasób dostępnych algorytmów i wzorców jest jednak bardziej ubogi i nie zawsze istnieje powszechna zgoda co do ich słuszności. Niemniej jednak w literaturze bardzo często poleca się wzorce takie jak: monitor, aktywny obiekt, *fork–join* czy lokalna składnica danych zadania [32, 39]. Wśród istotnych algorytmów warto wspomnieć z kolei o algorytmie czytelników i pisarzy czy problemie producenta–konsumenta [11, 112]. Wykorzystanie tych zalecanych rozwiązań nie gwarantuje sukcesu, jednak w specyficznych sytuacjach daje szansę na uzyskanie właściwego zarządzania współbieżnością. Należy zatem być świadomym istnienia wzorców i algorytmów dla programowania współbieżnego i korzystać z nich rozważnie.

Znajomość platformy uruchomieniowej

Programowanie współbieżne wymaga dogłębnej znajomości środowiska, w którym program będzie uruchamiany. Osoby tworzące system nie mogą polegać na zjawiskach niegwarantowanych przez platformę, takich jak choćby domniemana atomowość pewnych sekcji czy prawdopodobny schemat działania planisty [122]. Warto zatem korzystać z warstw pośrednich (takich jak np. biblioteka standardowa), które zapewniają spójny widok na dostępne operacje i umożliwiają tworzenie kodu przenośnego.

Wiele środowisk testowych

Testowanie kodu współbieżnego powinno odbywać się w zróżnicowanych środowiskach, ponieważ jak już wielokrotnie zauważono, na ujawnianie się błędów mają

wpływ różne czynniki. Powszechną strategią jest używanie zarówno maszyn szybkich jak i bardzo wolnych [122], ponieważ te pierwsze ujawniają błędy występujące w wąskich oknach czasowych, natomiast te drugie zwiększają szansę na ujawnienie się błędów wynikających z założeń dotyczących długości operacji. Kolejna metoda sugeruje testowanie na maszynach jedno– jak i wieloprocesorowych z podobnych powodów [121] — niektóre błędy ujawniają się wyłącznie w środowisku prawdziwie równoległym, inne łatwiej spowodować w środowisku z pozorną równoległością. Ostatnim zaleceniem jest przeprowadzanie współbieżnych testów obciążeniowych, ponieważ przy właściwej organizacji pomagają one zweryfikować dużą liczbę możliwych przeplotów w krótkim czasie [69]. Proces testowania systemów współbieżnych nie może zatem ograniczać się do ustalonego środowiska rozwojowego, a powinien angażować możliwie najwięcej dostępnych, docelowych konfiguracji.

3.2. Narzędzia

Jak pokazano w rozdziałach 1 i 2 narzędzia wspierające programowanie współbieżne są jeszcze bardzo niedojrzałe, a przez to wyjątkowo rzadko stosowane. Z tego powodu, aby odszukać przyczynę obecnego stanu rzeczy i ewentualnie zidentyfikować obszary, w których możliwe jest polepszenie tej sytuacji, obszerną analizę koncepcji i konkretnych narzędzi przeprowadzono w kolejnym rozdziale. Uzasadnieniem specjalnego potraktowania tej dziedziny wytwarzania systemów jest wspomniane wcześniej zapotrzebowanie na nowe i skuteczne narzędzia dla współbieżności.

3.3. Techniki i mechanizmy

Kolejnym, jednak nieco bardziej wymagającym krokiem w kierunku niezawodnych programów współbieżnych jest stosowanie nowoczesnych technik i mechanizmów oferowanych przez dostępne platformy programistyczne. Rozwój tych metod jest jednak istotnie dynamiczniejszy od postępu w dziedzinie narzędzi. Komitety opiekujące się standardami platform i języków programowania coraz częściej dostrzegają potrzebę dostosowania istniejących technologii do programowania współbieżnego. Efektem tych starań są np. standardy *C++11* [110] czy *Parallel Extensions* dla platformy *.NET* [78], które wprowadzają wiele użytecznych technik do konstruowania programów współbieżnych. Do wielu platform wprowadzane są także znane od lat mechanizmy, które ze względu na niegdyś małe zastosowanie współbieżności zostały zapomniane, jak np. pamięć transakcyjna czy też komunikacja poprzez przekazywanie komunikatów. Można zatem powiedzieć, że metody programowania współbieżnego na wielu platformach rozwijają się w sposób dynamiczny.

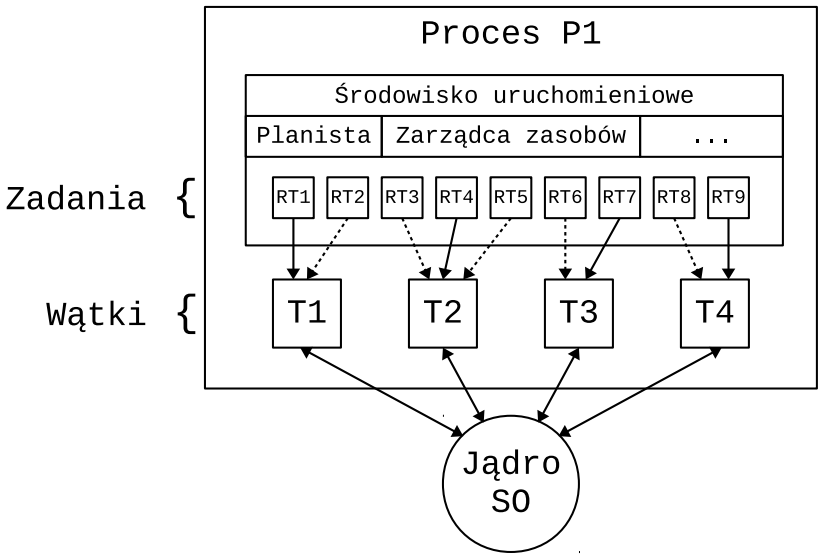
W niniejszym podrozdziale przedstawiono nowoczesne, najpopularniejsze techniki i mechanizmy wykorzystywane w funkcjonujących językach programowania i środowiskach programistycznych. W opisach abstrahuje się od konkretnych implementacji, koncentrując się na cechach rozważanego rozwiązania. Niektóre z koncepcji zostały zilustrowane przykładami z istniejących platform programistycznych.

3.3.1. Programowanie zorientowane na zadania

Najpopularniejszym i mogłoby się zdawać najważniejszym trendem w programowaniu współbieżnym jest odchodzenie od tradycyjnego modelu opartego na wątkach, które są kojarzone z niskopoziomowym prymitywem oferowanym przez system operacyjny, na rzecz lekkich zadań istniejących wyłącznie w ramach aplikacji i niewidocznych z poziomu systemu operacyjnego. Takie podejście ma cechy ideologiczne, polegające na jawnym odrzuceniu paradygmatu niskopoziomowego programowania współbieżnego, ale przede wszystkim wynika z pewnych istotnych przesłanek technicznych, które opisane zostały poniżej.

Pule wątków

Podstawową jednostką wykonawczą w programowaniu zorientowanym zadaniowo jest zadanie (nazywane często zielonym wątkiem (ang. *green thread*)[1] [109] lub wątkiem w przestrzeni użytkownika [112]), a nie tradycyjny wątek. Zadania rezydują zwykle w tzw. środowisku uruchomieniowym (ang. *runtime*), które odpowiedzialne jest m.in. za rozporządzanie zasobami takimi jak pamięć czy deskryptory plików oraz za szeregowanie zadań na dostępnych dla środowiska wątkach z puli wątków (ang. *thread pool*)) poziomu systemu operacyjnego. Rozmiar puli wątków może być ustalany dynamicznie w zależności od liczby zadań lub statycznie — przy uruchamianiu lub kompilacji.



Rysunek 3.1: Poglądowa architektura procesu z zadaniami zamiast tradycyjnych wątków.

Środowiskiem uruchomieniowym może być maszyna wirtualna (jak np. w przypadku języków dla *CLR* platformy *.NET*), interpreter (jak np. dla języka *Ruby*) lub

[1] Niekiedy można spotkać się z ograniczeniem pojęcia *wątek zielony* do modeli 1:N, czyli gdy wiele zadań wykorzystuje wyłącznie jeden procesor. Zwykle jednak z pojęcia tego korzysta się także w modelach N:M, gdy wiele zadań wykorzystuje wiele procesorów.

może być linkowane do programu w czasie kompilacji (jak np. dla języka kompilowanych do kodu maszynowego takich jak *Go* czy *Haskell*). W zależności od konstrukcji środowiska uruchomieniowego, może ono dedykować pewne wątki dla planisty, menadżera zasobów i innych elementów wewnętrznych lub mogą one współdzielić z zadaniami wątki z puli. Szeregowanie zadań bardzo często jest elementem konfigurowalnym i może polegać m.in. na statycznym łączeniu konkretnych zadań z konkretnymi wątkami lub na dynamicznym podkradaniu zadań (ang. *task stealing*).

Programowanie zorientowane na zadania jest więc pewną, stworzoną dla wygody, warstwą abstrakcji na dobrze znanej koncepcji puli wątków. Programowanie zadaniowe swoją prostotą ma zachęcać do częstego tworzenia wielu zadań, które w pełni będą wykorzystywać zasoby obliczeniowe maszyny, na której działają. Zadania z założenia są lekkie, a więc ich stworzenie i przełączanie wymaga mniejszych zasobów niż w przypadku wątków poziomu jądra [112]. Z drugiej strony, ich ogólna wydajność jest nieznacznie pomniejszana przez dodatkowy narzut, który pochodzi od działania samego środowiska uruchomieniowego. Można zatem powiedzieć, że zadania doskonale nadają się do rozwiązywania problemów złożonych z wielu dynamicznie organizowanych podproblemów (np. obsługa wielu zapytań w serwerze *HTTP*), natomiast gorzej sprawują się w przypadku, gdy liczba jednostek wykonawczych jest z góry znana (np. odczytywanie danych z kilku urządzeń).

W celu ukazania różnic wydajności przy tworzeniu wielu wątków poziomu jądra i wielu zadań (czyli wątków w przestrzeni użytkownika) wykonano prosty test z użyciem platformy *.NET*, która pozwala tworzyć zarówno wątki tradycyjne jak i zielone. W teście tworzono 10000 wątków/zadań, które po stworzeniu zliczały się i zawieszały w oczekiwaniu na zakończenie. W badaniu mierzono: ile wątków było widzianych z poziomu jądra systemu operacyjnego, jakie było użycie pamięci po stworzeniu wszystkich wątków/zadań oraz to, ile czasu trwało stworzenie wszystkich wątków/zadań. Programy uruchamiano za pomocą platformy *Mono*. Wyniki badania zamieszczono w tabeli 3.1.

Typ	**Liczba wątków**	**Użycie pamięci [MiB]**	**Czas tworzenia [s]**
Standardowe wątki	10003	316.20	10.70
Zadania na puli wątków	6	19.53	0.12

Tablica 3.1: Pomiary rzeczywistej liczby wątków widocznej dla systemu operacyjnego, użytej pamięci i czasu tworzenia dla dwóch aplikacji współbieżnych (wątkowej i zadaniowej) napisanych w *C# .NET*, tworzących po 10000 wątków/zadań.

Wyniki pokazują, że stworzenie 10000 jednostek obliczeniowych za pomocą zadań było ok. 100 razy szybsze i wymagało ponad 16 razy mniej pamięci niż za pomocą wątków. Znaczna różnica w wydajności wynika oczywiście z liczby wątków widocznych z poziomu systemu operacyjnego — 10003 w wersji wątkowej i zaledwie 6 w wersji zadaniowej. Nadmiarowe 3 wątki w wersji wątkowej pochodzą od wątku głównego i dodatkowych wątków *CLR* (najprawdopodobniej stworzonych przez odśmiecacz) — zapewne podobnie jest w wersji zadaniowej.

Tworzenie zadań z punktu widzenia programisty w zasadzie niczym nie różni się od tworzenia wątków — jest tak samo łatwe. W przypadku platformy *.NET* główna różnica polega na wykorzystaniu klasy `Task` zamiast klasy `Thread`. Programista

może zatem bardzo łatwo uzyskać zalety korzystania z zadań bez istotnych ingerencji w kod aplikacji. Przykładowe języki i platformy, które pozwalają tworzyć programy zorientowane zadaniowo to: *C++* (w standardzie *C++11* [110] lub dzięki bibliotece *Intel TBB* [51]), *.NET*, *Go*, *Haskell*, *Lua*, *Ruby*.

Decentralizacja przepływu

Klasyczne wątki ze względu na swoją niskopoziomową implementację i ograniczenia większości architektur sprzętowych nie oferują zwykle nic poza podstawowymi operacjami takimi jak np.: uruchomienia wątku, połączenie (ang. *join*) z innym wątkiem, odczytanie wartości zwracanej czy awaryjne zakończenie. Inaczej jest w przypadku zadań, które dzięki istnieniu wewnątrz własnego środowiska uruchomieniowego mogą udostępniać znacznie bardziej zaawansowane czynności, które umożliwiają decentralizację przepływu. Decentralizacja przepływu sterowania polega na kierowaniu przebiegiem programu za pomocą zadań niekoordynowanych przez żadną centralną jednostkę — istniejące i tworzone zadania zyskują pewną autonomię i działają w ramach dążenia do własnego celu.

Klasycznym przykładem mechanizmu decentralizacji przepływu są kontynuacje zaczerpnięte głównie z funkcyjnych języków programowania. Nieformalnie kontynuacja to po prostu zadanie, które zostaje uruchomione bezpośrednio po zakończeniu działania innego (określonego) zadania, niezależnie od tego czy wykonanie zakończyło się sukcesem czy np. wyjątkiem. Zadanie kontynuacyjne ma zwykle dostęp do wartości zwracanej oraz dodatkowych informacji nt. stanu zadania poprzedzającego. Kontynuacje są nieco podobne do powszechnie znanych wywołań zwrotnych, które są jednak często uważane za kłopotliwe ze względu na niejasny (asynchroniczny) kontekst występowania. Inną cechą odróżniającą wywołania zwrotne od zadań kontynuacyjnych jest to, że każda kontynuacja może mieć swoją kontynuację i tym sposobem możliwe jest tworzenie łańcuchów zadań następujących po sobie (tworzenie zagnieżdżonych wywołań zwrotnych jest niepraktyczne, a często niemożliwe). Dodatkową zaletą tej konstrukcji jest fakt, że najczęściej ma ona postać domknięcia (ang. *closure*), czyli ma dostęp do zmiennych lokalnych widocznych w miejscu deklaracji, co pozwala tworzyć zwięzły i elegancki kod. Kontynuacje mogą być użyte np. do implementacji warunków zakończenia (ang. *post–conditions*), dynamicznego tworzenia ciągów przetwarzania lub do obsługi błędów w wykonaniu zadania pierwotnego (nieco podobnie do wyjątków). Przykład prostej kontynuacji do zwalniania zasobów i logowania przedstawiono na rysunku 3.2.

Kontynuacje pozwalają zaimplementować inną konstrukcję decentralizującą — współprogramy (ang. *coroutines*). Współprogramy pozwalają na tworzenie wielu punktów wznowienia/zawieszenia działania zadania w określonych miejscach w kodzie. Mogą być zatem używane do tworzenia wywołań asynchronicznych. Bardzo niewiele języków wspiera jednak współprogramy, głównie ze względu na małe zainteresowanie programistów, którzy przyzwyczajeni są do pojedynczych punktów wejścia/wyjścia do/z zadania. *Coroutines* występują np. pod postacią słów kluczowych *async/await* na platformie *.NET 4.5* [75].

Poza kontynuacjami istnieje wiele innych operacji dostępnych dla zadań — ich zbiór zależy jednak od konkretnej platformy i ciężko wyróżnić kanoniczną grupę dostępną w każdym języku i bibliotece. Ciekawym przykładem mechanizmu decentralizacji przepływu jest konstrukcja *once*, która pozwala zagwarantować, że niezależnie od tego ile wątków spróbuje wywołać funkcję oznaczoną jako *once*, wykona się ona tylko jeden raz. Technika ta doskonale nadaje się do inicjowania danych

```
public Client GetClient() { ... }

public Status HandleClient(Client client) { ... }

public void Log(Client client, Status status) { ... }

public void HandleClients()
{
  while (true)
  {
    var client = GetClient();
    clients.Add(client);
    var task = new Task<Status>(() => HandleClient(client));
    task.ContinueWith(ancestor => {
                        clients.Remove(client);
                        Log(client, ancestor.Result);
                        });
    task.Start();
   }
}
```


Rysunek 3.2: Przykład użycia kontynuacji do zwolnienia zasobów i zarejestrowania wyników w wielozadaniowym serwerze napisanym w *C# .NET*.

współdzielonych pomiędzy wszystkimi wątkami bezpośrednio przed rozpoczęciem działania.

Jeszcze innym przykładem konstrukcji decentralizujących przepływ są *when–all*/*when–any* (alternatywnie *wait–all*/*wait–any*), które pozwalają na uruchomienie nowego zadania w momencie, gdy wszystkie/którykolwiek ze wskazanych zadań zakończy działanie. Operacje te pozwalają w prosty i wygodny sposób wyrazić relację następstwa. Aby pokazać zastosowanie tych technik poprawiono z ich wykorzystaniem błędny kod przedstawiony na rysunku 1.8. Przykładową implementację wykonano w *C# .NET* ze względu na dostępność omawianej konstrukcji, jednak mogłaby być ona zrealizowana także np. w *C++* z odpowiednimi bibliotekami. Wynikowy kod znajduje się na rysunku 3.3.

```
void js_DestroyContext(...) { ... }

void js_UnpinPinnedAtoms(...) { ... }

void init_Tasks()
{
  var tasks = new Task[N];
  for (int i = 0; i < N; ++i)
  {
    tasks[i] = new Task(() => js_DestroyContext(...));
    tasks[i].Start();
  }
  new Task(() => {
             Task.WaitAll(tasks);
             js_UnpinPinnedAtoms(...);
             }).Start();
}
```


Rysunek 3.3: Przykład użycia konstrukcji *wait–all* do naprawienia błędu z projektu *Mozilla*, który polegał na przedwczesnym zwolnieniu pamięci.

Widać zatem, że rozbudowana semantyka, którą oferuje programowanie zorientowane na zadania pozwala wyrazić wiele różnych zależności w sposób zwięzły, elegancki i co najważniejsze — skuteczny. Mogłoby się wydawać, że powyższe

konstrukcje nie są w żaden sposób innowacyjne jeśli chodzi o ideę, ale należy zauważyć, że bardzo mało bibliotek wątków oferuje jakiekolwiek wysokopoziomowe operacje porównywalne do omówionych technik. Można toteż uznać, że programowanie zorientowane na zadania jest istotnym krokiem ku poprawnym programom współbieżnym.

3.3.2. Schematy komunikacji

Tradycyjnym sposobem komunikacji wątków czy zadań jest pamięć współdzielona. Wykorzystanie tego medium wymiany danych jest bardzo proste, efektywne i naturalne — wynika wprost z architektury dzisiejszych komputerów. Pamięć współdzielona, jak wielokrotnie już wspominano, pociąga jednak za sobą wiele niebezpieczeństw wynikających z konieczności precyzyjnej synchronizacji dostępu. Poprawne wykorzystywanie tego sposobu komunikacji jest zatem zadaniem trudnym i podatnym na błędy. Z tego powodu w dzisiejszych programach współbieżnych i rozproszonych coraz częściej wraca się do historycznych modeli komunikacji takich jak pamięć transakcyjna czy przekazywanie wiadomości, które niegdyś nie zyskały popularności głównie ze względu na ograniczenia zasobów sprzętowych. Nowoczesne podejście do tych metod współdziałania zadań i wątków przedstawiono w niniejszej sekcji.

Pamięć transakcyjna

Najbardziej popularną metodą synchronizacji dostępu do pamięci współdzielonej jest wykorzystanie blokad. Technika ta ma jednak kilka istotnych wad. Przede wszystkim wymaga ona rozważania i odpowiedniego chronienia sekcji programu zlokalizowanych bardzo często w odległych, niezwiązanych fragmentach kodu. Jest to trudne i bardzo podatne na błędy wynikające z pominięcia. Ponadto z blokadami wiążą się problemy naruszenia żywotności takie jak zakleszczenia czy zjawiska typu *livelock*.

Rozszerzeniem idei pamięci współdzielonej, pozbawionej jednak wyżej wspominanych wad, jest pamięć transakcyjna. Pamięć transakcyjna to mechanizm komunikacji i synchronizacji realizujący w sposób transakcyjny operacje na współdzielonej pamięci operacyjnej w sposób bardzo podobny do tego znanego z baz danych (oferuje własności atomowości, spójności, izolacji, ale nie trwałości). W przeciwieństwie do blokad, pamięć transakcyjna stosuje optymistyczny (nieblokujący) schemat działania. Każdy wątek wykonuje swoje operacje nie zważając na to, co robią inne wątki, rejestrując przy tym każdy odczyt i zapis w odpowiednim dzienniku. Po zakończeniu wszystkich działań weryfikowane jest, czy interferowały one z innymi operacjami na rozważanym obszarze pamięci — jeśli tak, to transakcja musi zostać przerwana (ang. *abort*), a skutki jej działania odwrócone (ang. *rollback*); w przeciwnym wypadku transakcja może zostać zatwierdzona (ang. *commit*). Możliwe są także inne, równoważne implementacje wyżej opisanego schematu. Niezależnie jednak od sposobu realizacji, pamięć transakcyjna gwarantuje, że wykonywane operacje albo wykonają się w całości albo wcale (atomowość), system po wykonaniu transakcji nie będzie naruszał żadnej zasady integralności (spójność) oraz że żadne dwie transakcje nie będą operowały na stanach pośrednich (izolacja).

Zaletą stosowania pamięci transakcyjnej jest zwiększona współbieżność poprzez eliminację blokowania, co przyczynia się do zwiększenia ogólnej wydajności programu. Może jednak zdarzyć się, że przy wielu konfliktach i narzutach związanych

z rejestrowaniem i wykonywaniem operacji *abort* i *rollback* narzut wydajnościowy będzie większy niż w przypadku drobnoziarnistej, precyzyjnej synchronizacji za pomocą blokad. Zbadano jednak, że w typowym przypadku pogorszenie wydajności jest co najwyżej dwukrotne [53].

Zwiększona współbieżność nie jest największą zaletą pamięci transakcyjnej. Jak wspomniano w rozdziale 1, możliwe jest uniknięcie aż do 81% błędów stosując omawianą technikę [69]. Pamięć transakcyjna może zatem sprzyjać tworzeniu niezawodnych programów współbieżnych. Niestety, mechanizm ten nie jest jeszcze powszechnie wykorzystywany, a zatem trudno ocenić czy te teoretyczne rozważania miałby faktyczne przełożenie na rzeczywistość.

Wadą omawianej techniki jest brak możliwości wykonywania transakcyjnie operacji wejścia/wyjścia ze względu na semantykę operacji *rollback* — większość interakcji ze sprzętem (np. wypisanie ciągu znaków na ekranie) jest niemożliwa do odwrócenia. Niektóre języki programowania (np. *Haskell*) statycznie — podczas kompilacji sprawdzają poprawność kodu transakcji ze względu na brak operacji wejścia/wyjścia.

Innym istotnym problemem jest sama implementacja pamięci transakcyjnej. Bardzo często nie istnieje możliwość bibliotecznej realizacji transakcji w pamięci — potrzebne jest wsparcie kompilatora lub środowiska uruchomieniowego. Ponadto specjalnej obsługi i dodatkowych zabiegów wymagają sytuacje, gdy błąd na skutek przeplatających się transakcji powoduje awarię (np. naruszenie ochrony pamięci) lub naruszenie żywotności (np. nieskończoną pętlę).

```
type Score = Int

data Goal = Goal { completed :: Bool
                 , reward :: Score }

handlePlayerGoal :: TVar Score -> TVar Goal -> IO ()
handlePlayerGoal score goal = atomically $ do
  uscore <- readTVar score
  ugoal <- readTVar goal
  unless (completed ugoal) $ do
    writeTVar score (uscore + reward ugoal)
    writeTVar goal Goal { completed = True, reward = reward ugoal }
```

Rysunek 3.4: Przykład wykorzystania pamięci transakcyjnej w języku *Haskell* do poprawienia błędnej funkcji z gry *Forumwarz* pozwalającej na zwielokrotnienie liczby punktów.

Przykład wykorzystania pamięci transakcyjnej do poprawienia błędnej funkcji z gry *Forumwarz* (zob. podrozdział 2.2.3) przedstawiono na rysunku 3.4. W rozważanym przykładzie cały region funkcji oznaczono jako atomowy (wywołanie `atomically`) — wykonywany transakcyjnie. Wewnątrz transakcji odczytywane, a następnie zapisywane są dwie zmienne transakcyjne (`score` oraz `goal`) będące instancją algebraicznego typu `TVar`. Dzięki zorganizowaniu transakcji wielokrotne wywołania funkcji z tymi samymi argumentami nigdy nie spowodują wielokrotnego zliczenia punktów. Pakiet `stm` języka Haskell proponuje również interesujące rozszerzenia koncepcji pamięci transakcyjnej takie jak semantyka *retry* i *orElse*, które pozwalają wykorzystać omawianą technikę do większej grupy problemów [43]. Różne implementacje pamięci transakcyjnej można znaleźć dla praktycznie każdego

popularnego języka programowania, jednak niewiele z nich dorównuje łatwością użycia tej znanej z języka *Haskell*.

Powyższe rozważania dotyczą programowej pamięci transakcyjnej (ang. *software transactional memory, STM*), jednak pierwsza koncepcja [58] pamięci transakcyjnej z 1986 roku proponowała sprzętową jej realizację. W ówczesnych czasach było to trudne ze względu na ograniczenia technologiczne i z tego powodu idea ta ewoluowała do postaci programowej pamięci transakcyjnej [45, 103]. Dopiero niedawno producenci sprzętu wrócili do pomysłu sprzed blisko 30 lat i ponownie podjęto próby stworzenia architektury ze sprzętową pamięcią transakcyjną. Przykładem sprzętowej realizacji opisywanej koncepcji jest *Intel Transactional Synchronization Extensions* (*TSX*), które mają zostać wprowadzone w kolejnej generacji procesorów [24]. Warto przy tym nadmienić, że trwają obecnie prace nad wdrożeniem *TSX* do biblioteki *glibc* [57], co umożliwi powszechne zastosowanie sprzętowej pamięci transakcyjnej. Wszystko to daje nadzieję, że pamięć transakcyjna, zarówno w formie programowej jak i sprzętowej, doczeka się w końcu wdrożenia w programach współbieżnych i sprawi, że ich tworzenie będzie łatwiejsze.

Przekazywanie wiadomości: CSP i model aktorów

Kolejną alternatywą do pamięci współdzielonej, dobrze znaną z systemów rozproszonych [111] i systemów operacyjnych z mikrojądrem [112], jest przekazywanie wiadomości (lub wymiennie — komunikatów). Technika ta opiera się na bardzo prostej idei — procesy, wątki czy też zadania nie komunikują się poprzez współdzielenie (jak w przypadku pamięci współdzielonej), ale współdzielą poprzez komunikację. Komunikacja, co wynika wprost z nazwy, odbywa się poprzez przekazywanie wiadomości (zawierających zwykle dowolne dane) pomiędzy jednostkami obliczeniowymi. W ten sposób operują one jedynie na danych, które zostały im jawnie przekazane. W programowaniu współbieżnym zwyczajowo wyróżnia się dwie odmiany przekazywania wiadomości: CSP oraz model aktorów.

CSP (ang. *Communicating Sequential Processes*) [47] to klasyczna koncepcja opracowana w 1978 przez Tony'ego Horae, która od lat pozostaje jedną z najważniejszych prac w dziedzinie informatyki[2]. Na koncepcję CSP składa się kilka założeń:

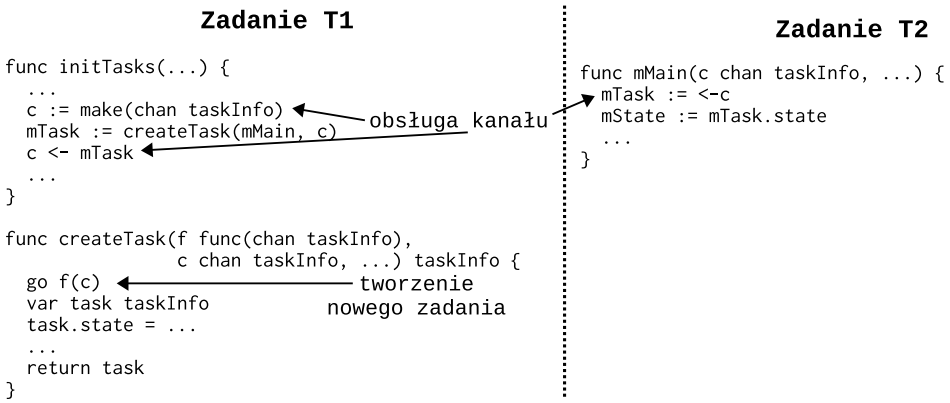
— CSP zakłada istnienie wielu anonimowych, sekwencyjnych programów,
— programy działają w sposób od siebie niezależny,
— programy komunikują się za pomocą wiadomości,
— przekazywanie wiadomości odbywa się w trybie *randezvous*, w sposób synchroniczny — nadawca nie może przekazać wiadomości, jeśli odbiorca nie jest gotowy na jej odebranie,
— wiadomości przesyłane są poprzez nazwane, jawne kanały.

Z CSP związana jest dosyć złożona algebra, która jest użyteczna w modelowaniu zachowań jednostek obliczeniowych, jednak w praktyce jej znajomość nie jest niezbędna do tworzenia poprawnych programów. Ta wiedza może być jednak pomocna w wykrywaniu zakleszczeń, które są możliwe w modelu CSP.

CSP wpłynęło na model współbieżności wielu języków programowania m.in. *Limbo* [22] i *Go* [41]. Przykład wykorzystania komunikacji w stylu CSP w języku *Go* zaprezentowano na rysunku 3.5. W tym przykładzie, aby rozwiązać opisany

[2] Drugie miejsce w rankingu najczęściej cytowanych prac z dziedziny informatyki wg *CiteSeer*. Stan na 23.04.2013.

wcześniej (zob. podrozdział 1.2.2) problem naruszenia kolejności, stworzono kanał komunikujący zadania tworzące i tworzone, przez który przekazywane są informacje na temat stanu. W ten sposób odczyt z kanału synchronizuje dostęp do danych tak, aby zadanie tworzone odczytywało informacje w odpowiedniej kolejności.



Rysunek 3.5: Przykład wykorzystania komunikacji w stylu CSP do naprawienia problemu w oprogramowaniu *Mozilla*.

Nieco inną koncepcją przekazywania wiadomości jest model aktorów opracowany jeszcze przed CSP w 1973 roku [46] na potrzeby metod sztucznej inteligencji. Technika ta opiera się na idei aktorów — prymitywów współbieżnych, które na podstawie odbieranych wiadomości mogą podjąć decyzję lokalną, np. stworzyć kolejnych aktorów lub wysłać wiadomość. Ważną różnicą pomiędzy modelem aktorów, a CSP jest fakt, że procesy w CSP są anonimowe, natomiast aktorzy identyfikowani są przez nazwane punkty (skrzynki pocztowe (ang. *mailboxes*)), które mogą gromadzić wiadomości. Aktor może wymieniać więc wiadomości tylko z aktorem, którego adres posiada (może on znać go *a priori* lub otrzymać w pewnej wiadomości). Pomiędzy aktorami nie istnieje zatem żadna jawna topologia, która w CSP wyznaczana jest poprzez kanały komunikacyjne. Co istotne, przekazywanie wiadomości w modelu aktorów odbywa się, w przeciwieństwie do CSP, całkowicie asynchronicznie — punkt odbiorczy nie musi być gotowy do odbierania, aby nadawanie się powiodło. Omawiana technika jest zatem mniej podatna na zakleszczenia niż CSP.

Model aktorów jest z powodzeniem wykorzystywany w językach takich jak *D*, *Erlang*, *Rust* czy *Scala*, w systemach rozproszonych, a także w rozwiązaniach typu *cloud*. Przykład prostego aktora napisanego w języku *Scala* przedstawiono na rysunku 3.6.

3.3.3. Nietrywialne konstrukcje synchronizujące

Standardowe biblioteki wątków dostarczają zwykle bardzo wąskiego zbioru mechanizmów synchronizacji. Dla przykładu *Pthreads* [64] udostępnia: muteksy, zmienne warunkowe, bariery, semafory, semafory r/w i blokady wirujące. Konstrukcje te doskonale nadają się i w zupełności wystarczają do stworzenia prostej synchronizacji, jednak nierzadko zdarzają się sytuacje, w których przydatne byłyby mechanizmy bardziej wyrafinowane. W takich przypadkach warto skorzystać z bardziej zaawansowanych metod synchronizacji, które lepiej dostosowane są do rozważanego problemu.

```
class Ping extends Actor {
  def act() {
    loop {
      react {
        case Ping =>
          sender ! Pong
        case Stop =>
          exit()
      }
    }
  }
}
```

Rysunek 3.6: Prosty aktor odpowiadający obiektem `Pong` na każdą wiadomość typu `Ping` i kończącym działanie w przypadku odebrania wiadomości `Stop`.

Operacje atomowe

Przykładem bardzo prostej, ale rzadko implementowanej w przestrzeni użytkownika konstrukcji są operacje atomowe. Operacje atomowe pozwalają w sposób niepodzielny wykonać pewne proste czynności takie jak dodanie dwóch liczb, inkrementacja czy zamiana wartości dwóch zmiennych. Dobrym przykładem uzasadnionego użycia tej techniki jest dynamiczny podział tablicy dla wielu wątków, co przedstawiono na rysunku 3.7.

```
void process_event(Event *event);

void event_processor()
{
  ...
  while (true) {
    idx = __sync_fetch_and_add(&cursor, 1); // x86-64: lock xadd %rax,0x20046c(%rip)
    if (idx >= N)
      break;

    process_event(&events[idx]);
  }
  ...
}
```

Rysunek 3.7: Przykład użycia operacji niepodzielnego pobrania z dodawania do dynamicznego podziału tablicy pomiędzy wiele procesorów zdarzeń. Na architekturze *x86–64* kompilator *gcc* udostępniający tę operację wygenerował instrukcję *lock xadd*.

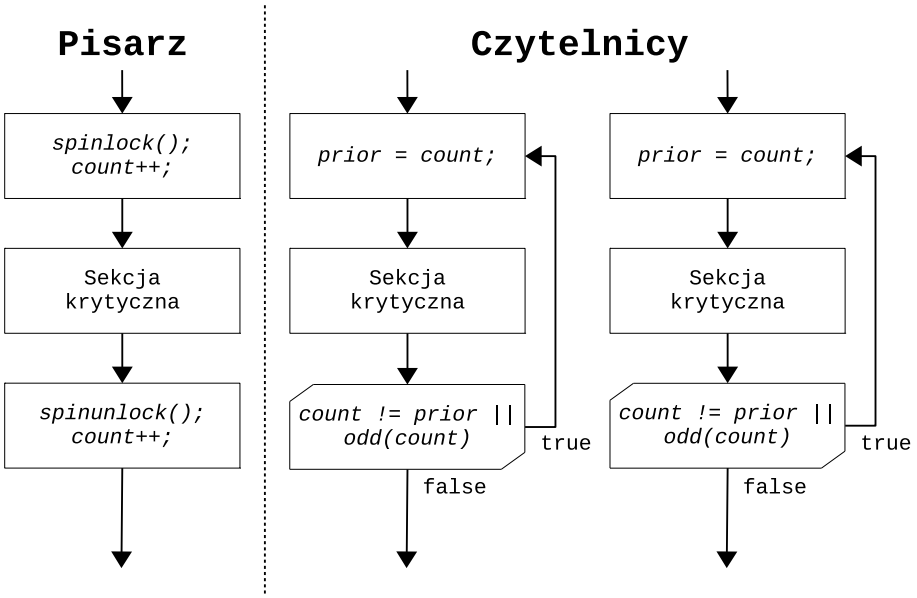
Programiści bardzo często w nieuzasadniony sposób zakładają, że działania takie jak np. przypisanie wykonują się w sposób niepodzielny. Jest to faktem na wielu architekturach, ale niektóre konfiguracje dzielą tę operację na dwie mniejsze (np. dotyczące bitowych połówek całej zmiennej), co może powodować niespójne odczyty w innych wątkach. Zawsze zatem należy wykorzystywać warstwę pośrednią (np. udostępnianą przez kompilator w postaci makr), aby mieć gwarancję poprawności.

Operacje niepodzielne są wielokrotnie szybsze od prostych zmiennych chronionych semaforem wykluczającym, a więc należy je stosować zawsze, gdy jest to możliwe. Niestety, ze względu na oczywiste problemy z przenośnością, działania atomowe bardzo rzadko włączane są do bibliotek standardowych i często dostępne

są jedynie jako rozszerzenia standardów. Wsparcie dla operacji atomowych oferowane jest m.in. przez kompilator *gcc*, w standardzie *C++11*, na platformie *.NET* oraz *J2SE*.

Blokady sekwencyjne

Blokady sekwencyjne (ang. *seqlock, sequential lock*) to specjalny mechanizm synchronizacji dla problemu czytelników i pisarzy, podobny nieco do standardowego semafora r/w, jednak usuwający problem głodzenia pisarzy. Cechuje się on bardzo wysoką wydajnością, ponieważ nie wykorzystuje blokujących konstrukcji synchronizacyjnych (jak np. semafory), a jedynie blokady wirujące. Jego wykorzystanie wymaga jednak, aby sekcja krytyczna czytelników była idempotentna — mogła być powtarzana wielokrotnie bez efektów ubocznych, co w wielu zastosowaniach może być kłopotliwe.



Rysunek 3.8: Zasada działania mechanizmu synchronizacji *seqlock*. Na podstawie: [104].

Idea działania *seqlocks* jest bardzo prosta. Pisarze przed wejściem i po wyjściu ze swojej sekcji krytycznej wykonują taką samą operację — inkrementują licznik związany z egzemplarzem blokady. Zwiększanie o jeden jest dodatkowo powiązane z założeniem blokady wirującej, która synchronizuje pracę pisarzy. Czytelnicy natomiast nie zajmują żadnych blokad, a przed wejściem do sekcji krytycznej zapisują jedynie aktualną wartość licznika w zmiennej. Jeśli po zakończeniu działania na zasobach współdzielonych aktualna wartość licznika nie jest równa wartości przechowywanej lub jest ona nieparzysta, oznacza to, że odczytane dane mogą być niespójne, gdyż operował (bądź operuje) na nich wątek pisarza. W takiej sytuacji należy powtórzyć sekcję krytyczną od nowa. Blokady sekwencyjne są powszechnie wykorzystywane w jądrze *Linuksa* [61, 63].

Futures

Futures [3] to konstrukcja synchronizacyjna będąca obiektem pośrednim dla wyniku obliczenia, które nie zostało jeszcze zakończone [39]. *Futures* pozwalają więc uruchomić obliczenia asynchronicznie pozostawiając w wątku wywołującym uchwyt do wyniku tych obliczeń. Jeśli wartość *future* (wynik działania wątku) zostanie odczytana przed zakończeniem asynchronicznych obliczeń, to wątek odczytujący zostanie zawieszony w oczekiwaniu na zakończenie działania; w przeciwnym wypadku wynik zostanie zwrócony natychmiastowo. *Futures* pozwalają zatem na traktowanie wywołań asynchronicznych w sposób niemalże całkowicie przezroczysty (taki sam jak obliczeń synchronicznych) dla programisty. Ich wykorzystanie może ułatwiać zatem utylizację zasobów obliczeniowych środowiska wieloprocesorowego.

Futures występują w wielu odmianach. Mogą one przybierać postać jawną (zwykle implementowaną bibliotecznie), czyli taką która wymaga deklaracji bądź domniemaną (jako element języka), gdy każda zmienna jest potencjalnie konstrukcją typu *future* i każdy odczyt może być blokujący. Inna zmienna cecha, to synchroniczność i asynchroniczność — niektóre implementacje zamiast blokować generują błąd lub wyjątek, gdy wynik w obiekcie *future* nie jest jeszcze gotowy. Wybrane platformy oferują także *futures* tylko do odczyty, gdy inne pozwalają na wielokrotny zapis wartości.

Koncepcję *futures* zaprezentowano już przy okazji klasy `Task` (zob. podrozdział 3.3.1) dostępnej na platformie *.NET* — pobranie wyniku działania zadania jest bardzo podobne do odczytania wyniku obiektu *future*. Inne platformy udostępniające koncepcję *futures* to m.in. *C++11* (szablon `std::future`), *Java* (klasa generyczna `Future`) czy *Haskell* (typ `MVar` i `IVar`).

Inne

Operacje atomowe, blokady sekwencyjne i *futures* nie wyczerpują oczywiście repertuaru dostępnych mechanizmów synchronizacji, a jedynie reprezentują małą próbkę tego, co można odnaleźć na różnych platformach programistycznych. Wśród istotnych technik warto wymienić jeszcze: *RCU* [61], wymiennik (ang. *exchanger*) [39], wszelkiego rodzaju zatrzaski (ang. *latch*) [39] czy regiony krytyczne [51]. Mnogość różnych konstrukcji nie pozwala ich wszystkich opisać, jednak tworząc program współbieżny warto być zaznajomionym z metodami synchronizacji oferowanymi przez platformę i wtedy gdy to uzasadnione stosować je, aby tworzyć zwięzły, prostszy i wydajniejszy kod.

3.3.4. Schematy implementacyjne

Analizując różne programy współbieżne można zauważyć pewne wielokrotnie powtarzane schematy implementacyjne wykorzystywane przez programistów. Z tego powodu projektanci współczesnych technik wspierających programowanie współbieżne coraz częściej włączają owe schematy do tworzonych bibliotek i mechanizmów tak, aby nie musiały być one implementowane za każdym razem od nowa. Wykorzystywanie dostępnych szkieletów ułatwia, a tym samym przyspiesza tworzenie oprogramowania, a także zmniejsza prawdopodobieństwo wystąpienia błędu.

[3] Często wymiennie do *future* wykorzystuje się nazwę *promise* lub *delay*. Nie istnieje jednak powszechna zgoda co do tego, czy pojęcia te rzeczywiście znaczą to samo [72].

Współbieżne (kooperatywne) struktury danych

Każdy nietrywialny program składa się z algorytmów i struktur danych. Obydwa te elementy muszą zostać zaprojektowane w szczególny sposób, jeśli mają wykorzystywać zalety wieloprocesorowych jednostek obliczeniowych. Algorytmy współbieżne muszą zakładać dekompozycję podstawowego problemu na mniejsze zadania, natomiast struktury danych powinny być przystosowane do współdzielenia poprzez właściwą synchronizację. O ile ciężko odnaleźć odpowiednie uogólnienia dla wszystkich algorytmów, o tyle podstawowe struktury danych takie jak listy, kolejki, tablice asocjacyjne we wszystkich programach są podobne. W związku z powyższym w wielu bibliotekach struktur danych można odnaleźć implementacje przygotowane specjalnie dla programowania współbieżnego.

Oczywiście istnieje możliwość wykorzystania praktycznie każdej struktury danych w programie współbieżnym — wystarczy zapewnić wzajemne wykluczanie każdej wykonywanej operacji. Takie podejście jest jednak bardzo podatne na błędy związane z pominięciem zajęcia lub zwolnienia zasobów w miejscu użycia. Można zatem synchronizację umieścić w kodzie wykonywanych operacji implementując tym samym wariant wzorca monitora. Ta metoda pomimo wysokiej skuteczności jest bardzo nieefektywna, bo wiele operacji może zostać zrealizowanych w sposób nieblokujący, co może mieć istotny wpływ na przepustowość procedur dostępnych dla struktury. Przykładem może być funkcja pobierająca liczbę elementów przechowywanych w pewnym kontenerze — przy odpowiedniej implementacji (z użyciem działań niepodzielnych) operacja ta może być nieblokująca.

Stosowanie wzorca monitora sprzyja zatem konkurencji, a nie współpracy pomiędzy zadaniami współdzielącymi daną strukturę. Z tego powodu w nowoczesnych bibliotekach współbieżne struktury danych implementowane są w możliwie najwydajniejszy sposób tak, aby wzajemne blokowanie było zjawiskiem możliwie rzadkim. Ponadto współbieżne odpowiedniki sekwencyjnych struktur danych bardzo często posiadają nieco inny, dostosowany do środowiska pracy, zbiór dostępnych operacji. Niektóre z nich, niemożliwe do zrealizowania w wydajny czy bezpieczny sposób, są usuwane, inne są natomiast dodawane. Współbieżne struktury danych bardzo często oferują procedury typu `Try*`, które pozwalają na wykonanie standardowej operacji w sposób nieblokujący — sytuacja, która wymaga blokowania powoduje zwrócenie błędu. Innym, ważnym dodatkiem do omawianego rodzaju struktur danych jest łączenie niektórych operacji tak, aby były one niepodatne na błędy typu *TOCTOU*. Przykładem takiej operacji może być np. sprawdzenie czy element istnieje i wykonanie na tym elemencie transformacji w sposób niepodzielny. W celu zwiększenia wydajności często zdarza się, że biblioteki udostępniają struktury z semantyką *Copy–on–Write* (np. `CopyOnWriteArrayList` na platformie *JVM*), która umożliwia bezkolizyjne współdzielenie aż do momentu, gdy nie są wykonywane operację modyfikujące strukturę.

Dynamicznie rozwijającą się dziedziną badań są całkowicie nieblokujące struktury danych o bardzo szybkim dostępie, które mają kluczowe znaczenie m.in. w szybkim handlu na giełdzie (ang. *high–frequency trading, HFT*). Budowa takich struktur jest jednak niezwykle trudna, a ze względu na bardzo ograniczony zbiór dostępnych operacji nie mogą być one używane we wszystkich projektach. Obecnie najpopularniejsze nieblokujące struktury oparte są na buforach cyklicznych [65].

Konstruując dowolny system współbieżny warto zatem pamiętać i wszędzie tam, gdzie to możliwe, wykorzystywać kooperatywne struktury danych. Dzięki temu

możliwe jest uniknięcie wielu błędów związanych z niewłaściwą synchronizacją, a także zachowanie wysokiej przepustowości, a tym samym wydajności.

Wzorce synchronizacji

Jak wspomniano wcześniej, trudno o dobre uogólnienia dla algorytmów stosowanych w programach współbieżnych. Niemniej jednak istnieje pewna klasa problemów, które powtarzają się często i noszą one wspólne cechy. Przykładami mogą tu być problemy: ucztujących filozofów, czytelników i pisarzy oraz producentów i konsumentów. Prawidłowość ta została dostrzeżona przez twórców bibliotek wspierających współbieżność i coraz częściej można spotkać się z gotowymi implementacjami wzorców rozwiązań tych klasycznych zagadnień. Implementacje te najczęściej oferują całą logikę synchronizacji pozostawiając programiście jedynie stworzenie kodu realizującego właściwe przetwarzanie.

Dobrym przykładem nowoczesnej biblioteki realizującej wzorce synchronizacji jest *.NET*, która poprzez klasę `BlockingCollection` umożliwia bardzo łatwe rozwiązanie problemu producentów i konsumentów. Klasa ta poprzez udostępnione metody realizuje najważniejsze zadania takie jak:
— blokowanie w przypadku zapełnienia kontenera (blokujące `Add()`)
— blokowanie w przypadku pustego kontenera (blokujące `Take()`)
— sygnalizacja/sprawdzenie zakończenia pracy producentów (`CompleteAdding()`, `IsAddingCompleted`)
— sprawdzenie zakończenia pracy producentów i konsumentów (`IsCompleted`).
Ponadto klasa ta umożliwia m.in. tworzenie iteratorów konsumujących, semantykę *try*, a także równoważenie obciążenia poprzez zastosowanie wielu kontenerów i metod `TakeFromAny()` oraz `AddToAny()`.

Gotowe implementacje wzorców synchronizacji wydatnie zmniejszają ilość kodu potrzebną do realizacji zadania. Co ważniejsze, pozwalają one uniknąć podstawowych błędów, podobnie jak kooperatywne struktury danych. Niestety, nie we wszystkich bibliotekach wspierających współbieżność można odnaleźć gotowe do wykorzystania schematy synchronizacji. Należy się jednak spodziewać, że z czasem sytuacja ta się poprawi.

Zrównoleglanie w przepływie sterowania

Do ostatniej omówionej techniki zaliczają się wszystkie metody pozwalające na proste zrównoleglanie pętl, instrukcji warunkowych i wszelkich innych elementów kierujących ścieżką wykonania programu. Bardzo często zdarza się, że w programie istnieje potencjał na przyspieszenie wykonania poprzez równoległe wykonanie pętli dla kolejnych wartości iteratora. Wprowadzenie tam wątków w sposób jawny jest pracochłonne i niewygodne i z tego powodu programiści bardzo często rezygnują z tej możliwości. Nowoczesne kompilatory i biblioteki udostępniają jednak odpowiednie makra, dyrektywy preprocesora, adnotacje czy po prostu funkcje, które pozwalają w sposób niejawny zlecić wykonanie pętli (czasem także instrukcji warunkowych i innych elementów sterowania) w sposób współbieżny. Dzięki temu nawet niedoświadczony programista, minimalnym wysiłkiem może tworzyć aplikacje wykonujące obliczenia równoległe.

Wygoda stosowania tych metod wiąże się jednak z ich niewielkimi możliwościami — kompilator czy środowisko uruchomieniowe nie zawsze rozpoznaje właściwy sposób zrównoleglenia, ponadto praktycznie nigdy współbieżne fragmenty kodu nie mogą operować na wartościach współdzielonych. Mechanizmy tego typu są

zatem użyteczne i skuteczne dla prostych programów, a w przypadku bardziej skomplikowanych wymagane jest ręczne skonstruowanie struktury wątków czy zadań. Przykładem realizacji omawianych technik są *.NET Parallel Extensions* [78] oraz *OpenMP* [94]. Przykład zrównoleglenia pętli za pomocą *OpenMP* przedstawiono na rysunku 3.9.

```
#pragma omp parallel for
for (std::size_t i = 0; i < data_size / sizeof(std::uint64_t); ++i
{
  cipher[i] = blowfish.encryptBlock(*(block + i));
}
```

Rysunek 3.9: Zrównoleglenie pętli programu szyfrującego za pomocą dyrektywy `#pragma omp parallel for` standardu *OpenMP*.

3.4. Języki programowania

Kolejną, zdecydowanie najbardziej wymagającą, a przez co być może najskuteczniejszą, metodą tworzenia niezawodnych programów współbieżnych jest wykorzystywanie nie tylko odpowiednich technik i mechanizmów, ale całych języków i platform zaprojektowanych z myślą o tworzeniu oprogramowania współbieżnego. Wiele tradycyjnych, istniejących od lat języków takich jak np. *C* czy *C++*, ze względu na uwarunkowania historyczne, było projektowanych dla programowania sekwencyjnego. Zmieniające się realia wymusiły na komitetach opiekujących się takimi językami dostosowanie ich do programowania współbieżnego przy zachowaniu pełnej zgodności wstecznej, co najczęściej powodowało, że otrzymywane wsparcie, obarczone wieloma kompromisami, niekoniecznie prezentowało najwyższą jakość.

Sytuacja ta, począwszy od połowy lat dziewięćdziesiątych (od powstania języka *Java*), zaczęła się jednak zmieniać — projektanci języków dostrzegli, jak ważnym zagadnieniem w programowaniu jest współbieżność i stała się ona istotnym elementem rozważań nad ogólnym kształtem powstających platform. W niniejszej sekcji przedstawiono istotne trendy w powstających i popularnych językach dedykowanych programowaniu współbieżnemu.

3.4.1. Języki z wbudowaną współbieżnością

Pierwszą klasą języków programowania zorientowanych na współbieżność, o których warto wspomnieć, są języki, w których współbieżność jest pełnoprawnym składnikiem syntaktyki i semantyki, a nie tylko elementem biblioteki standardowej. W językach tych programowanie współbieżne jest łatwe, a dostępne konstrukcje wręcz zachęcają do tworzenia wielu wątków czy zadań. Bardzo często istotnym elementem ich składni są także pewne konstrukcje lub wzorce synchronizacyjne, które ułatwiają procedurę tworzenia oprogramowania poprzez pokazywanie właściwego i skutecznego podejścia do problemów współdzielenia danych.

Przykładem języka z wbudowaną współbieżnością jest minimalistyczny, strukturalny język *Go* [40] stworzony przez firmę *Google*. Współbieżność w *Go* opiera się na tzw. *goroutines*, które pełnią rolę zadań (rozumianych jako jednostki obliczeniowe w programowaniu zorientowanym na zadania) i są uruchamiane za pomocą prostej instrukcji `go`. Stworzenie nowego zadania w języku *Go* jest zatem bardzo proste.

Dla porównania na rysunku 3.10 przedstawiono tworzenia nowej niezależnej jednostki obliczeniowej w *Go* i w *Javie*. Jak widać, kod umożliwiający wielowątkowe wykonanie w języku *Go* jest możliwie krótki (jedynie trzy znaki dłuższy niż samo wywołanie funkcji), a przy tym czytelny i elegancki. Takie podejście ułatwia tworzenie i utrzymanie współbieżnego kodu nawet złożonych projektów.

Go

```
go someFunc(0, 42)
```

Java

```
new Thread(new Runnable() {
  @Override
  public void run() {
    someFunc(1, 42);
  }
}).start();
```

Rysunek 3.10: Porównanie długości kodu tworzącego odrębną jednostkę obliczeniową w językach *Go* i *Java*.

Zalecanym sposobem współdzielenia danych w *Go* jest przedstawiony wcześniej model CSP (zob. podrozdział 3.3.2), który wymaga od programisty dyscypliny w tworzeniu i zarządzaniu kanałami. W celu utrzymania prostoty i minimalizmu języka, projektanci pozostawili jednak możliwość współdzielenia danych poprzez klasyczne zmienne globalne chronione standardowymi mechanizmami synchronizacyjnymi. Bezpieczeństwo kodu współbieżnego jest zatem opcjonalne, a nie obowiązkowe.

Przykładem innego języka współbieżnego może być *Ada* [2] zaprojektowana na zamówienie Departamentu Obrony Stanów Zjednoczonych (ang. *United States Department of Defense*). *Ada* jest wieloparadygmatowym językiem skoncentrowanym na bezpieczeństwie z bardzo rozwiniętą współbieżnością. Dzięki surowej specyfikacji, statycznemu, silnemu typowaniu oraz zaawansowanemu kompilatorowi wiele błędów jest wykrywanych już na etapie kompilacji. Identyfikowanie pozostałych usterek (np. użycie niezalokowanej pamięci czy przepełnienie bufora) wykonywane jest natomiast przez wyrafinowane środowisko uruchomieniowe.

Przyjęty model współbieżności wydaje się bardzo przemyślany. Pojedynczą jednostką obliczeniową w języku *Ada* jest zadanie, które może posiadać odpowiadający wątek systemu operacyjnego lub może być zarządzane przez wewnętrznego planistę. Do komunikacji zadań w języku *Ada* wybrano specjalną formę przekazywania wiadomości opartą na tzw. pozycjach zadań (ang. *task entries*), które umożliwiają blokowanie aż do momentu, gdy inne zadanie aktywuje daną pozycję. Kod obsługi danej pozycji może posiadać dodatkowo warunki wstępne, które oferują specjalną synchronizację (warunkowe sekcje krytyczne). Ponadto omawiany język dostarcza konstrukcje podobne do monitorów z rozszerzoną semantyką, możliwość multipleksacji wyboru pozycji (operacja typu *select*), wykrywanie zakleszczeń na etapie kompilacji, kolejkowanie pozycji o tej samej sygnaturze i wiele innych, zaawansowanych konstrukcji współbieżnych. Język *Ada* nie stawia zatem na prostotę i minimalizm jak *Go*, ale na bogactwo mechanizmów dostępnych dla programisty.

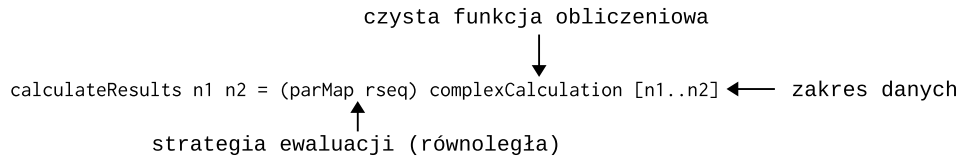
Ada ze względu na wysoką jakość i bezpieczeństwo kodu jest chętnie wykorzystywana w zastosowaniach typu *mission critical* takich jak awionika, bankowość, militaria czy astronautyka.

3.4.2. Języki funkcyjne

Kolejną grupę języków dobrze wspierających współbieżność stanowią języki funkcyjne. Programowanie funkcyjne jest bardzo wiekowym, jak na informatykę, wynalazkiem sięgającym późnych lat pięćdziesiątych i stworzenia języka *Lisp*. Mogłoby się zatem wydawać, że ten paradygmat ma bardzo niewiele wspólnego z nowoczesnym programowanie współbieżnym i równoległym, jednak jak niedawno na nowo odkryto, elegancja wywiedziona wprost z matematycznej koncepcji rachunku lambda znakomicie ułatwia i wspomaga współczesną inżynierię oprogramowania. Ostatnimi czasy bardzo często wręcz przewiduje się „rewolucję funkcyjną", która na zawszę odmieni tendencję do pisania imperatywnego kodu [25]. Warto zatem zastanowić się, co powoduje, że języki funkcyjne tak dobrze dostosowane są do współbieżności.

Pierwszą istotną cechą języków funkcyjnych jest koncentracja na eliminacji jakiegokolwiek stanu współdzielonego. Brak współdzielenia danych jest równoznaczny z minimalizacją problemów z synchronizacją dostępu, co stanowi istotne ułatwienie w stosunku do języków z dzieleniem danych. Programowanie funkcyjne opiera się zatem na niezmiennych strukturach danych transformowanych i przekazywanych pomiędzy poszczególnymi funkcjami programu. Takie podejście upraszcza przepływ danych, często jednak kosztem wydajności związanej z wielokrotną alokacją pamięci. Jest to jednak koszt, który może zostać zminimalizowany np. dzięki memoizacji.

Kolejną powszechną praktyką w programowaniu funkcyjnym jest ścisła separacja obliczeń i efektów ubocznych. Niektóre języki funkcyjne stawiające na wysoką niezawodność i bezpieczeństwo kodu (takie jak np. *Haskell*) wręcz na poziomie kompilacji (poprzez silny system typów algebraicznych) zabraniają wykonywania operacji wejścia/wyjścia w obliczeniach. Uzyskane dzięki temu funkcje czyste zachowują własność przezroczystości referencyjnej (ang. *referential transparency*), co oznacza, że konkretne wywołanie może zostać zastąpione samym wynikiem bez zmiany zachowania programu. Wykonanie takiej funkcji może być zatem zrównoleglone bez obaw związanych z synchronizacją [54, 55, 72].



Rysunek 3.11: Bardzo proste zrównoleglenie wykonania obliczeń `complexCalculation` w języku *Haskell* dla argumentów z przedziału `[n1..n2]` dzięki zastosowaniu strategii ewaluacji `parMap rseq` zamiast standardowej `map`. Przyspieszenie jest możliwe dzięki czystości funkcji `complexCalculation`.

Oczywiście praktycznie każdy złożony system współbieżny wymaga współużytkownia pewnej części danych (np. liczników czy uchwytów do zasobów zdalnych) i jest to także możliwe w programowaniu funkcyjnym. Co ważne, ze względu na akademickie pochodzenie tej klasy języków, przyjęte w nich modele współdzielenia są bardzo często głęboko przemyślane i mają podstawy teoretyczne w postaci matematyki i informatyki teoretycznej, co powoduje, że są wysoce wyrafinowane.

W językach funkcyjnych najczęściej wykorzystuje się opisaną wcześniej (zob. podrozdział 3.3.2) pamięć transakcyjną (której wzorcowa implementacja znalazła się w języku *Haskell*, dzięki funkcjom czystym, które są zarazem idempotnentne) albo przekazywanie wiadomości w modelu CSP lub aktorów (zrealizowane w bardzo interesujący sposób w języku *Erlang*).

Wyżej opisane cechy tj.: skoncentrowanie na czystej transformacji danych, ścisła separacja efektów ubocznych, a także minimalizacja stanu współdzielonego powodują, że programowanie funkcyjne w naturalny sposób sprzyja programowaniu współbieżnemu.

3.4.3. Języki eksperymentalne

Ostatnim zagadnieniem, które warto zasygnalizować w niniejszym podrozdziale jest to, że wciąż powstają nowe języki programowania, także takie, które prezentują całkowicie innowacyjne, niespotykane wcześniej podejście do tematu współbieżności. Trudno to zjawisko opisać w sposób wyczerpujący i kompletny, gdyż ze względu na eksperymentalny charakter tych technologii nie są one jeszcze w zastosowaniu przemysłowym i z tego powodu nie doczekały się odpowiednich opracowań. W związku z tym w celu zarysowania aktualnych trendów poniżej opisano dwie najbardziej obiecujące technologie.

Rust [83] jest językiem łączącym wiele paradygmatów, tworzonym przez fundację *Mozilla*. Innowacyjny charakter tego przedsięwzięcia polega głównie na próbie stworzenia języka całkowicie uniwersalnego, łączącego jak najwięcej dobrych praktyk z innych języków programowania. Najważniejsze cechy języka *Rust* to m.in.:
— statyczne, algebraiczne typy danych zaczerpnięte z programowania funkcyjnego,
— brak wskaźników zerowych, brak arytmetyki wskaźników, statyczny analizator poprawności użycia wskaźników
— współbieżność zorientowana zadaniowo,
— brak pamięci współdzielonej bezpośrednio na rzecz modelu przekazywania wiadomości,
— domyślnie niezmienne struktury danych,
— opcjonalny odśmiecacz pozwalający na minimalizację narzutu wydajnościowego,
— zgodność na poziomie kompilacji z *C*/*C*++.

Żadna z przedstawionych koncepcji nie jest zatem nowa, jednak ich połączenie jest całkowicie unikalne. To co jest najważniejsze w kontekście współbieżności, to model zarządzania pamięcią. Przede wszystkim nie istnieje możliwość wykorzystywania zmiennych globalnych (jak choćby w *Go*) — przekazywanie wiadomości jest zatem jedynym sposobem komunikacji. Aby umożliwić tworzenie efektywnych programów w języku *Rust*, udostępniono aż cztery rodzaje wskaźników:
— wskaźniki unikalne, z wyłącznie jednym, transferowalnym właścicielem, niewymagające odśmiecacza,
— wskaźniki zarządzane ze standardowym mechanizmem *ARC* (ang. *automatic reference counting*) obsługiwane przez odśmiecacz,
— wskaźniki zapożyczone, które mogą wskazywać na dowolny blok pamięci, ale są zabezpieczane przed błędami w sposób statyczny, na poziomie kompilacji; są one pozbawione narzutu odśmiecacza,
— wskaźniki do kodu niebezpiecznego, użyteczne przy współdziałaniu z kodem napisanym w *C*/*C*++.

Taki model zarządzania pamięcią ma umożliwić łatwe tworzenie bardzo bezpiecznych programów wielowątkowych. Można się jednak obawiać, że duża liczba konstrukcji semantycznych tego języka spowoduje, że stanie się on zbyt trudny do opanowania przez typowych programistów. Istnieją plany [83] wydania stabilnej wersji *Rust* już pod koniec 2013 roku, a więc niebawem wszelkie nadzieje i obawy zostaną zweryfikowane.

Dużo bardziej radykalne podejście do problemu współbieżności proponują twórcy języka *Flow* [49], który określany jest jako język bezpieczny współbieżnie, niejawnie zrównoleglający. Zespół pracujący nad *Flow* uważa, że tworzenie poprawnych programów współbieżnych przy obecnej technologii nie jest możliwe, gdyż programiści nie są w stanie zapanować nad właściwą organizacją zadań i ich synchronizacją. Z tego powodu zrównoleglanie ma być wykonywane wyłącznie przez kompilator *Flow*, który sam będzie decydował, które fragmenty i w jaki sposób wykonać równolegle.

Zgodnie ze studium wykonanym przez zespół pracujący nad *Flow* [48], niejawne zrównoleglenie standardowego, imperatywnego programu jest zadaniem niemożliwym do obliczenia, ponieważ graf zależności odwołań do pamięci nie może być określony bez wstępnego uruchomienia takiego programu. Z tego powodu twórcy *Flow* próbują stworzyć język będący hybrydą języka funkcyjnego i imperatywnego tak, aby był zarazem łatwy do wykorzystania oraz możliwy do systematycznego przeanalizowania w celu odnalezienia fragmentów możliwych do zrównoleglenia. W obecnym stadium rozwoju trudno powiedzieć, czy twórcy tego języka mają szansę na osiągnięcie założonego celu. Można mieć obawy, że analiza wykonywana na poziomie kompilacji będzie zbyt pobłażliwa i otrzymywane rezultaty będą istotnie gorsze od standardowych mechanizmów, popularnych w tradycyjnym programowaniu funkcyjnym i imperatywnym. Z drugiej strony, jeśli cel zostanie osiągnięty, to zostaną spełnione wieloletnie marzenia o łatwej i poprawnej współbieżności.

Rust i *Flow* nie wyczerpują oczywiście repertuaru eksperymentalnych języków wspierających współbieżność. Wciąż trwają pracę nad nowymi rozwiązaniami — bardziej wyważonymi tak jak wspomniany wcześniej *Rust* i zmierzającymi w zupełnie nieznanym kierunku tak jak *Flow*. Trudno jednoznacznie określić, czy którekolwiek podejście okaże się wystarczająco skuteczne, niemniej jednak warto obserwować rozwój współczesnych języków oprogramowania, bo być może właśnie w nich tkwi rozwiązanie problemu niezawodnej i bezpiecznej współbieżności.

3.5. Podsumowanie

W niniejszym rozdziale przedstawiono nowoczesne metody tworzenia oprogramowania współbieżnego o różnym stopniu złożoności — od prostych do stosowania dobrych praktyk poprzez zaawansowane mechanizmy aż do wyrafinowanych języków programowania. Łatwo zauważyć, że najmniejsze szanse na sukces wiążą się z samym przestrzeganiem dobrych praktyk, gdyż są one bardzo relatywne, niejasne i niekoniecznie możliwe do zastosowania w każdym projekcie. Dużo większe możliwości daje stosowanie nowoczesnych mechanizmów, które rozwijają się bardzo dynamicznie i pozwalają zastosować powszechnie znane środowiska programistyczne do całkiem nowych wyzwań. Najciekawszą kwestią są jednak języki programowania, tworzone bądź dostosowywane do współbieżności. Poza niechlubnymi wyjątkami (takimi jak np. *Python* [59]), współczesne języki intensywnie rozwijane są

w kierunku programowania współbieżnego i należy spodziewać się, że ta tendencja się utrzyma i w niedługim czasie powstaną bardzo ciekawe rozwiązania.

Warto jednak pamiętać, że w porównaniu do stosowania dobrych praktyk i odpowiednich narzędzi, wykorzystywanie nowoczesnych technik, mechanizmów czy języków programowania wymaga nieco więcej zmian w samym procesie wytwórczym — potrzebne są szkolenia, zmiany środowisk rozwojowych i metod pracy. Wszystko to pociąga za sobą inwestycje, co przy mentalności wielkich korporacji i starszego pokolenia programistów powoduje, że najwięksi wytwórcy oprogramowania bardzo powolnie wdrażają najnowsze osiągnięcia w tej dziedzinie [69]. Nietrudno zatem znaleźć powstające projekty, w których synchronizacja i komunikacja bezzasadnie wciąż odbywa się za pomocą standardowych mechanizmów takich jak semafory i pamięć współdzielona. Pomimo tego dynamika rynku oprogramowania współbieżnego z całą pewnością będzie stopniowo wymuszać zmiany nawet w najbardziej konserwatywnych środowiskach i w związku z tym należy spodziewać się dalszego rozwoju tej dziedziny.

Tematem nieomówiony w niniejszym rozdziale pozostają narzędzia wspierające programowanie współbieżne. Wszystkie przytoczone opracowania dotyczące błędów we współbieżności [28, 69, 122, 124] zgodnie zauważają, że w tej kwestii technologia najbardziej pozostaje w tyle. Mogłoby się wydawać, że nie ma to znaczenia w momencie, gdy dysponujemy skutecznymi mechanizmami i ekspresyjnymi językami. Należy jednak pamiętać, że bardzo wiele programów wciąż jest pielęgnowanych i pisanych w technologiach tradycyjnych, dla których nie istnieje możliwość zastosowania najnowszych technik i platform — te niszę doskonale mogłyby wypełnić właściwe narzędzia. Z tego powodu, jak już wcześniej zauważono, potrzebny jest dynamiczny rozwój narzędzi dla współbieżności i właśnie to zagadnienie podjęto w kolejnym rozdziale.

4. Narzędzia wspierające programowanie współbieżne

Tematyka narzędzi wspierających programowanie współbieżne nie została opisana w rozdziale 3, ponieważ jak już wielokrotnie zauważono, nie rozwijają się one na równi z mechanizmami, technikami i językami programowania [69, 122, 124]. Warto zatem dogłębnie zastanowić się, jaka jest geneza tego problemu i jak można obecny stan tej dziedziny poprawić.

Może wydawać się, że rozwój narzędzi wspierających programowanie współbieżne nie jest potrzebny ze względu na bogactwo dostępnych bardziej zaawansowanych mechanizmów i platform. Należy jednak pamiętać, że współczesny przemysł informatyczny w dalszym ciągu opiera się na kodzie napisanym i rozwijanym w tradycyjnych technologiach obarczonych wieloma problemami, pokazanymi w rozdziałach 1 oraz 2. Nie istnieje przy tym możliwość prostej i szybkiej zmiany tej sytuacji — proces wprowadzania nowoczesnych technik do fundamentalnych projektów będzie z całą pewnością bardzo powolny. Kluczową rolę w ulepszaniu tych istotnych rozwiązań mogą odegrać właśnie odpowiednie narzędzia, ponieważ ich użycie jest mniej kosztowne — łatwiejsze i nieinwazyjne dla całego ekosystemu. Z tego powodu badania w kierunku stworzenia lepszych narzędzi wydają się być koniecznością.

W niniejszym rozdziale skoncentrowano się zatem na opisie istniejących narzędzi, przykładając szczególną wagę do analizy użyteczności ich wykorzystania w rzeczywistych programach. Prezentowane narzędzia pogrupowano według ich przeznaczenia. Wyodrębnione grupy to narzędzia do:
— automatycznej formalnej lub półformalnej weryfikacji modelu,
— transformacji programów współbieżnych,
— statycznej analizy kodu,
— dynamicznej analizy kodu i wykonania,
— wzmacniania styku programu z *API*,
— próbnego uruchamiania,
— testowania.
Rozdział zakończono podsumowaniem, w którym podjęto temat dalszych kierunków rozwoju narzędzi wspierających programowanie współbieżne.

4.1. Weryfikacja modelu

Praktycznie każdy program komputerowy można zapisać w pewnym sformalizowanym języku matematycznym, który pozwala dowodzić pewnych własności aplikacji. Takie działania są powszechne w dziedzinach, w których oprogramowanie ma kluczowe znaczenie dla bezpieczeństwa jak np. awionika, energetyka czy astronautyka. Dla programowania współbieżnego powstało również wiele teorii, które pozwalają na formalną weryfikację poprawności. Do najpopularniejszych metod

należą wspomniane w podrozdziale 3.3.2 algebry dla CSP i modelu aktorów, a także sieci Petriego [85].

Metody weryfikacji modelu nie znalazły jednak powszechnego zastosowania w inżynierii oprogramowania ze względu na trudność użycia pośród typowych programistów. Podjęto więc próby, również w programowaniu współbieżnym, stworzenia narzędzi, które w sposób automatyczny, na podstawie kodu źródłowego byłyby w stanie przeprowadzić formalne dowody odpowiednich własności. Opis najbardziej znanych rozwiązań tego typu zaprezentowano poniżej.

Z przeglądu wyłączono narzędzia wymagające tworzenia formalnych specyfikacji w wyspecjalizowanych językach (jak np. *Spin* [10]) ze względu na małe praktyczne zastosowanie tego typu rozwiązań w procesie wytwarzania typowego oprogramowania.

4.1.1. Zing

Zing [4, 79] to narzędzie badawcze stworzone przez firmę *Microsoft* służące do tworzenia i weryfikowania modeli programów współbieżnych poprzez eksplorację stanów. Jest to produkt w fazie eksperymentalnej, który aktywnie wykorzystywany jest przy tworzeniu sterowników do systemów operacyjnych z rodziny *Windows*. *Zing* składa się z czterech elementów:
— języka modelowania do tworzenia modeli systemów współbieżnych,
— kompilatora ww. języka,
— eksploratora stanów stworzonego modelu,
— generatora modeli na podstawie programów napisanych w C# oraz C.

Standardowy proces działania narzędzia *Zing* składa się z kilku kroków. W pierwszym program tłumaczony jest (automatycznie lub ręcznie) na język modelowania. W języku tym znajdują się elementy znane ze typowych języków programowania takie jak: instrukcje warunkowe, pętle, funkcje, wyjątki, typy prymitywne i referencyjne oraz wątki. Tak zapisany program kompilowany jest do pliku modelu, który może być uruchamiany, by wygenerować przejścia pomiędzy stanami. Stan w narzędziu *Zing* składa się z trzech komponentów: stosów, składnic globalnych oraz sterty, które znane są ze standardowego modelu uruchomieniowego. Plik modelu uruchamiany jest w eksploratorze stanów, który systematycznie odkrywa osiągalne stany wgłąb. Dokładny algorytm eksploracji przestrzeni stanów oraz ich interpretacji zależy od wybranych ustawień i dostępnych rozszerzeń.

W obecnej chwili ciężko jest rzetelnie ocenić przydatność omawianego narzędzia ze względu na to, że wykorzystywane jest ono w zasadzie wyłącznie wewnątrz firmy *Microsoft* i według licencji nie istnieje możliwość komercyjnego wdrożenia [79]. Ponadto nie jest dostępny kod źródłowy tego rozwiązania, co znacząco utrudnia jego rozszerzanie. W tym miejscu warto jednak zauważyć, że wciąż jest ono aktywnie rozwijane przez komórkę badawczą firmy *Microsoft*. To co pozytywnie odróżnia to narzędzie od innych weryfikatorów modeli jest fakt, że obsługuje nowoczesne technologie (platformę *.NET*) i według badania [4], zachowuje wysoką wydajność ze względu na szereg optymalizacji. Rozwój narzędzia *Zing* należy uważnie śledzić, gdyż istnieje szansa, że zostanie wdrożone do środowiska *Visual Studio*, co sprawi, że natychmiast stanie się popularne wśród programistów.

4.1.2. VeriSoft

Innym, nieco starszym od *Zing*, narzędziem do weryfikacji modelu jest *VeriSoft* [9]. *VeriSoft* podobnie jak narzędzie firmy *Microsoft* oferuje systematyczną eksplorację przestrzeni stanów, która ma odkrywać sytuacje błędne. Technika działania opisywanego rozwiązania jest jednak zdecydowanie inna. *VeriSoft* obserwuje i rejestruje komunikację pomiędzy procesami poprzez przechwytywanie wywołań systemowych takich jak np. operacje semaforowe czy komunikacja międzyprocesowa — pozostałe, wewnętrzne operacje procesów są nieobserwowalne. Wykonane przez procesy działania definiują stany i tranzycje. Pojedyncze wykonanie (tzn. ścieżka w grafie stanów) tworzy scenariusz, który potencjalnie zawiera błędy. W scenariuszach *VeriSoft* poszukuje: zakleszczeń, naruszeń asercji, zaników komunikacji w zadanym okresie oraz zjawisk typu *livelock*. Algorytm przeszukiwania stanów w *VeriSoft* jest odpowiednio zmodyfikowany tak, aby sam w sobie był bezstanowy — gwarantuje to niskie zużycie pamięci. Ze względu na sposób działania, *VeriSoft* przypomina dynamiczny analizator wykonania, ale według twórców rozwiązanie to jest bardziej sformalizowane [37], dlatego zasługuje na miano weryfikatora modelu.

Obrany model nadzorowania powoduje, że *VeriSoft* umożliwia sprawdzanie programów napisanych w dowolnych językach programowania, jednak jest trudny do zastosowania dla aplikacji wielowątkowych z pamięcią współdzieloną. Istotną zaletą opisywanego narzędzia jest bogactwo dodatków — *VeriSoft* wyposażony jest w symulator, debugger, narzędzie do wizualizacji i wiele innych. Ze względu na działanie w trakcie wykonania, *VeriSoft* wymaga wielokrotnych, powtórnych uruchomień. Dodatkowo nie sprawdza on wszystkich stanów (co byłoby niemożliwe czasowo), a jedynie pewną określoną „głębokość". Pomimo tego używanie *VeriSoft* należy uznać za wygodne ze względu na brak konieczności modelowania w języku pośrednim.

VeriSoft był m.in. z powodzeniem wykorzystywany w firmie *Lucent* do weryfikacji kodu obsługującego telekomunikacyjne stacje bazowe. Firma ta, w kooperacji z *Bell Laboratories*, rozwijała projekt do roku 2006. Aktualnie nieznane są dalsze losy projektu, co budzi uzasadnione obawy, co do możliwości dalszego wykorzystania tego narzędzia.

4.1.3. Java PathFinder

Java PathFinder (*JPF*) [44, 89] to system do weryfikacji kodu bajtowego *Javy*, stworzony przez *NASA Ames Research Center* i udostępniony publicznie w 2005 roku. Do zadań *JPF* należy m.in.:
— znajdowanie i opisywanie błędów (w tym zakleszczeń i wyścigów),
— akwizycja danych uruchomieniowych, w tym metryk pokrycia,
— wnioskowanie na temat odpowiednich testów i tworzenie ich.

Kluczowym elementem *JPF* jest zmodyfikowana maszyna wirtualna *Javy* napisany w tymże języku. W celu zweryfikowania programu, maszynie wirtualnej *JPF* dostarcza się skompilowane oprogramowanie, a także zbiór własności do zweryfikowania. Po zakończeniu procedury weryfikacji tworzony jest raport, który zawiera informacje na temat spełnialności przedstawionych własności, a także dodatkowe artefakty, takie jak np. przypadki testowe.

Działanie *JPF* opiera się na tzw. punktach wyboru — miejscach w programie, w których wykonanie może potoczyć się inaczej niż w podstawowym uruchomieniu.

Omawiane narzędzie sprawdza systematycznie wszystkie ścieżki wynikające z istnienia punktów wyboru. W sprawdzeniu używane są zwykle różnorodne sekwencje planisty oraz wartości losowe, ale może to być konfigurowane przez użytkownika. Tak jak we wszystkich narzędziach do weryfikacji modelu, największym problemem w *JPF* jest kombinatoryczna eksplozja stanów. Obranym w narzędziu sposobem na obniżenie złożoności jest analiza maszyny stanów prowadząca do znajdowania i usuwania stanów podobnych. Innym pomysłem jest redukcja częściowego uporządkowania (ang. *partial order reduction*), która łączy operacje lokalne dla wątku w pojedynczy, atomowy blok zmniejszając tym samym przestrzeń przeplotów wątków.

Opisywane narzędzie jest powszechnie znanym i wykorzystywanym w środowisku programistów *Javy* narzędziem do analizy modelu. Sukces *JPF* spowodował nawet, że zostało ono przeniesione na platformę *.NET* w postaci projektu *MoonWalker* [117]. Niestety, nie istnieje możliwość wykorzystania *JPF* do analizy programów niekompilowalnych do kodu bajtowego *JVM*. Co więcej, *JPF* nie może być używany do analizy programów wykonujących wywołania natywne *JNI*. Kolejną wadą tego weryfikatora jest bardzo trudna konfiguracja, która niekiedy wymaga nawet programowania. Z drugiej strony, modularna konstrukcja *JPF* umożliwia jego swobodne rozszerzanie i modyfikowanie. Co istotne, narzędzie to jest nadal aktywnie rozwijane i nie traci popularności wśród programistów języka *Java*. Można zatem powiedzieć, że środowisko *JVM* doczekało się użytecznego i dobrze wspieranego weryfikatora modelu.

4.1.4. Komentarz

Przedstawione powyżej narzędzia do automatycznej analizy i weryfikacji modelu oczywiście nie wyczerpują listy wszystkich rozwiązań z dziedziny, które kiedykolwiek powstały. Niestety, poza wspomnianymi, mało który projekt został wykorzystany w środowisku produkcyjnym — poza samym procesem badawczym. Pojawiające się innowacyjne pomysły, takie jak choćby bezpieczny system typów gwarantujący brak zakleszczeń i wyścigów [13], okazywały się nieprzystające do współczesnej, dynamicznie rozwijającej się inżynierii oprogramowania. Tworzenie i weryfikowanie modeli wciąż, pomimo procesów automatyzujących, wydaje się być zbyt trudne i czasochłonne do efektywnego wykorzystania. Można zatem spodziewać się, że metody weryfikacji modelu będą coraz bardziej zmierzać w kierunku nieformalnych metod heurystycznych reprezentowanych przez opisaną w podrozdziale 4.3 statyczną analizę kodu.

4.2. Transformacja

Głównym problemem związanym z formalną analizą modelu jest wspomniana wcześniej eksplozja stanów. Na drodze do rozwiązania tego problemu zaproponowano, aby programy współbieżne przekształcać do postaci programów sekwencyjnych, które są wielokrotnie prostsze do analizy. Takie podejście pozwala na dokonywanie analizy w czasie wielomianowym zamiast czasu wykładniczego i ponadwykładniczego. Jak nietrudno się domyślić, taka synteza jest trudna do wykonania i implikuje pewne uproszczenia i kompromisy. Z tego powodu metoda nie znalazła powszechnego zastosowania. Najbardziej znaną, akademicką implementację narzędzia do transformacji opisano poniżej.

4.2.1. KISS

KISS (ang. *Keep It Simple and Sequential!*) [100] to weryfikator właściwości dla wielowątkowych programów napisanych w *C*, oparty na narzędziu *SLAM* dla programów sekwencyjnych. Zasada działania *KISS* jest bardzo prosta — narzędzie to przekształca graf przepływu programu współbieżnego do odpowiadającego grafu programu sekwencyjnego.

Najważniejszym elementem opisywanego narzędzia jest algorytm transformacji, który pozwala wyrazić podzbiór programu współbieżnego jako program sekwencyjny. Transformacja w *KISS* w uproszczeniu polega na wykonaniu współbieżnego programu w sposób sekwencyjny przez niedeterministycznego planistę. Zmieniony program sekwencyjny posiada zatem fragmenty wyodrębnione jako wątki programu współbieżnego, które mogą być uruchamiane w niedeterministycznej kolejności. W ten sposób wykonanie przypomina uruchamianie wielowątkowych programów na komputerze jednoprocesorowym z tym, że rola planisty jest przeniesiona z jądra do przestrzeni użytkownika. Główny problem w konstrukcji takiego środowiska uruchomieniowego polega na właściwym zarządzaniu kontekstem i jego przełączaniem.

Stosując wyżej opisany algorytm otrzymuje się podzbiór wszystkich możliwych uruchomień. *KISS* oferuje parametry pozwalające na definiowanie liczby przełączeń kontekstu, które wpływają oczywiście na złożoność obliczeniową całego procesu testowania.

KISS to ciekawe narzędzie, które pozwala na stosowanie sprawdzonych rozwiązań dedykowanych programom jednowątkowym do programów współbieżnych przy zachowaniu obniżonej skuteczności i przy zwiększonej złożoności, zależnej od liczby wygenerowanych, różnych programów sekwencyjnych. Podejście to nie zyskało jednak odpowiedniego uznania i nie zostało przeniesione z fazy badawczej do przemysłu informatycznego. Trudno jednoznacznie wskazać przyczyny niepowodzenia tego przedsięwzięcia — być może metody transformacji wydają się zbyt mało dokładne lub zbyt powolne dla bardzo złożonych programów, aby wydawały się atrakcyjnym środkiem weryfikacji poprawności oprogramowania.

4.3. Statyczna analiza kodu

Statyczna analiza kodu to jedna z najpopularniejszych technik wspomagania wytwarzania oprogramowania zarówno sekwencyjnego jak i współbieżnego, która zyskała dużą sympatię programistów [15]. Narzędzia tej klasy, na podstawie analizy syntaktyczno–semantycznej kodu, udzielają programistom podpowiedzi na temat problemów, które mogą wystąpić podczas wykonania kodu, a także bardzo często sugerują jak poprawić bądź ulepszyć sekcje tego wymagające. Techniki analizy statycznej zyskały powodzenie ze względu na łatwość wykorzystania w środowisku produkcyjnym. Powszechnie uważa się je, ze względu na ich heurystyczną naturę, za łatwiejsze w implementacji od metod weryfikacji modelu, ale jednocześnie także za mniej skuteczne [4] — przeszukiwanie w narzędziach statycznej analizy kodu jest zwykle płytkie, ale wydajne, więc może być wykonywane np. przez zintegrowane środowiska programistyczne.

Tak jak wspomniano, także dla programowania współbieżnego zaproponowano wiele narzędzi do statycznej analizy. Najciekawsze rozwiązania tego typu zaprezentowano poniżej.

4.3.1. CheckThread

CheckThread [16] to narzędzie do statycznej analizy dla języka *Java*, które opiera się na adnotacjach dodawanych przez programistę do kodu. Adnotacje służą do zawiązywania pewnego, silnego kontraktu między twórcą klasy, a jej użytkownikiem. Adnotacje udostępniane przez *CheckThread* to:

— **`@ThreadSafe`** — metoda oznaczona w ten sposób może być swobodnie wywoływana przez wiele wątków, gdyż implementujący zadbał o jej bezpieczeństwo w środowisku wielowątkowym,
— **`@NotThreadSafe`** — metoda nie może być bezpiecznie wywoływana przez wiele wątków, wymagana jest dodatkowa synchronizacja,
— **`@ThreadConfined`** — metoda powinna być wołana tylko przez specjalnie określone wątki, np. wątki *UI*.

W celu zagwarantowania spełnienia kontraktu zawartego przez dodanie adnotacji, *CheckThread* sprawdza następujące własności:

— Dla metod oznaczonych `@ThreadSafe`:
  — nie wykonują operacji niesynchronizowanego odczytu i zapisu na danych współdzielonych,
  — nie wywołują metod oznaczonych `@ThreadConfined`,
  — nie wywołują metod typu `@NotThreadSafe` bez odpowiedniej synchronizacji.
— Dla metod oznaczonych `@NotThreadSafe` nie wywołują metod typu `@ThreadConfined`.
— Dla metod oznaczonych `@ThreadConfined`:
  — nie wywołują metod typu `@NotThreadSafe` bez odpowiedniej synchronizacji.
  — nie wywołują innych metod typu `@ThreadConfined` o różnych wątkach dozwolonych.

Poprzez proste wzmocnienie systemu typów, *CheckThread* pozwala na tworzenie kodu wielowątkowego, wolnego od większości błędów wyścigów na danych. Oczywistą wadą tego rozwiązania jest to, że wymaga od programisty pamiętania o odpowiednich oznaczeniach metod, jednak czynność ta nie jest zbyt obciążająca i sama w sobie nie powinna zniechęcać do jej stosowania. Istotną zaletą *CheckThread* jest natomiast dobra integracja ze wszystkimi liczącymi się zintegrowanymi środowiskami programistycznymi takimi jak *Eclipse*, *IntelliJ IDEA* oraz *NetBeans*. Omawiane narzędzie jest bardzo dobrym przykładem, jak w prosty i niekosztowny sposób można zwiększyć bezpieczeństwo programów współbieżnych.

4.3.2. ConSeq

ConSeq [127] to ciekawe narzędzie łączące techniki statycznej i dynamicznej analizy programów napisanych w *C* i *C++*. Unikalność *ConSeq* polega na odwróceniu standardowego sposobu analizy: najpierw omawiane narzędzie identyfikuje potencjalne błędy metodami analizy statycznej, następnie podobnymi sposobami wyszukuje krytyczne dla poprawności odczyty, a na końcu obserwuje pojedyncze, poprawne wykonanie programu, aby wykryć przeploty, które mogą doprowadzić do ujawnienia się błędu. Dzięki wstępnej, statycznej analizie, *ConSeq* zawęża przestrzeń różnych przeplotów, które potem mogą być sprawdzone systematycznie. Ten sposób działania oparto na założeniu, że pomimo istotnych różnic w genezie występowania błędów we współbieżności i błędów sekwencyjnych, ich skutki są podobne.

Pierwsza faza identyfikacji potencjalnych błędów rozważa pięć różnych ich typów: naruszenie asercji, nieskończona pętla, niewłaściwe wywołania funkcji wyjścia, niewłaściwe wywołania funkcji wiadomości błędów oraz niebezpieczne odczyty zmiennych globalnych. Identyfikacja odbywa się poprzez prostą analizę kodu binarnego — poprzez wyszukiwanie wywołań odpowiednich funkcji lub wykonania konkretnych instrukcji.

Kolejna faza analizuje krytyczne odczyty, które mogą mieć wpływ na ujawnianie się błędów wspomnianych powyżej. W celu efektywnego wykonania tego etapu *ConSeq* wykorzystuje metody wyznaczania statycznych dekompozycji programu (ang. *static slicing*) oraz ogranicza się tylko do krótkich sekcji, zgodnie z założeniem, że instrukcje powodujące błąd znajdują się relatywnie blisko miejsca, w którym błąd się ujawni. Faza ta stosuje zatem wiele uproszczeń w celu zachowania balansu pomiędzy wydajnością a skutecznością.

Ostatni etap, który polega na wykrywaniu podejrzanych przeplotów opiera się na zbieraniu informacji na temat poprawnego wykonania programu. Na podstawie tych danych budowany jest graf kontroli przepływu sterowania i operacji, który służy podejmowaniu decyzji, jaka sekwencja przeplotów może doprowadzić do niewłaściwego odczytu krytycznego. W tej fazie stosuje się także uproszczenia zawężające przestrzeń wszystkich przeplotów, aby zwiększyć wydajność kosztem skuteczności.

Interesującym dodatkiem do narzędzia *ConSeq* jest moduł testowania podejrzanych przeplotów, który próbuje wykonać sekwencję oznaczoną jaką błędną, aby sprawdzić czy rzeczywiście doprowadza ona do błędu. W celu wymuszenia konkretnego przeplotu do programu, wstrzykiwane są opóźnienia warunkowe, które zwiększają prawdopodobieństwo ujawnienia się błędu.

ConSeq to bardzo złożone narzędzie oparte na wieloetapowej analizie, która wymusza pewne uproszczenia. Z tego powodu można mieć istotne obawy co do skuteczności rozwiązania, szczególnie ze względu na mały zakres błędów analizowanych w pierwszym etapie. Niemniej jednak koncepcja zastosowana w tym narzędziu wydaje się być interesująca i innowacyjna i być może mogłaby zostać wykorzystana w innych, podobnych narzędziach. Niestety, póki co twórcy *ConSeq* nie zdecydowali się na opublikowanie swojego rozwiązania, a więc nie ma sposobu, aby zweryfikować jego skuteczność w środowisku produkcyjnym.

4.3.3. Komentarz

Poza opisanymi wyżej rozwiązaniami powstało także wiele innych koncepcji akademickich, które jednak nigdy nie zostały wdrożone do środowiska produkcyjnego. Warto tutaj wspomnieć o projektach takich jak: *RacerX* [26], który używa intensywnej analizy interproceduralnej do odnajdowania informacji na temat danych współdzielonych i odpowiadających im zamków; *Autolocker* [73], który automatycznie zamienia pesymistyczne sekcje atomowe na ziarniste blokowanie oraz *MUVI* [68], który jako nieliczny opiera się na analizie wielu zmiennych. Te, jak i wiele innych projektów badawczych, z nie zawsze znanych powodów nie znalazło zastosowania w przemyśle informatycznym.

Warto jednak wspomnieć, że w niniejszej sekcji omówiono wyłącznie rozwiązania dedykowane współbieżności. Wiele standardowych analizatorów statycznych, takich jak choćby *Coverity* [21] czy *PC–Lint* [33] także udziela wskazówek dotyczących budowy właściwego kodu współbieżnego. Można zatem powiedzieć, że dziedzina

statycznej analizy programów współbieżnych rozwija się i już w obecnej chwili ma się całkiem dobrze.

4.4. Dynamiczna analiza kodu i wykonania

Narzędzia do dynamicznej analizy kodu i wykonania programu, zyskały bardzo dużą popularność wśród programistów. Rozwiązania te dostarczają o wiele więcej i bardziej wnikliwych informacji niż choćby aplikacje do statycznej analizy kodu. Ich istotną wadą jest natomiast to, że badają tylko faktycznie wykonywane ścieżki programu, a te, które są wywoływane stosunkowo rzadko, będą gorzej sprawdzone. Pomimo tego łatwość implementacji i użycia, a także bogactwo danych dostarczanych przez omawiany typ narzędzi spowodowały, że w wielu środowiskach powstały wysokiej jakości rozwiązania, które są powszechnie stosowane w środowiskach produkcyjnych.

Także dla programowania współbieżnego najliczniejszą grupę wartościowych narzędzi stanowią te dedykowane dynamicznej analizie wykonania. Poniżej skoncentrowano się zatem na opisie rozwiązań popularnych i tych, które mają szansę odnieść sukces w niedługim czasie. Pozostałe (tzn. te o znaczeniu naukowym i historycznym) krótko scharakteryzowano w podrozdziale 4.4.5.

4.4.1. Valgrind

Valgrind [90, 118] to najbardziej znany wśród programistów szkielet do budowania narzędzi do instrumentacji i dynamicznej analizy wykonania. Z technicznego punktu widzenia *Valgrind* jest maszyną wirtualną (z wirtualnym procesorem) wykorzystującą technikę kompilacji *JIT* (ang. *just–in–time*) do uruchamiania programów przekazywanych w postaci binarnej. Żadna z oryginalnych instrukcji programu nie jest wykonywana bezpośrednio na komputerze–gospodarzu, a wyłącznie na wirtualnym procesorze narzędzia *Valgrind*.

Valgrind poza swoim rdzeniem składa się z wielu wtyczek, które służą do różnych rodzajów analizy np. poszukiwania wycieków pamięci, profilowania sterty czy weryfikowania pracy wątków. Przy pierwszym uruchomieniu kod z rdzenia przekazywany jest do wybranej wtyczki tak, aby mógł zostać wzbogacony o odpowiednie funkcje służące do instrumentacji wykonania. Dzięki temu programista uzyskuje żądane informacje na temat analizowanego programu.

Valgrind został wyposażony w dwie wtyczki przydatne przy analizie błędów w programach wielowątkowych: *DRD* oraz *Helgrind*. Rozszerzenia te są do siebie dosyć podobne, jednak zastosowano w nich inne techniki analizy. Krótki opis obu wtyczek zamieszczono poniżej.

DRD

Wtyczka *DRD* służy do wykrywania błędów w wielowątkowych programach napisanych z wykorzystaniem *API POSIX threads*. Typy błędów, które rozpoznaje *DRD* to: niewłaściwe użycie *API POSIX*, głodzenie oraz wyścigi na danych.

Pierwszy z problemów jest zdecydowanie najłatwiejszy do wykrywania i polega na sprawdzaniu czy odpowiednie funkcje są używane zgodnie z założeniami standardu. Przykładowe błędne zachowania, które rozpoznaje *DRD* to: otworzenie zamka, który nie został zamknięty, destrukcja zamkniętego zamka czy ponowna inicjalizacja struktury synchronizującej.

Błąd głodzenia również wykrywany jest w bardzo prosty sposób. Użytkownik przy uruchamianiu badanego programu definiuje najdłuższy możliwy czas oczekiwania wątku na zwolnienie zamka. Jeśli czas ten zostanie przekroczony, sygnalizowany jest błąd.

Najciekawszym elementem *DRD* jest detektor wyścigów na danych. Omawiana wtyczka wykorzystuje zaawansowany algorytm opierający się na relacji poprzedzania (ang. *happens–before algorithm*) [111]. W uproszczeniu, *DRD* sygnalizuje wyścig w przypadku, gdy dostęp do tych samych obszarów pamięci z różnych wątków nie jest uporządkowany za pomocą odpowiedniej synchronizacji. Zaletą algorytmu zastosowanego w tej wtyczce jest to, że nie zgłasza ona *false–positives*.

Helgrind

Wtyczka *Helgrind* podobnie jak *DRD* służy do wykrywania błędów w wielowątkowych programach napisanych z wykorzystaniem *API POSIX threads*. *Helgrind* także rozpoznaje błędne użycie *API* oraz wskazuje wyścigi na danych (za pomocą zmodyfikowanego algorytmu *happens–before*), ale nie pozwala na wykrywanie głodzenia. Unikalną cechą omawianej wtyczki jest natomiast rozpoznawanie błędów związanych z niekonsekwentną kolejnością zajmowania zamków, co może prowadzić do zakleszczeń.

Trudno jednoznacznie wskazać, która z wtyczek jest lepsza. Oba rozwiązania zyskały przychylność programistów, gdyż prezentują bardzo wysoką jakość i duże możliwości. Z tego powodu zaleca się weryfikację programów obiema wtyczkami.

4.4.2. ThreadSanitizer

ThreadSanitizer (*TSan*) [42] to nowe narzędzie do wykrywania wyścigów w programach napisanych w *C*, *C++* i *Go*, tworzone przez firmę *Google*. Rozwiązanie to najprawdopodobniej zostanie dodane do kompilatora *gcc* w wersji 4.8 oraz do tzw. *front–endu clang* w wersji 3.2. Włączenie analizy narzędzia *TSan* odbywa się przy kompilacji — instrumentacja zostaje dodana bezpośrednio do wyjściowego kodu binarnego.

Proces instrumentacji jest bardzo prosty — każdy dostęp do pamięci (poza pewnymi rzadkimi wyjątkami) poprzedzony jest wywołaniem odpowiedniej funkcji narzędzia *TSan*. Rodzaj wywoływanej procedury zależy od rodzaju pamięci — inne funkcje wywoływane są dla pamięci zwykłej, atomowej i znajdującej się pod adresami *vptr*.

Dużo bardziej zaawansowany jest sam algorytm wykrywania wyścigów, który opiera się na maszynie stanów i stanach cieni (ang. *shadow state*). Każdy wyrównany 64 bitowy fragment pamięci jest mapowany do N (2, 4 lub 8) słów cieni (ang. *shadow words*), które łącznie tworzą jeden stan cieni. Pojedyncze słowo cieni składa się z 64 bitów, które przechowują: numer wątku, zegar skalarny, flagę zapisu, rozmiar dostępu (1, 2, 4 lub 8) oraz przesunięcie. Pojedynczy stan cieni zapamiętuje zatem N różnych dostępów do pamięci. Rdzeniem algorytmu jest natomiast maszyna stanów, która uaktualnia stan cieni przy każdym dostępie do pamięci. Jeśli przy aktualizacji wykryty zostanie konflikt zapisu (związany z relacją następowania) jednego ze słów z nowym słowem, zgłoszony zostaje potencjalny wyścig. Wszystkie operacje związane z maszyną stanów wykonywane są w sposób niepodzielny, aby zachować jej spójność.

Algorytm *TSan* wciąż bardzo dynamicznie się zmienia, co utrudnia odnalezienie wyczerpujących informacji na temat samego przebiegu ewolucji maszyny stanów. Wstępne wyniki działania narzędzia wydają się jednak bardzo obiecujące — porównywalne do tego, co oferuje w kwestii wykrywania wyścigów *Valgrind* (zob. podrozdział 4.4.1), jednak ze znacznie mniejszym narzutem wydajnościowym. Należy zatem uważnie przyglądać się rozwojowi tego rozwiązania, gdyż niebawem stanie się ono częścią standardowych narzędzi programistycznych.

4.4.3. lockdep

Lockdep [19, 63] jest wyrafinowanym narzędziem jądra *Linuksa* służącym do wykrywania niepoprawnych sekwencji używania zamków, które mogą prowadzić do zakleszczeń. Podstawowym obiektem, na którym operuje ten walidator jest klasa zamków. Klasę zamków stanowią wszystkie zamki (wszystkie instancje), które logicznie wykorzystywane są w ten sam sposób. Modelowym przykładem klasy są wszystkie zamki skojarzone ze strukturami *inode*. Klasa, w przeciwieństwie do instancji, nigdy nie jest usuwana z jądra — istnieje od uruchomienia do wyłączenia systemu.

Lockdep dostarcza ciągłego dowodu (ang. *rolling proof*) na poprawność wykorzystanych w systemie strategii blokowania. Technicznie odbywa się to poprzez śledzenie stanu i zależności pomiędzy różnymi klasami. Walidator zatem rozważa wzorce użycia, a nie konkretne sekwencje. Przykładem obserwowanej zależności może być np. zamykanie zamka klasy A zawsze po zamku klasy B — wykrycie sekwencji o odwrotnej kolejności wskazuje na prawdopodobieństwo zakleszczenia. W celu odnalezienia błędnego wykorzystania zamków, *lockdep* wykonuje następujące testy:

— w momencie zajmowania nowego zamka klasy Z sprawdzane jest czy którykolwiek z aktualnie zajętych klas zamków była kiedykolwiek zajęta po Z — jeśli tak, sygnalizowana jest możliwość zakleszczenia,
— utrzymywany jest stos aktualnie zajętych zamków — zawsze powinien być zwalniany zamek znajdujący się na szczycie stosu, w przeciwnym wypadku sygnalizowana jest podejrzana sekwencja,
— rygiel pętlowy wykorzystywany w funkcjach obsługi przerwań (zarówno sprzętowych jak i programowych) nie może być nigdy zajmowany, gdy przerwania są włączone — taka sytuacja stwarza możliwość poważnej awarii systemu.

Wykonując wyżej opisane testy walidator osiąga matematyczny dowód poprawności blokowania (braku zakleszczeń) dla pojedynczego, samodzielnego zadania.

Lockdep to niezwykle zaawansowane, wysokiej jakości narzędzie wykorzystujące ciekawą koncepcję klasy zamków. Wadą tego rozwiązania jest wysoki narzut wydajnościowy. Zwyczajowo jednak narzędzia tego nie stosuje się na jądrach w środowisku produkcyjnym, a jedynie na maszynach testowych. *Lockdep* jest narzędziem dedykowany, przystosowanym i powszechnie używanym do weryfikacji jądra *Linuksa*, jednak powstały implementacje i adaptacje tego mechanizmu dla przestrzeni użytkownika [20].

4.4.4. checkedthreads

Checkedthreads [60] to biblioteka do zrównoleglania kodu *C* i *C++* ze zintegrowanym wyszukiwaniem wyścigów. Omawiane narzędzie proponuje intensywne wykorzystanie współbieżności w modelu *fork–join* (znanym np. z *OpenMP*), który może być łatwo zweryfikowany pod względem poprawności. *Checkedthreads* implementuje dwie niezależne metody dynamicznego sprawdzania niezawodności: zamianę kolejności zdarzeń (ang. *event reordering*) oraz monitorowanie dostępów do pamięci. Według autora, użycie obu tych technik łącznie niemalże gwarantuje współbieżną poprawność kodu.

Zamiana kolejności zdarzeń to prosta technika, która polega na uruchomieniu kodu z dwoma różnymi planistami: produkcyjnym oraz losowym, a następnie porównaniu otrzymanych wyników, które oczywiście w poprawnym programie powinny być takie same. Aby właściwie przeprowadzić ten eksperyment, należy dobrać taką serializację, aby zamienić kolejność każdej pary instrukcji, która może być uruchomiona równolegle. Okazuje się, że dla modelu *fork–join* jest to bardzo łatwe, a dla standardowych, surowych wątków obliczeniowo niemożliwe. Za pomocą skierowanych grafów acyklicznych można pokazać, że aby osiągnąć ten efekt dla *fork–join*, wystarczy w drugim przebiegu wszelkie zrównoleglone pętle wykonać od tyłu.

Metoda zamiany kolejności zdarzeń może pomijać pewne błędy (np. niewłaściwe aktualizowanie zmiennych akumulujących). Dodatkowo technika ta wykazuje jedynie ewentualną obecność błędu, nie pokazuje natomiast, gdzie błąd wystąpił. W celu rozwiązania tych problemów dla *checkedthreads* stworzono dynamiczny analizator oparty na narzędziu *Valgrind* (zob. podrozdział 4.4.1), który śledzi odwołania do pamięci. Analizator ten rejestruje wszystkie zapisy do pamięci i wiąże je z wątkiem (właścicielem), który to wykonał. W sytuacji naruszenia własności komórki pamięci, zgłaszany jest błąd poprzez pokazanie śladu stosu.

Checkedthreads oferuje ciekawe podejście do niezawodnej współbieżności oparte na ulepszonym, acz zubożonym *API*. Równoległość zastosowana w omawianym narzędziu uniemożliwia tworzenie wydajnych serwerów wielowątkowych oraz wszelkich rozwiązań, gdzie wątki są autonomiczne. *Checkedthreads* jest zatem bardzo ciekawą, lecz wciąż niedojrzałą alternatywą dla rozwiązań takich jak *OpenMP*. Trudno powiedzieć, czy *checkedthreads* znajdzie powszechne zastosowanie komercyjne, jednak pomysł i ogólna idea przedstawiona w omawianym narzędzi warta jest rozważenia i ewentualnej implementacji w podobnych rozwiązaniach.

4.4.5. Komentarz

Dziedzina narzędzi do dynamicznej analizy wykonania programów współbieżnych wciąż intensywnie się rozwija i co istotne, już w obecnej chwili istnieją rozwiązania wysokiej jakości, które pomagają diagnozować błędy takie jak wyścigi i zakleszczenia. Oprócz opisanych w niniejszym podrozdziale rozwiązań, na przestrzeni lat zaproponowano także wiele innych narzędzi, takich jak m.in.: *RaceTrack* [125], który posiada adaptacyjny model wykrywania wyścigów; *Atomizer* [29] i *AVIO* [70], które wykrywają naruszenia atomowości, a nie tylko wyścigi; *Bugaboo* [71], który łączy dobre cechy analizy dynamicznej i analizy modelu grafowego oraz *SVD* [30], który metodami heurystycznymi wykrywa sekcje, które powinny być atomowe. Widać zatem, że narzędzia do dynamicznej analizy programów współbieżnych to temat gruntownie przebadany, co procentuje ciekawymi i różnorodnymi narzędziami

wspomagającymi proces wytwarzania oprogramowania wielowątkowego i wielozadaniowego.

4.5. Wzmacnianie API

Badania związane z bezpieczeństwem programów współbieżnych wskazują, że istotna część ataków jest możliwa z powodu błędów w *API* [122, 124]. Dobrze zaprojektowane *API* powinno właściwie chronić poufne dane, a także uniemożliwiać jakąkolwiek eskalację uprawnień przy wrogim użyciu [5]. Jest to szczególnie ważne, ponieważ interfejs programistyczny jest jednym z najczęściej i najszerzej wykorzystywanych elementów każdego systemu. Proponuje się zatem, aby nadawać wysoki priorytet technikom wykrywania błędów właśnie w tych warstwach oprogramowania.

Niestety, opracowania wskazujące istotność ochrony *API*, poza zadeklarowaniem potrzeby wykorzystywania takich narzędzi, nie wskazują ani istniejących implementacji, ani praktycznych dróg rozwoju dziedziny. Tym co jest możliwe w obecnej chwili, jest szczególnie intensywne aplikowanie znanych metod weryfikacji takich jak statyczna i dynamiczna analiza czy formalna weryfikacja modelu. Tematyka narzędzi pozwalających wzmacniać bezpieczeństwo *API* pozostaje jednak otwartym tematem badawczym.

4.6. Środowiska próbnego uruchamiania

Podstawowym narzędziem, dostępnym na praktycznie wszystkich platformach i wykorzystywanym przez znaczną liczbę programistów jest debugger. W zależności od zaawansowania i środowiska, pozwala on na różne operacje — od najprostszy, jak nadzorowania wykonania w trybie krokowym aż do zaawansowanych, takich jak np. wstrzykiwania kodu w locie. O ile z perspektywy programisty debugger wiernie odwzorowuje wykonanie programu, o tyle istotnie wpływa on na model wykonania widoczny z poziomu systemu operacyjnego. Standardowo, operacje takie jak: podłączenie debuggera do działającego programu (np. za pomocą wywołania systemowego `ptrace()`), natrafienie na pułapkę przerwania, dostęp do konkretnych zmiennych czy wykonanie wywołania systemowego powodują wstrzymanie wykonania programu i transfer sterowania do procesu nadzorczego. Powoduje to, że wykonanie w środowisku debuggera istotnie zmienia szeregowanie wątków procesu nadzorowanego. W związku z tym można spotkać się z opiniami, że w środowisku wielowątkowym debuggery częściej ukrywają błędy, niż pozwalają je wykryć i zrozumieć [122].

Ze względu na powszechne użycie debuggerów i przywiązanie programistów do tego typu narzędzi, niezbędne wydają się zatem zabiegi, które mogłyby ulepszyć i ułatwić wykorzystanie ich w programach współbieżnych. Typowy, nowoczesny debugger, jak np. *gdb*, rozpoznaje istnienie wątków i pozwala śledzić ich wykonanie. Niektóre zaawansowane narzędzia (np. *Microsoft Visual Studio Debugger* czy wspomniane *gdb*) udostępniają także wsteczny tryb pracy krokowej (ang. *replay debugging, reversible debugging*), który umożliwia cofanie się w wykonaniu programu, co może być przydatne przy próbie poznania przyczyny błędu tuż po obserwacji. Wciąż wydaje się jednak, że w tej dziedzinie istnieje możliwość dalszego rozwoju. W przeszłości prowadzono rozległe badania m.in. nad różnymi wariantami

uruchamiania wstecznego [88] czy rozwiązaniami, które oprócz pełnienia funkcji debuggera wykrywają i automatycznie naprawiają błędy [98]. Niestety, żaden z tych pomysłów nie został wdrożony do środowiska produkcyjnego. Wspomaganie próbnego uruchamiania aplikacji współbieżnych pozostaje zatem otwartym kierunkiem rozwoju narzędzi programistycznych.

4.7. Testowanie

Testowanie oprogramowania współbieżnego jest zadaniem niezwykle trudnym. Standardowe testy klienckie (jak np. testowanie akceptacyjne) zwykle przebiegają w sposób identyczny, jak dla programów sekwencyjnych, jednak ze względu na niedeterminizm, zupełnie inaczej należy podejść do automatycznych testów kodu. Popularne testy jednostkowe w modelu *black box* są praktycznie niemożliwe do wykorzystaniu w środowisku współbieżnym ze względu na synchronizację. W programach wielowątkowych większość testów wymaga uwzględnienia kontekstu — współdziałania jednostek. Można zatem tworzyć testy modułowe i integracyjne sprawdzające większe fragmenty systemu, ale znów poprzez niedeterminizm niemożliwe jest uzyskanie powtarzalności, która jest kluczowym aspektem procesu testowania. Testowanie programów współbieżnych wymaga zatem zupełnie odmiennego, innowacyjnego podejścia, wspieranego przez dedykowane temu narzędzia. Przegląd osiągnięć na tym polu zaprezentowano poniżej.

4.7.1. CHESS

CHESS [76, 86] to nowe narzędzie tworzone przez firmę *Microsoft*, które służy do systematycznego i wyczerpującego testowaniu aplikacji współbieżnych napisanych w *C*, *C++* i językach z platformy *.NET*. Testowanie za pomocą *CHESS* polega na tworzeniu scenariuszy, czyli interesujących akcji zachodzących w systemie (np. jednoczesna obsługa wiadomości wyłączenia i wykonania operacji I/O w sterowniku urządzenia), które będą sprawdzane. Sprawdzanie polega natomiast na wykorzystaniu metod weryfikacji modelu do systematycznego pokrycia wszystkich możliwych serializacji planisty poprzez wielokrotne uruchomienia. Jeśli błąd zostanie odnaleziony w jednym z przebiegów, to *CHESS* umożliwi wielokrotne uruchamianie pod nadzorem debuggera w celu ułatwienia poznania źródła błędu.

Z testowaniem za pomocą *CHESS* wiąże się jednak kilka ograniczeń. Przede wszystkim, aby testy były powtarzalne, muszą być idempotentne. Dodatkowo w celach optymalizacji *CHESS* nie przechowuje żadnych stanów (np. inicjalnego) — muszą być one skonfigurowane przed użyciem. Co istotne, *CHESS* nie przeszukuje całej przestrzeni wszystkich możliwych przeplotów — w celu ograniczenia eksplozji stanów programiści narzędzia zdecydowali się wprowadzić wywłaszczanie wyłącznie na punktach synchronizacji (poprzez opakowania funkcji *API* systemowego) i w miejscach, które odwołują się do zmiennych ulotnych, mogących brać udział w wyścigach.

Pomimo wyżej opisanych ograniczeń, ze względu na łatwość użycia, narzędzie to zyskuje coraz większą popularność i zaczyna być używane również poza samą firmą tworzącą *CHESS*. Istnieje zatem realna szansa, że narzędzie to wyjdzie z fazy badawczej i stanie się standardowym sposobem testowania poprawności aplikacji współbieżnych. Na zakończenie warto dodać, że *CHESS* od niedawna stał się

projektem z otwartym kodem źródłowym, a więc można spodziewać się jego dynamicznego rozwoju, napędzanego przez społeczność.

4.7.2. ConTest

ConTest [28, 50] to komercyjne narzędzie stworzone i rozwijane przez firmę *IBM*, które ma za zadanie wymuszać ujawnianie się błędów we współbieżności w aplikacjach na platformę *JVM*. Idea działania tego narzędzia jest bardzo prosta — program poddawany testom jest wzbogacany o wywołania metod powodujących zmianę pracy planisty takich jak: `sleep()`, `yield()` oraz `priority()`. Skoki do tych funkcji umieszczane są w miejscach, które mogą okazać się interesujące z punktu widzenia niezawodności jak np. dostępy do pamięci dzielonej czy zdarzenia synchronizacyjne. Perturbacje wprowadzane do pracy planisty w ten sposób, według twórców *ConTest*, zwiększają prawdopodobieństwo ujawnienia błędu.

Ze względu na losowość, skuteczność i zasadność wykorzystania tego narzędzia jest podważana [69]. Pomimo że *ConTest* posiada opcję nagrywania i odtwarzania testów, nie ma pewności, że raz ujawniony błąd zostanie powtórzony po odtworzeniu — planista *JVM*, nawet w połączeniu z omawianym narzędziem nie jest deterministyczny. W związku z powyższym nie należy spodziewać się, że *ConTest* będzie dalej intensywnie rozwijany i doczeka się wielu wdrożeń.

4.7.3. MultithreadedTC

MultithreadedTC [99, 116] to biblioteka do jednostkowego testowania aplikacji współbieżnych napisanych w języku *Java*. Pojedynczy test zbudowany zgodnie z główną ideą *MTC* sprawdza pojedynczy, zdefiniowany z góry przeplot wątków. Dzięki temu programista może upewnić się, że konkretna serializacja wątków nie powoduje błędnego działania. Opisywane narzędzie oferuje testowanie w stylu *black box* i opiera się na znanej i powszechnie wykorzystywanej bibliotece do testów jednostkowych *JUnit*.

Aby wymusić konkretne przeploty wątków, *MTC* wprowadza koncepcję metronomu — logicznego zegara, który porusza się naprzód wyłącznie w momencie, gdy wszystkie wątki są zablokowane. Typowy test napisany za pomocą omawianego narzędzia składa się z wielu wątków, które mogą oczekiwać na konkretny stan licznika metronomu (metoda `waitForTick()`) lub wywoływać metody testowanych obiektów. Poprawność działania kodu sprawdzana jest za pomocą asercji znanych z *JUnit* wzbogaconych dodatkowo o asercje stanu metronomu (m.in. `assertTick()`).

MTC jest jedynym narzędziem, które pozwala precyzyjnie i powtarzalnie sprawdzać konkretne przeploty. Zastosowane w opisywanym rozwiązaniu pomysły powodują, że doskonale nadaje się ono do łatwego wzbogacenia procesu standardowego testowania jednostkowego — testy *MTC* są łatwe do napisania, wykonują się równie szybko, co testy sekwencyjne i są powtarzalne. Z drugiej strony, przyjęty w *MTC* sposób wymuszania przeplotów nie pozwala na tworzenie większych testów modułowych i integracyjnych, gdyż ze względu na izolację nie istnieje możliwość wpływania na działanie obiektów fasadowych. Dodatkowo przyjęty w *MTC* interfejs oparty na liczniku może wydawać się nieco niewygodny, gdyż wymaga od programisty tworzącego test śledzenia aktualnego stanu wyrażonego liczbowo, co może powodować pomyłki typu *off–by–one*, czyli pomyłki w indeksowaniu sekwencji. Wygodniejsze mogłoby być zastosowanie np. nazwanych zdarzeń, łatwiejszych

do zapamiętania. Pomimo tego *MultithreadedTC* jest bardzo interesującym narzędziem, które dobrze pasuje do aktualnego ekosystemu wytwarzania oprogramowania obiektowego. Można zatem spodziewać się powstawania rozwiązań podobnych, adekwatnych dla innych platform.

4.7.4. Komentarz

Przedstawione w niniejszej sekcji narzędzia stanowią przegląd rozwiązań, które wyszły poza fazę wyłącznie badawczą i znajdują zastosowanie w środowisku produkcyjnym. Spośród omówionych narzędzi, najbardziej obiecującym wydaje się być *CHESS*, który zyskuje coraz większy rozgłos i być może niebawem stanie się częścią oprogramowania *Visual Studio*. Testowanie jest niezwykle ważnym elementem wytwarzania oprogramowania. Nie należy zatem poprzestawać wyłącznie na rozwijaniu istniejących rozwiązań — wciąż potrzebne są nowe pomysły i sposoby, które pomogą polepszać jakość oprogramowania. Na zakończenie warto dodać, że wraz z rozwojem testowania aplikacji współbieżnych, rozwijają się także dziedziny pokrewne, takie jak miary skuteczności testów i metryki pokrycia [14, 67, 113]. Można zatem spodziewać się, że przy aktualnym zainteresowaniu, w niedługim czasie testowanie aplikacji współbieżnych stanie się na równi popularne z testowaniem typowych aplikacji sekwencyjnych.

4.8. Podsumowanie

Jak pokazano w niniejszym rozdziale, rozwój narzędzi wspierających programowanie współbieżne jest bardzo nierównomierny — dostępność konkretnego rodzaju rozwiązań jest bardzo często uzależniona od wybranej platformy programistycznej. Ponadto nie wszystkie rodzaje aplikacji wspomagających wytwarzanie oprogramowania rozwijają się równie dynamicznie. Rozwiązania opierające się na rozważaniach teoretycznych, takie jak transformacje i weryfikacje modelu, są często wypierane przez rozwiązania heurystyczne takie jak statyczna analiza kodu. Z kolei narzędzia do dynamicznej analizy wykonania posiadają wielu wysokiej jakości reprezentantów, w momencie gdy debuggery i narzędzia do testowania są dopiero w początkowej fazie rozwoju i dostosowywania do programowania współbieżnego.

Spostrzeżenia te uwypuklają konieczność dalszego technik, które mogłyby wspomóc ewolucję istniejących i powstawanie nowych aplikacji wielowątkowych i wielozadaniowych. Z tego powodu w kolejnym rozdziale przedstawiono kompletny proces tworzenia, począwszy od koncepcji poprzez projekt, implementację aż do testowania i badania skuteczności, nowego narzędzia, które może wspomóc zarówno uruchamianie jak i testowanie programów współbieżnych.

5. Narzędzie Coconut

W 1 i 2 rozdziale niniejszej pracy zaprezentowano, na przykładach z istniejących aplikacji, jakie problemy powszechnie występują w programach współbieżnych. W toku badania, pośród wniosków rozważono także przyczyny i skutki pojawiających się błędów. Dyskusję nad najważniejszymi przyczynami, tj. przyczynami technicznymi omawianych problemów podjęto w rozdziale 3. Zauważono jednak, że pośród wszystkich technik niezawodnego programowania współbieżnego, narzędzia wspomagające wytwarzanie oprogramowania są zagadnieniem, które rozwija się najwolniej. Tematykę zbadano zatem dokładniej w rozdziale 4, gdzie dostrzeżono konkretne braki i płaszczyzny rozwoju tej dziedziny. W wyniku tych spostrzeżeń powstało innowacyjne narzędzie o nazwie *Coconut*, które jest zwieńczeniem wniosków z badań przeprowadzonych w toku powstawania niniejszej pracy.

Coconut (od **Co**mpact **Con**currency **U**nit **T**esting) to nowatorskie, eksperymentalne narzędzie do strukturalnego testowania jednostkowego i próbnego uruchamiania programów współbieżnych napisanych w językach *C* i *C++* poprzez deterministyczne weryfikowanie konkretnych przeplotów. W niniejszym rozdziale szczegółowo opisano cały proces powstawania wspomnianego rozwiązania: od koncepcji, która opiera się na badaniach z poprzednich rozdziałów poprzez projekt aż do właściwej implementacji. Rozdział zakończono wnikliwymi testami, oceną użyteczności narzędzia *Coconut* i skuteczności zastosowanej w nim koncepcji testowania, a także krótkim podsumowaniem.

5.1. Koncepcja

Pierwszym punktem prezentacji narzędzia *Coconut* jest przedstawienie ogólnej koncepcji, z której to rozwiązanie się wywodzi. Niniejsze rozważania mają na celu pokazanie, skąd wziął się pomysł na *Coconut* oraz przekonanie czytelnika, iż takie właśnie narzędzie jest faktycznie potrzebne. Niniejsze rozważania stanowią nieformalny wstęp do wymagań, które opisano w kolejnym podrozdziale.

5.1.1. Platforma

Analizując błędy opisane w rozdziałach 1 i 2 łatwo zauważyć, że większość z nich dotyczy programów pisanych w językach *C* i *C++*. Taki stan rzeczy wynika głównie z dwóch powodów. Przede wszystkim, większość krytycznego dla współczesnej informatyki, powszechnie wykorzystywanego oprogramowania (m.in. systemy operacyjne, kompilatory, przeglądarki) powstało z przyczyn historycznych i wydajnościowych właśnie w tych językach. Po drugie, języki te powstały na tyle dawno, że nie zostały odpowiednio przystosowane do pisania programów współbieżnych i jak pokazano w rozdziałach 3 i 4, dopiero ostatnio podjęto działania zmierzające do odmienienia tej niekorzystnej sytuacji. Ze względu na konieczność, a także trudność w pielęgnowaniu i rozwijaniu istotnych dla inżynierii oprogramowania projektów

tworzonych w tradycyjnych technologiach, podjęto decyzję o stworzeniu narzędzia wspierającego programowanie współbieżne w *C* i *C++*.

Nie istnieje wspólny dla wszystkich powszechnie wykorzystywanych systemów operacyjnych niskopoziomowy interfejs tworzenia programów wielowątkowych. Z tego powodu powstające narzędzia, o ile nie wykorzystują warstwy pośredniej (np. w postaci biblioteki *boost*), nie są przenośne. Niestety, nie istnieje wysokiej jakości *middleware* dla języka *C*, który wspierałby wszystkie popularne systemy operacyjne. Z tego powodu konieczne jest podjęcie decyzji determinującej docelową platformę rozważanego narzędzia. Wybraną platformą są systemy implementujące standard *POSIX* (w tym systemy *UNIX–owe*, *Linux* oraz *Mac OS X*), gdyż najwięcej istotnych usług (np. serwery, klastry, systemy czasu rzeczywistego) wykorzystuje właśnie te systemy operacyjne jak swój fundament.

Wybrana platforma, tj. *POSIX C/C++* stwarza jednak pewną istotną trudność. Środowisko wykonawcze tych języków jest surowe — nie udostępnia żadnych mechanizmów wpływania na wykonywany program, jak np. refleksja. Możliwość realizacji zaawansowanego narzędzia w tak ubogim środowisku będzie zatem także przedmiotem badań.

5.1.2. Typ narzędzia

Znając docelową platformę, warto zastanowić się, jakiego typu narzędzie byłoby najbardziej przydatne dla jej programistów. Podczas analizy przeprowadzonej w rozdziale 4, dla *C* i *C++* opisano wysokiej jakości narzędzia do analizy dynamicznej *Valgrind* i *ThreadSanitizer* (zob. podrozdział 4.4), ciekawe rozwiązania do weryfikacji modelu w postaci *Zing* i *VeriSoft* oraz wyrafinowaną, hybrydową aplikację *CHESS*. Wśród zestawu narzędzi brakuje zatem rozwiązań do statycznej analizy kodu oraz testowania i próbnego uruchamiania. Wydajna i efektywna analiza kodu programów napisanych na platformę *C++* jest zadaniem niezwykle trudnym i podatnym na błędy ze względu na bardzo skomplikowaną gramatykę tego języka. Co więcej, tak jak zauważono w podrozdziale 4.3.3, wiele istniejących analizatorów statycznych dedykowanych programom sekwencyjnym rozpoznaje błędy, które mają znaczenie dla współbieżności.

W kręgu zainteresowania pozostają zatem narzędzia do testowania i próbnego uruchamiania. Wybór pomiędzy tymi dwoma zagadnieniami nie musi być jednak wykluczający, gdyż łatwo doszukać się związków pomiędzy testowaniem, a próbnym uruchamianiem. Bardzo często po procedurze testowania, w wyniku znalezienia błędu, uruchamiany jest debugger lub odwrotnie — po odnalezieniu błędu tworzony jest test zabezpieczający przed ponownym wprowadzeniem usterki. Częsta integracja obu technik nie jest więc sytuacją dziwną czy niespotykaną.

Zagadnienie testowania dla *C* i *C++* jest także o tyle ciekawe, że powszechnie stosowane techniki, takie jak testowanie jednostkowe czy modułowe, zostały spopularyzowane długo po stworzeniu tych języków i z tego powodu nie stanowią one części ekosystemu wytwarzania oprogramowania na tej platformie. Jest to sytuacja niepożądana, gdyż wymienione wyżej metody udowodniły w ostatnich latach swoją przydatność w nowoczesnej inżynierii oprogramowania. Warto zatem dodatkowo zastanowić się, jak można ten stan rzeczy zmienić.

Jak pokazały badania z rozdziału 1, a także przytoczone tam miary ilościowe, 30% błędów jest poprawianych w niewłaściwy sposób za pierwszym razem. Konieczne wydaje się zatem stworzenie narzędzi, które umożliwiałyby sprawdzanie

tworzonych łat. Sam proces testowania kodu współbieżnego musi zostać natomiast istotnie ulepszony.

Podczas badań dodatkowo zauważono, że wielokrotnie programistom znany jest wyłącznie skutek błędu, nie istnieją natomiast praktycznie żadne przesłanki, dlaczego usterka się ujawniła. Warto zatem zastanowić się nad metodami, które umożliwiłyby stopniowe odkrywanie błędów w programie — zaczęcie od miejsca, w którym prawdopodobnie błąd wystąpił i systematyczne analizowanie kolejnych fragmentów, które mogły usterkę sprowokować. Jak pokazano w kolejnym podrozdziale, obie techniki tj. testowanie i próbne uruchamianie kodu współbieżnego, mogą doskonale się uzupełniać.

5.1.3. Niedeterminizm

Największą przeszkodą w rzetelnym testowaniu kodu współbieżnego jest brak powtarzalności spowodowany niedeterminizmem wykonania. Niedeterminizm, o którym mowa, wynika natomiast ze sposobu działania planisty systemu operacyjnego, który ze względu na wydajność działa w sposób nieprzewidywalny. Usunięcie lub chociaż częściowe ograniczenie niedeterminizmu i zapewnienie powtarzalności w środowisku rozwojowym mogłoby zatem znacząco pomóc w testowaniu i uruchamianiu.

Doskonałym dowodem postawionej wyżej hipotezy są przytoczone w rozdziałach 1 i 2 przykłady błędów. Prawie wszystkie opisane usterki dało się zilustrować za pomocą diagramów strzałkowych, które przedstawiają serializacje, które prowadzą do ujawnienia się błędów. Jeśli zatem istniałaby możliwość syntetycznego wymuszenia takiego uszeregowania instrukcji wątków jak na diagramie, to możliwe byłoby deterministyczne obserwowanie występowania błędów.

Spostrzeżenia te popierają również opisy algorytmów omówionych wcześniej narzędzi (m.in. *Java PathFinder*, *CHESS*, *ConSeq*), które w sposób systematyczny przeszukują przestrzeń możliwych przeplotów, aby odnaleźć błędy. Przeszukiwanie wyczerpujące jest jednak ograniczone przez eksplozję kombinatoryczną. Sprawdzenie wszystkich dozwolonych przebiegów wymaga wykonania programów przekształconych do postaci sekwencyjnej w liczbie wersji równej:

$$M = \frac{(\sum_{i=1}^{N} n_i)!}{\prod_{i=1}^{N}(n_i!)}$$

gdzie N oznacza liczbę wątków, a n_i liczbę instrukcji w i–tym wątku [89]. Dla przykładu, dla dwóch niewielkich wątków liczących po 50 instrukcji ($N = 2$, $n_1, n_2 = 50$), $M \approx 10^{29}$. Automatyczne sprawdzanie przeplotów jest zatem powolne i wymusza wiele uproszczeń, które mogą maskować błędy.

Warto zatem zastanowić się czy lepszym pomysłem nie byłoby zaoferowanie programiście możliwości sprawdzenia kilku interesujących go przebiegów. Takie podejście wymaga dodatkowego zaangażowania twórcy programu, ale wykorzystuje wiedzę osoby najlepiej zorientowanej w sposobie działania testowanej aplikacji i pozwala łatwo rozwiać pojawiające się wątpliwości w kwestii niedeterminizmu. Ponadto taka metoda weryfikacji jest ideologicznie podobna do wspomnianego wcześniej testowania jednostkowego i dodatkowo umożliwia deterministyczne uruchamianie wadliwych programów. Nieco podobne podejście prezentuje wspomniana wcześniej, interesująca biblioteka *MultithreadedTC* dla języka *Java*, która jednak

nie ustrzegła się kilku uciążliwych błędów (zob. podrozdział 4.7.3) i ograniczeń (zob. podrozdział 5.1.4).

Proponowany sposób testowania znajduje również poparcie w statystykach przytoczonych w podrozdziale 1.2.5. Ponad 70% błędów we współbieżności to błędy naruszenia kolejności i atomowości, które można ujawnić za pomocą modelowanych przebiegów [69]. Co więcej, 96% błędów ujawnia się w sposób deterministyczny przy konkretnym, częściowym przeplocie dokładnie dwóch wątków [69]. Programista tworzący proponowane testy aplikacji współbieżnych nie musi zatem rozważać skomplikowanych zależności między wieloma wątkami, a jedynie pomiędzy ich parami. Dodatkowo 92% błędów ujawnia się w sposób deterministyczny przy konkretnym, częściowym przeplocie co najwyżej czterokrotnego dostępu do pamięci [69]. Tworzone testy nie muszą zatem dotyczyć wielu odległych odwołań do pamięci, ale powinny skupiać się na małych grupach operacji lokalnych. W krytycznym, brzegowym przypadku test o pokryciu aż 92% musi sprawdzać zaledwie sześć różnych przeplotów. Znajomość tych charakterystyk, w połączeniu z wiedzą programisty o działaniu jego programu powoduje, że wyposażony w odpowiednie narzędzie, może on stworzyć testy istotnie lepsze niż automatyczny weryfikator.

5.1.4. Testowanie w modelu white i black box

Tradycyjne testowanie jednostkowe obiektowych aplikacji sekwencyjnych odbywa się w modelu *black box*, co znaczy, że testowane jest wyłącznie to, co udostępnia interfejs publiczny, przy pominięciu rozważań na temat konstrukcji wewnętrznej. Typowo takie testowanie odbywa się w trójfazowym wzorcu *AAA* składającym się z: aranżacji (ang. *arrange*), która przygotowuje środowisko i dane wejściowe; wykonania (ang. *act*), które przeprowadza pojedynczą operację oraz weryfikacji (ang. *assert*), w której sprawdzana jest poprawność wyników. Dobrze skonstruowany test ma niewielkie rozmiary, nie jest obarczony żadnymi założeniami na temat budowy wewnętrznej systemu i jest powtarzalny. Powtarzalność testu gwarantuje, że niezależnie od tego ile razy zostanie uruchomiony, otrzymany wynik będzie taki sam. Jednostkowe testy *black box* wymagają zatem eliminacji jakiejkolwiek losowości wykonania.

Wspominane kilkukrotnie narzędzie *MultithreadedTC* podejmuje próbę pełnego wpisania się w ten tradycyjny model, narzucając jednak bardzo ostre ograniczenia. *MTC* pozwala na testowanie klas, w których synchronizacja jest zupełnie ukryta przed klientem (np. za pomocą wzorca monitora), co efektywnie uniemożliwia zastosowanie tego rozwiązania dla bardziej złożonych systemów. Silnie wiąże się to jednak z poczynionymi założeniami, gdyż w ogólności, w związku z niedeterminizmem, testowanie jednostkowe programów współbieżnych w modelu *black box* nie jest w ogóle możliwe, ponieważ wynik każdego uruchomienia potencjalnie zależy od pracy nieprzewidywalnego planisty.

Narzucającym się sposobem umożliwienia powtarzalnego wykonania jest modelowanie źródła niedeterminizmu czyli wspomnianego planisty. Sposób działania tego elementu systemu bardzo silnie zależy jednak od wewnętrznej realizacji programu (punktów synchronizacji, wywołań blokujących, wywłaszczania itd.), co rodzi omawianą sprzeczność z modelem *black box*. Współbieżności nie można zatem rozważać abstrahując zupełnie od wewnętrznej budowy, gdyż to ona jest składnikiem realizacji.

Do testowania implementacji typowo wykorzystuje się model *white box* (nazywany inaczej testowaniem strukturalnym) i wydaje się, że jest on także odpowiedni dla programów współbieżnych [113]. Wykorzystanie ww. techniki umożliwia deterministyczną instrumentację pracy planisty, która transformuje program współbieżny do (częściowo) sekwencyjnego, co z kolei ze względu na powtarzalność pozwala zastosować elementy modelu *black box*.

Na mocy powyższych ustaleń, narzędzie opisywane w niniejszym rozdziale oferuje testowanie w modelu *white box*. Na zakończenie warto zauważyć, że testowanie strukturalne jest nadzbiorem testowania typu *black box*, a więc prezentowane rozwiązanie będzie posiadało możliwości nie mniejsze niż choćby *MultithreadedTC*.

5.1.5. Sposób realizacji

Idealne narzędzie, niezależnie od tego jak zaawansowane możliwości oferuje, powinno być łatwe w obsłudze. Powszechnie wiadomo jednak, że te dwie cechy zwykle są sobie przeciwstawne — im bardziej wyrafinowane narzędzie, tym trudniejsze jest w obsłudze. Bogactwo opcji nie jest jednak jedyną cechą, która wpływa na łatwość zastosowania. Podczas analizy raportów o błędach z rozdziałów 1 i 2 zauważono, że programiści bardzo przywiązani są do swoich przyzwyczajeń i niechętnie przyjmują nowe rozwiązania, ale bardzo ochoczo korzystają z narzędzi, które już dobrze znają i używają na co dzień. W związku z tym wprowadzenie zupełnie nowego rozwiązania do już funkcjonującego środowiska bywa zwykle trudne i powolne. Z tego też powodu za jedną z kluczowych cech opisywanego narzędzia przyjęto, aby bezproblemowo integrowało się z istniejącym ekosystemem i mogło być natychmiastowo wdrożone w każdym projekcie.

5.1.6. Kluczowe ustalenia

W poprzednich podrozdziałach zarysowano kluczowe cechy i założenia na których opiera się *Coconut*. Ustalenia te zebrano w poniższej liście.

Coconut to narzędzie:
— działające na platformach *POSIX–owych*,
— dla programów napisanych w językach *C* i *C++*,
— służące do testowania i próbnego uruchamiania programów współbieżnych,
— pozwalające wymusić określoną sekwencję przeplotów,
— działające w modelu *white box*,
— łatwe do natychmiastowego wdrożenia w dowolnym projekcie.

Sformalizowaną postać tych cech przedstawiono w kolejnej sekcji w postaci wymagań funkcjonalnych i niefunkcjonalnych.

5.2. Projekt

Po opracowaniu klarownej i spójnej koncepcji narzędzia można przejść do jego projektowania, czyli fazy, w której pomysły przybierają formalną postać i prowadzą do podjęcia konkretnych decyzji. Pomimo że proces tworzenia oprogramowania składa się zwyczajowo z trzech równie ważnych etapów: projektowania, implementacji i testowania, to od tego pierwszego zależy najwięcej. Tradycyjnie faza projektowania zaczyna się od zebrania i opisania wymagań projektowych, które następnie powinny być dokładnie przeanalizowane. Wnioski pochodzące z tych

działań służą następnie do podjęcia decyzji projektowych sterujących kolejnym etapem, czyli implementacją. W ostatnie fazie, czyli podczas testowania, efekty pracy projektanta służą do weryfikacji i oceny stworzonego systemu. Etap projektowania oprogramowania jest zatem nieodzownym elementem każdego przedsięwzięcia informatycznego i ma zasadniczy wpływ na pozostałe fazy i tym samym na całokształt produktu.

Niniejszy podrozdział w naturalny sposób odzwierciedla wyżej wymienioną kolejność działań. W następnej jego części przedstawiono wymagania zarówno funkcjonalne jak i niefunkcjonalne. W kolejnym kroku wymagania te zostały dokładnie przeanalizowane, a na ich podstawie podjęto i opisano wstępne decyzje projektowe, które posłużą przy implementacji.

5.2.1. Wymagania

Dobre praktyki inżynierii oprogramowania [52] nakazują, aby każdy projekt informatyczny zaczynać od zebrania, zgrupowania i wyczerpującego opisania wymagań funkcjonalnych i niefunkcjonalnych. Tak jak wcześniej wspomniano, właściwe wykonanie tego etapu przedsięwzięcia ma istotny wpływ na kolejne kroki takie jak projektowanie architektury, implementacja, testowanie i wreszcie końcowa weryfikacja i ocena.

Wymagania służą sformalizowaniu i sprecyzowaniu tego, co ma być efektem prac nad projektem. Sterują one procesem projektowania, który ma doprowadzić do implementacji produktu rozwiązującego odpowiednie problemy. Warto dodatkowo nadmienić, że w fazie prac nad projektem, wymagania stanowią narzędzie, które pozwala ocenić stopień zaawansowania prac i oszacować czas potrzebny do zakończenia całego przedsięwzięcia. Na etapie testowania wymagania są natomiast podstawą do konstrukcji testów. Odpowiednio zaprojektowany przypadek testowy pozwala rozstrzygnąć czy pewien podzbiór wymagań został spełniony. Wszystkie przypadki testowe powinny natomiast w idealnej sytuacji pokrywać cały zbiór wymagań projektu. Wiedza zdobyta w fazie testowania opartego na wymaganiach pozwala w końcu rzetelnie ocenić jakość końcowego produktu. Wymagania zdefiniowane na początku projektu mają zatem fundamentalny wpływ na sposób jego realizacji i końcowy sukces.

Wymagania dla omawianego narzędzia sporządzono przede wszystkim na podstawie pracy koncepcyjnej z podrozdziału 5.1, która opiera się na badaniach z poprzednich rozdziałów niniejszej pracy. Wszystkie wymogi zostały sporządzone na podstawie obserwacji konkretnych potrzeb i problemów programistów aplikacji współbieżnych, a także w oparciu o funkcjonujące i sprawdzone rozwiązania w tej dziedzinie.

Wymagania funkcjonalne

Lista wymagań funkcjonalnych projektu *Coconut* nie jest obszerna — zawiera jedynie ustalenia dotyczące sposobu testowania, funkcje oferowane przez dodatkowe elementy rozwiązania, a także warunki wykorzystania w środowisku rozwojowym i produkcyjnym. Ograniczenie liczby możliwości narzędzia wynika przede wszystkim z chęci stworzenia rozwiązania łatwego we wdrożeniu i użyciu, które jednocześnie dobrze wykonuje postawione mu zadania. Dodatkowo należy pamiętać, że

prezentowanie rozwiązanie ma charakter eksperymentalny, dla którego funkcjonalność wykraczającą poza ustalone ramy podstawowe można z łatwością dodać w późniejszym czasie.

1. **Modelowanie dowolnych przeplotów**
   Narzędzie powinno umożliwiać modelowanie dowolnych przeplotów. W praktyce oznacza to, że rozwiązanie pozwala wskazać porządek liniowy instrukcji wszystkich (wybranych przez użytkownika) wątków w programie. Taki zabieg pozwala przeprowadzić deterministyczne wykonanie wybranego fragmentu programu poprzez ograniczenie współbieżności. Zgodnie z dyskusją z punktu 5.1.3 ułatwia to uruchamianie i testowanie programów współbieżnych.

2. **Tworzenie testów różnych przeplotów**
   Narzędzie powinno umożliwiać tworzenie testów różnych przeplotów. W wielu przypadkach może zdarzyć się, że programista wykorzystujący narzędzie chciałby w łatwy sposób sprawdzić kilka możliwych sekwencji przełączeń wątków. W takiej sytuacji wielokrotne zmienianie ustalonego porządku może okazać się niewygodne, a więc musi istnieć sposób na automatyczne przetestowanie kilku różnych kolejności. Ta funkcjonalność jest szczególnie potrzebna do wykonywania automatycznych testów jednostkowych.

3. **Wykrywanie niemożliwych przeplotów**
   Narzędzie powinno udostępniać wykrywanie przeplotów, których wystąpienie nie jest możliwe. Programista może zażądać sprawdzenia czy pewna sekwencja przełączeń wątków jest dopuszczalna, np. w celach weryfikacji pewnych założeń. W sytuacji, gdy taka kolejności instrukcji nie jest możliwa (np. ze względu na synchronizację czy komunikację), narzędzie powinno ten fakt zasygnalizować. Próba wymuszenia sekwencji niedopuszczalnej, w zależności od sposobu realizacji, może doprowadzić do zakleszczenia np. na skutek wstrzymania wątku w sekcji krytycznej, co jest zjawiskiem niepożądanym.

4. **Sprawdzanie warunków**
   Narzędzie powinno umożliwiać sprawdzanie podstawowych warunków: prawdziwość/fałszywość podanego wyrażenia, wystąpienie instrukcji przed/po nazwanym przeplocie. Rozwiązanie powinno informować o niespełnieniu konkretnego warunku poprzez zasygnalizowanie miejsca, w którym nastąpiło naruszenie. Funkcje do weryfikacji założeń występują w praktycznie każdej bibliotece do testowania jednostkowego, najczęściej w postaci procedur typu `Assert.AreEqual()`, `Assert.That()` czy `Assert.NotNull()`.

5. **Tymczasowe wyłączanie instrumentacji**
   Narzędzie powinno umożliwiać łatwe wyłączanie instrumentacji tak, aby istniała możliwość łatwego przełączania się pomiędzy wersją standardową i tą do testowania. W środowisku rozwojowym pożądane jest, aby programiści mogli łatwo wybierać, jaki tryb uruchomienia jest im potrzebny.

6. **Izolacja środowiska produkcyjnego**
   Narzędzie powinno umożliwiać całkowite wyłączenie przy kompilacji produkcyjnej. Jeśli program, dla którego zastosowano instrumentację, został skompilowany do użytku w środowisku produkcyjnym, to rozważane rozwiązanie nie powinno mieć żadnego wpływu (funkcjonalnego ani wydajnościowego) na testowany wcześniej program.

Wymagania niefunkcjonalne

Nieliczne wymagania niefunkcjonalne dla *Coconut*, poza zdefiniowaniem podstawowych parametrów takich jak platforma czy sposób kompilacji, koncentrują się przede wszystkim na wygodzie użycia, która jest bardzo istotną cechą rozważanego projektu. Z rozwiązaniem nie są zawiązane żadne ograniczenia ilościowe, w szczególności też wydajnościowe.

1. **Platforma**
   Narzędzie powinno działać na systemach *POSIX–owych* i umożliwiać testowanie programów napisanych w *C* i *C++*. Platforma ta została wybrana, zgodnie z rozważaniami z podrozdziału 5.1.1, ze względu na chęć wspierania istotnego oprogramowania napisanego w tradycyjnych, gorzej dostosowanych do współbieżności technologiach.

2. **Sposób kompilacji**
   Narzędzie powinno być możliwe do skompilowania za pomocą kompilatora *gcc 4.7* lub nowszego, przy użyciu biblioteki standardowej *glibc 2.17* lub nowszej. Dozwolone jest wykorzystanie innych bibliotek i narzędzi, pod warunkiem, że spełniają wymienione wcześniej warunki.

3. **Efekt pracy**
   Efektem prac nad narzędziem może być zarówno biblioteka jak i samodzielna aplikacja. Kluczowym zagadnieniem jest jednak to, żeby finalny produkt wykorzystywał jak najwięcej koncepcji znanych programistom tak, aby był łatwy do użycia. Implementacja powinna również bezproblemowo integrować się ze środowiskami do testowania i próbnego uruchamiania.

4. **Ingerencja w kod testowanej aplikacji**
   Dopuszcza się, aby interakcja z narzędziem odbywała się przy ingerencji w kod testowanej aplikacji. Zgodnie z rozważaniami z punktu 5.1.4 testowanie ma odbywać się w modelu *white box*, a więc w sposób zależny od implementacji wewnętrznej.

5. **Możliwości wdrożenia**
   Narzędzie powinno być możliwe do wdrożenia w każdym projekcie realizowanym w przestrzeni użytkownika na opisanej powyżej platformie. Jak pokazano w podrozdziale 5.1.5 łatwość wdrożenia ma kluczowe znaczenie dla powszechnego wykorzystania narzędzi, a zatem należy przywiązać szczególną wagę do odpowiedniego, przyjaznego dla użytkowników końcowych, sposobu realizacji.

5.2.2. Decyzje projektowe

Dogłębna analiza wymagań projektowych sformułowana w poprzednim punkcie, w połączeniu z wiedzą z badań przeprowadzonych w poprzednich rozdziałach, umożliwia podjęcie wstępnych decyzji projektowych. Postanowienia te mają charakter bardzo ogólny i stanowią wskazówki dla etapu implementacji.

Dla narzędzia *Coconut*, w fazie projektowania konieczne było podjęcie decyzji dotyczącej ogólnej architektury rozwiązania, gdyż zasadniczo wpływa ona na sposób implementacji. Drugim niezbędnym do rozważenia zagadnieniem był szkic interfejsu, który umożliwiałby spełnienie dwóch najważniejszych wymagań, tj. WF–1 i WF–2, dotyczących modelowania i testowania przebiegów. Posiadając wiążące decyzje w obu kwestiach, można przystąpić do eksperymentowania z implementacją. Pozostałe istotne z punktu widzenia projektu zagadnienia zostały omówione w kolejnym punkcie dotyczącym implementacji.

Architektura rozwiązania

Zasadniczym problemem do rozwiązania przy tworzeniu omawianego narzędzia jest modelowanie pracy niedeterministycznego planisty tak, aby szeregował on wątki w żądany sposób. W związku z tym, że programy pisane w *C* i *C++* nie posiadają własnego środowiska uruchomieniowego i wątki tworzone w tych językach są wątkami poziomu systemu operacyjnego, to są one zarządzane przez planistę systemowego. Pierwszym pomysłem rozwiązania przedstawionego problemu jest zatem dodanie do jądra dodatkowego trybu planisty, który umożliwiałby swobodne sterowanie szeregowaniem wątków. Takie rozwiązanie pozwalałoby efektywnie testować praktycznie każdą aplikację wykorzystującą wątki jądra — nie tylko napisaną w *C* i *C++*. Co więcej, modelowanie nie wymagałoby żadnej ingerencji w kod testowanej aplikacji — odbywałoby się całkowicie przezroczyście.

Opisane wyżej rozwiązanie posiada jednak bardzo istotną wadę, którą jest trudność w implementacji wynikająca ze skomplikowania dziedziny. Planista jest jednym z najdelikatniejszych elementów jądra systemu operacyjnego i każdy błąd w jego kodzie zwykle kończy się fatalną awarią całego systemu. Dodatkowo, aby wskazywać punkty wywłaszczeń wątków, ten moduł systemu musiałby na bieżąco odczytywać symbole debuggera zaszyte w kodzie binarnym aplikacji. Parsowanie plików pochodzących z przestrzeni użytkownika w przestrzeni jądra otwierałoby całą gamę możliwości ataku na system operacyjny. To rozwiązanie obniżałoby zatem ogólne bezpieczeństwo środowiska rozwojowego. Na zakończenie warto wspomnieć, że realizacja w przestrzeni jądra byłaby zupełnie nie przenośna pomiędzy systemami *POSIX–owymi* i naruszałaby wymaganie WNF–1.

Wady wyżej opisanego pomysłu zdają się przeważać nad zaletami. Warto zatem zastanowić się nad innym sposobem modelowania niedeterminizmu planisty. Typowo w przestrzeni użytkownika, aby wymusić określoną kolejność instrukcji w wątkach, wykorzystuje się konstrukcje synchronizacyjne, takie jak np. semafory. Można zatem zastanowić się, czy nie byłyby one wystarczające dla rozważanego problemu.

Dostępne w systemach *POSIX–owych* prymitywy synchronizacyjne pozwalają zawiesić wątek w oczekiwaniu na akcję ze strony innego wątku (np. podniesienie semafora). Umieszczając w kodzie odpowiednie sekwencje takich operacji można ograniczyć współbieżność i spowodować deterministyczne wykonanie sekwencji instrukcji wątków. Takie rozwiązanie jest stosunkowo łatwe w implementacji i pod

warunkiem wykorzystania odpowiednich mechanizmów, jest przenośne pomiędzy różnymi systemami. Istotną wadą tego pomysłu jest konieczność ingerencji w testowaną aplikację — punkty synchronizacji muszą być umieszczone w kodzie w sposób jawny.

Kompromisem pomiędzy tymi dwoma pomysłami jest stworzenie emulatora wykonującego program w sztucznym środowisku, podobnym do tego, który wykorzystywane jest w narzędziu *Valgrind*. Takie rozwiązanie pozwala na implementację dowolnego planisty i przezroczystą instrumentację wykonywanego programu. Stworzenie funkcjonalnego emulatora jest jednak zadaniem co najmniej tak samo trudnym, jak modyfikacja systemowego planisty. Proces emulacji wymaga odtworzenia nie tylko elementów systemu operacyjnego, ale także samego procesora dekodującego i wykonującego instrukcje. Łatwo zatem o przeoczenia, które mogą spowodować, że symulowane wykonanie będzie znacząco odbiegało od tego obserwowanego w rzeczywistym środowisku. Dodatkowo warto nadmienić, że emulowanie jest procesem wolnym, co mogłoby uniemożliwić przetestowanie bardziej złożonych aplikacji.

Ostatecznie zdecydowano o zaimplementowaniu rozwiązania opartego na systemowych prymitywach synchronizacyjnych. To podejście, mimo wspomnianych wad, jest w zupełności wystarczające do wykonania eksperymentów i ocenienia użyteczności testowania opartego o modelowanie pracy planisty. Co więcej, jak pokazano w podrozdziale 5.5 dotyczącym oceny implementacji, istnieją interesujące sposoby, które mogłyby zniwelować znaczenie opisanych wyżej problemów związanych z wybranym podejściem.

Szkic interfejsu

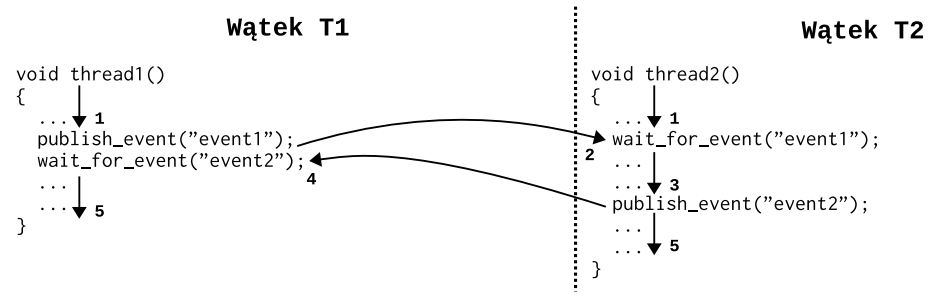
Po ustaleniu ogólnej architektury omawianego narzędzia możliwe jest zaproponowanie interfejsu, który umożliwi modelowanie przeplotów między wątkami w sposób określony w wymaganiach. W podobnym do omawianego rozwiązania narzędziu *MultithreadedTC* zastosowano blokowanie oparte na zegarze logicznym, który porusza się naprzód w momencie, gdy wszystkie wątki są zablokowane. Z tym pomysłem wiążą się dwie poważne wady: zdarzenia w czasie „nazwane” są liczbami, które łatwo pomylić, a dodatkowo niejasnym może być, w którym momencie kończy się cykl zegara.

W odpowiedzi na wyżej opisane problemy zaproponowano interfejs oparty na nazwanych zdarzeniach. Nazwami zdarzeń są dowolne ciągi znaków. Wątek może oczekiwać na zdarzenie poprzez wskazanie jego nazwy i pozostaje zablokowany aż do momentu, gdy inny wątek opublikuje zdarzenie o takiej samej nazwie. Próba oczekiwania na opublikowanym zdarzeniu nie powoduje blokowania — wątek jest dalej kontynuowany. Koncepcyjny przykład tego interfejsu zaprezentowano na rysunku 5.1.

Takie rozwiązanie jest bardzo proste w użyciu i przypomina uproszczony schemat działania zmiennych warunkowych. Nazwane, opisowe zdarzenia pozwalają łatwo zorientować się w kodzie, a mechanizm publikacji i subskrypcji jest programistom powszechnie znany. Jedynym użytkowym problemem omawianego interfejsu jest utrzymanie spójności wykorzystywanych nazw — można to jednak zapewnić stosując stałe lub definicje preprocesora.

Interfejs oparty na zdarzeniach doskonale nadaje się do testowego uruchamiania. Podczas tego procesu programista może dowolnie modelować przeploty i ograniczać współbieżność, aby sprawdzać zachowania systemu w przypadku różnych

serializacji wątków. Przy wykorzystaniu debuggerów udostępniających dynamiczne wstrzykiwanie kodu, instrumentacja pracy planisty może odbywać się w pełni dynamicznie.



Rysunek 5.1: Przykład działania interfejsu opartego na zdarzeniach. Numerami oznaczono sekwencję działań, ten sam numer oznacza współbieżne wykonanie. Na początku oba wątki działają swobodnie (1). Wątek *T2* wstrzymuje swoje działanie w oczekiwaniu na publikację zdarzenia `"event1"` przez wątek *T1* (2). Po publikacji wątek *T1* natychmiast zawiesza się w oczekiwaniu na zdarzenie `"event2"`, a wątek *T2* kontynuuje swoje działanie (3). Po publikacji zdarzenia `"event2"` przez wątek *T2* (4), oba wątki kontynuują współbieżne wykonanie (5).

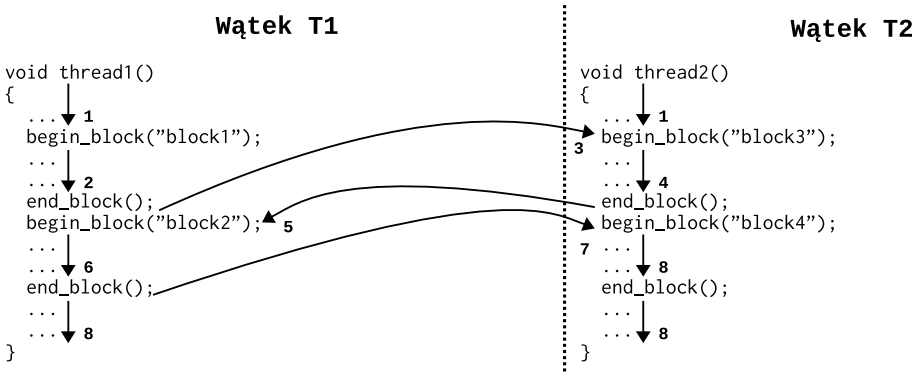
Omawiany sposób modelowania przeplotów może być również pomocny przy wyszukiwaniu błędów. Programista sprawdzający wystąpienie błędów typu *use after free* może np. pozwolić działać wątkowi zwalniającemu pamięć, wstrzymując jednocześnie pracę innych wątków do momentu faktycznego zwolnienia, a po wznowieniu ich pracy obserwować zachowanie systemu. Taki sposób wykorzystania pomaga szczególnie w diagnozie problemów w działaniu wielu autonomicznych zadań.

Pomimo wszystkich wyżej opisanych zalet, interfejs oparty na nazwanych zdarzeniach pozwala w pojedynczej kompilacji zweryfikować tylko jeden przebieg. Jest zatem bardzo przydatny w czynnym poszukiwaniu błędów, jednak nie nadaje się do tworzenia testów jednostkowych. Do rozwiązania tego problemu zaproponowano interfejs oparty na nazwanych blokach. Blok w tym rozwiązaniu jest rozumiany podobnie do bloku wyznaczonego przez symbole `{` oraz `}` w języku *C*, z tą różnicą, że jego granice wyznaczają wywołania funkcji `begin_block("nazwa")` oraz `end_block()`. Po podzieleniu kodu w wątkach na odpowiednie, nazwane bloki, programista może zażądać wykonania bloków w interesującej go kolejności liniowej. Wykonanie wątku blokowane jest przy wywoływaniu funkcji `begin_block()` do momentu, aż bezpośrednio poprzedzający go blok zostanie zakończony za pomocą wywołania `end_block()`.

Przy odpowiednim podziale na bloki (wynikającym np. z układu sekcji krytycznych) programista ma możliwość łatwego sprawdzenia dowolnej sekwencji interesujących przeplotów. Dysponując takim narzędziem możliwe jest zatem tworzenie testów jednostkowych sprawdzających wiele założeń poprzez wielokrotne, zautomatyzowane uruchomienia. Warto dodać, że koncepcja bloków powinna być dla programistów w naturalny sposób zrozumiała, gdyż występują one w wielu językach programowania. Interfejs ten jest jednak nieco trudniejszy w wykorzystaniu niż ten oparty na nazwanych zdarzeniach.

Podstawowy przykład wykorzystania interfejsu opartego na nazwanych blokach przedstawiono na rysunku 5.2. Kolejność zażądana w przykładzie jest możliwa

i programista będzie mógł zaobserwować interesujące go zjawiska. Nic nie stoi jednak na przeszkodzie, żeby żądaną sekwencją było np. "block2", "block1", "block3", "block4" — taka sekwencja nie może wystąpić ze względu na tzw. porządek programu w wątku *T1*. Narzędzie zgodnie z wymaganiem WF–3 wykryje jednak taką sytuację i poinformuje o tym programistę.



Rysunek 5.2: Przykład działania interfejsu opartego na blokach dla kolejności "block1", "block3", "block2", "block4". Numerami oznaczono sekwencję działań, ten sam numer oznacza współbieżne wykonanie. Na początku oba wątki działają swobodnie (1). Następnie wątek *T2* zawiesza się przed rozpoczęciem bloku "block3", gdyż "block1" jest jego poprzednikiem. *T1* po wykonaniu (2) "block1", sygnalizuje koniec bloku (3) i zawiesza się w oczekiwaniu na rozpoczęcie bloku "block2", którego poprzednikiem jest "block3". Dalsze wykonanie przebiega analogicznie aż do rozpoczęcia bloku "block4", który nie jest niczyim poprzednikiem — wątki znowu działają współbieżnie (8).

Ze względu na równy priorytet wymagań WF–1 i WF–2 w rozważanym narzędziu zdecydowano się zaimplementować oba interfejsy. Ich odmienne przeznaczenie powoduje, że uzupełniają się i pozwalają wykorzystać rozwiązanie zarówno do próbnego uruchamiania jak i tworzenia testów jednostkowych. Analizując ewentualne sposoby realizacji nie dostrzeżono również, aby istniały przeciwwskazania do zaimplementowania obu metod modelowania przeplotów.

5.3. Implementacja

Implementacja to faza życia projektu, w której następuje faktyczna realizacja przedsięwzięcia informatycznego. Efektami pracy na tym etapie są najczęściej kod oraz dokumentacja projektowa. Działania podejmowane podczas implementacji wynikają w większości z decyzji podjętych w fazie projektowania, a końcowe wyniki tego etapu są oceniane w kolejnej fazie — podczas testów. Implementacja jest wyjątkowym etapem projektu, bo poza faktycznym wykonywaniem produktu bardzo często zawiera elementy zarówno projektowania jak i testowania. Projektowanie wykonywane w fazie implementacji zwykle służy doprecyzowaniu elementów, które nie mogły być określone wcześniej, a także reagowaniu na zmiany w specyfikacji jak i doświadczeniu tworzących. Testowanie podczas implementacji służy natomiast weryfikacji postępu w pracach projektowych i kontroli czy działania zmierzają we właściwym kierunku.

Niniejszy podrozdział zawiera opis najważniejszych decyzji i spostrzeżeń poczynionych podczas implementacji. W omawianej sekcji przybliżono również elementy wewnętrznej budowy narzędzia *Coconut*, a także zamieszczono skrócony opis *API*, który umożliwia zrozumienie przykładów prezentowanych w kolejnej części rozdziału dotyczącej testów i badań.

5.3.1. Wybór technologii

Wybór konkretnej technologii do realizacji narzędzia *Coconut* był prosty i w całości podyktowany przez WNF–1. Aby narzędzie było przenośne pomiędzy systemami *POSIX–owymi* i bezproblemowo integrowało się z projektami napisanymi w językach *C* i *C++*, do instrumentacji pracy wątków wybrano bibliotekę *Pthreads*. Jest ona częścią standardu *POSIX* i z tego powodu można zakładać, że będzie dostępna na wszystkich systemach z tej rodziny.

5.3.2. Sposób wykorzystania

Zgodnie z postanowieniami z etapu projektowania (zob. podrozdział 5.2.2), narzędzie *Coconut* ma dostarczać zbioru funkcji, który będzie umożliwiał instrumentację pracy planisty. Naturalnym wyborem sposobu realizacji w tym przypadku jest zatem stworzenie biblioteki udostępniającej ww. funkcje. Postać biblioteczna jest łatwa w użyciu w praktycznie każdym projekcie i co bardzo istotne, ta formuła udostępniania funkcjonalności jest dobrze znana programistom. Takie podejście zapewnia spełnienie wymagań WNF–3 oraz WNF–5 dotyczących ergonomii użycia narzędzia.

W początkowej wersji biblioteki, od programisty wykorzystującego to rozwiązanie wymagano jedynie, aby dostarczył funkcję `c_main()` aranżującą testy, gdyż funkcja `main()` inicjująca struktury danych była zaimplementowana wewnątrz biblioteki. Podobne rozwiązanie zastosowano w innych bibliotekach testujących m.in. *boost::test*. Zauważono jednak, że takie podejście ułatwia tworzenie testów jednostkowych, ale utrudnia pracę z próbnym uruchamianiem ze względu na konflikty symboli konsolidatora. Dodatkowo taki sposób działania może być mylący dla programisty, gdyż występuje wbrew powszechnym metodom uruchamiania programów napisanych w *C/C++*. Ostatecznie zdecydowano zatem, że programista wykorzystujący bibliotekę *Coconut* standardowo tworzy funkcję `main()`, ale to na nim spoczywa odpowiedzialność wywołania na początku funkcji inicjującej `c_init()` i na końcu zwalniającej zasoby `c_free()`.

Opisywana w niniejszej pracy wersja *Coconut* kompilowana jest do postaci biblioteki statycznej `libcoconut.a`, która gotowa jest do linkowania z dowolnym projektem. Linkowanie z `libcoconut.a` wymaga dołączenia biblioteki `pthread`. Nie istnieją żadne techniczne przeciwwskazania, aby uczynić *Coconut* biblioteką linkowaną dynamicznie. Dokładną instrukcję kompilacji i wykorzystania *Coconut* zamieszczono w dodatku B.

5.3.3. Wewnętrzne struktury danych i blokowanie

Kluczowym elementem narzędzia są trzy struktury danych, które przechowują informacje o aktualnym stanie biblioteki. Pierwszą z tych struktur jest lista wątków, która zawiera ich identyfikatory i flagi do oznaczania wątków aktualnie zablokowanych. *Coconut* rejestruje wyłącznie te wątki, które wykorzystują interfejs biblioteki — pozostałe wątki są dla narzędzia niewidoczne. Lista wątków jest wykorzystywana przy wykrywaniu przeplotów, które są niemożliwe (zob. podrozdział 5.3.4).

Kolejną istotną strukturą danych jest lista wszystkich nazwanych zdarzeń. Pojedynczy obiekt przechowuje nazwę oraz informację o tym, czy dane zdarzenie zostało już opublikowane. Z każdym zdarzeniem skojarzona jest także zmienna warunkowa oraz odpowiadający jej muteks, które wykorzystywane są do blokowania w oczekiwaniu na publikację. Nowe zdarzenie tworzone jest w momencie publikacji albo w momencie rozpoczęcia oczekiwania na publikację.

Analogiczną strukturą do listy zdarzeń jest lista bloków, która jest tworzona w momencie ustalenia testowanego przeplotu. Pojedynczy obiekt bloku przechowuje następujące informacje: nazwę, stan (utworzony, w trakcie oczekiwania, uruchomiony lub zakończony), identyfikator wątku właściciela, listę bloków poprzedzających oraz stempel czasowy, który wykorzystywany jest przy szeregowaniu bloków zagnieżdżonych. W celu wstrzymywania wątków, które muszą oczekiwać na zakończenie bloku poprzedzającego, podobnie jak w przypadku zdarzeń, z każdym blokiem skojarzono zmienną warunkową oraz odpowiadający jej muteks.

W narzędziu *Coconut* do blokowania wątków wykorzystano zmienne warunkowe, gdyż doskonale nadają się one do oczekiwania na zmianę wartości zmiennych — stanu bloku lub zdarzenia. Ich użycie ułatwia stworzenie kodu wolnego od wyścigów, a także jest bardzo wygodne ze względu na mechanizm rozgłaszania (ang. *broadcast*) zmian.

5.3.4. Wykrywanie niemożliwych przeplotów

Jednym z najtrudniejszych, a zarazem najważniejszych zagadnień podczas implementacji było wykrywanie niemożliwych przeplotów zdefiniowane w wymaganiu WF–3. Podczas testowania może zdarzyć się, że programista spróbuje wymusić sekwencję, która nie może wystąpić np. ze względu na porządek programu czy synchronizację. Użycie opisanych wcześniej zmiennych warunkowych w takich sytuacjach doprowadzi do zakleszczenia. Takie zachowanie programu może być dla programisty dezorientujące, a dodatkowo utrudnia automatyzację testowania, gdyż wymaga manualnego restartowania testów.

W związku z powyższym w *Coconut* zaimplementowano mechanizm wykrywania zakleszczeń wynikających z próby wymuszenia niemożliwej serializacji wątków. Algorytm postępowania jest nietypowym rozwiązaniem heurystycznym, gdyż bardzo ograniczona wiedza o środowisku wykonawczym nie pozwala zaimplementować bardziej zaawansowanych metod np. opartych na wyszukiwaniu cykli w grafie oczekiwania. Języki *C/C++* nie posiadają środowiska uruchomieniowego, które mogłoby dostarczyć informacji o aktualnie istniejących wątkach i ich stanie, a więc wszelkie informacje muszą być zarządzane bezpośrednio przez narzędzie. Wykrywanie zakleszczeń w *Coconut* polega na utworzeniu dodatkowego wątku typu *watchdog*, który cyklicznie (domyślnie co jedną sekundę) sprawdza spełnienie któregokolwiek z warunków:

— lista wątków jest pusta — mechanizm nie został jeszcze użyty,

— logiczny zegar zliczający zdarzenia wewnętrzne narzędzia zmienił się od ostatniego cyklu — działania wątków mają charakter postępujący,
— którykolwiek z zarejestrowanych wątków nie jest w stanie zablokowanym,
— wszystkie wątki zakończyły pracę.

Jeśli żaden warunek nie jest spełniony, sygnalizowane jest zakleszczenie, wszystkie zdarzenia zostają opublikowane i wszystkie bloki zostają oznaczone jako gotowe do startu. Ze względu na opisane wcześniej ograniczenia środowiska, algorytm może błędnie zgłaszać zakleszczenia w sytuacji, gdy istnieją niezablokowane wątki, które nie zostały jeszcze zarejestrowane przez narzędzie i minął pojedynczy cykl wykrywania. Prostym rozwiązaniem tego problemu jest np. publikowanie próbnego zdarzenia zaraz po starcie każdego wątku. Zakleszczenie może nie być rozpoznane, jeśli wynika z blokujących wywołań systemowych, jednak rozwiązanie tego problemu nie jest możliwe bez wsparcia ze strony systemu operacyjnego.

5.3.5. Konfiguracja

Ze względu na dążenie do prostoty użycia dołożono starań, aby maksymalnie ograniczyć konieczność konfiguracji narzędzia. Z tego powodu jedynymi konfigurowalnymi elementami systemu są: jego aktywacja, kompilacja oraz sposób pracy wątku typu *watchdog*. Opis sposobu konfiguracji zamieszczono poniżej.

Zgodnie z WF–5 musi istnieć możliwość tymczasowego wyłączania instrumentacji. W narzędziu można to osiągnąć na dwa sposoby. Pierwszy wymaga usunięcia z kodu wywołania funkcji inicjującej `c_init()`. To rozwiązanie okazało się jednak mało wygodne ze względu na konieczność ponownej kompilacji i z tego powodu zaimplementowano drugi sposób. Opiera się on na zmiennej środowiskowej `C_DISABLE`. Jeśli została ona zdefiniowana i ma wartość liczbową różną od 0, to wszystkie funkcje narzędzia będą kompletnie ignorowane.

W wymaganiu WF–6 zażądano natomiast, aby istniała możliwość kompilacji produkcyjnej, która całkowicie pominie funkcje narzędzia i nie będą miały one żadnego wpływu na aplikację. Problem ten rozwiązano za pomocą symbolu preprocesora `NCOCONUT` — jeśli zostanie on zdefiniowany przy kompilacji, żadne wywołanie biblioteczne nie znajdzie się w wynikowym kodzie binarnym.

Podczas wstępnego testowania narzędzia zauważono, że nie istnieje uniwersalna dla wszystkich programów długość cyklu wątku wykrywającego zakleszczenia. Z tego powodu dodano dwa sposoby zmiany domyślnej wartości tego czasu. Pierwszą metodą jest zmienna środowiskowa `C_WATCHDOG_TICK`, która przyjmuje wartość liczbową wyrażającą długość cyklu w sekundach. Drugi (priorytetowy) sposób polega natomiast na wywołaniu funkcji `c_set_watchdog_tick()` z parametrem oznaczającym to samo, co wartość opisanej wcześniej zmiennej środowiskowej.

5.3.6. API

Publiczny interfejs biblioteki *Coconut* zaprojektowano zgodnie z koncepcją przedstawioną w podrozdziale 5.2.2. Ze względu na brak przestrzeni nazw w *C*, nazwy wszystkich funkcji poprzedzono przedrostkiem `c_`, aby ułatwić odróżnianie procedur bibliotecznych od kodu produkcyjnego i aby uniknąć ewentualnych konfliktów nazw. Poniżej zaprezentowano skrócony opis *API* udostępnianego przez

bibliotekę tak, aby umożliwić zrozumienie przykładów prezentowanych w kolejnych sekcjach. Pełny opis interfejsu znajduje się w dodatku A.

Uruchamianie:

— `c_init()` — inicjalizuje wewnętrzne struktury danych, musi być wywołana jako pierwsza funkcja biblioteki,

— `c_free()` — zwalnia pamięć zajmowaną przez wewnętrzne struktury danych, powinna zostać wywołana jako ostatnia funkcja biblioteki,

— `c_set_watchdog_tick(unsigned int tick)` — ustala długość pojedynczego cyklu wątku typu *watchdog* na `tick` sekund.

Interfejs zdarzeń:

— `c_publish_event(const char *event)` — publikuje zdarzenie o nazwie `event`,

— `c_wait_event(const char *event)` — zawiesza wątek w oczekiwaniu na zdarzenie o nazwie `event`.

Interfejs bloków:

— `c_begin_block(const char *block)` — rozpoczyna blok o nazwie `block` (nazwa nie powinna zawierać przecinków i średników) i zawiesza tym samym wątek w oczekiwaniu na zakończenie poprzednika,

— `c_end_block())` — kończy ostatnio rozpoczęty blok — sygnalizuje ten fakt następnikom,

— `c_set_blocks_interleaving(const char *interleaving)` — ustala żądany przeplot bloków; nazwy kolejnych bloków powinny być oddzielone średnikiem; w przypadku oddzielenia kilku nazw przecinkiem, bloki uruchamiane są jednocześnie po spełnieniu poprzedników,

— `c_cond_block(COND, BLOCK)` — makro, które może być wykorzystywane w miejscach, gdzie oczekiwane jest wyrażenie warunkowe, jak np. klauzula `if-else` (zob. rysunek 5.5); gwarantuje ono, że przed ewaluacją `COND` zostanie wywołane `c_begin_block(BLOCK)`, a po `c_end_block()`; makro zwraca obliczoną wartość wyrażenia `COND`; właściwa kolejność wywołań, a także jednokrotna ewaluacja `COND` wynikają z punktów sekwencyjnych (ang. *sequence point*) języków *C/C++*.

Sprawdzanie warunków:

— `c_assert_*(COND, FMT, ...)` — seria makr do definiowania pewnych założeń (`COND`) sprawdzanych podczas wykonania; jeśli założenie nie zostało spełnione, na standardowym wyjściu błędów zostanie wyświetlony komunikat `FMT` z podanymi parametrami (analogicznie jak dla funkcji `printf()`) poprzedzony nazwą pliku, nazwą funkcji i numerem linii, w której nastąpiło naruszenie.

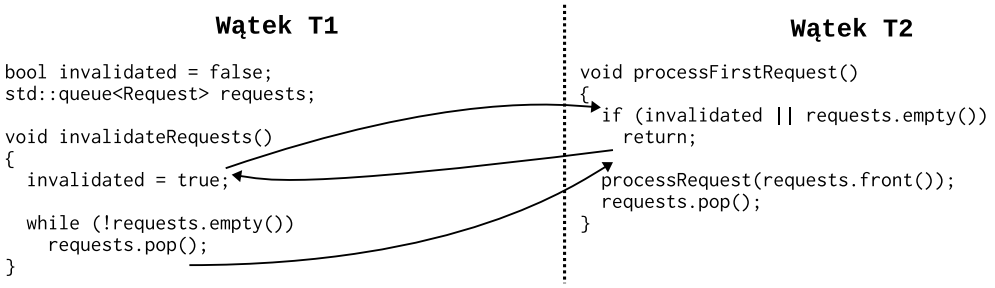
5.4. Testy i badania

Po zakończeniu fazy implementacji możliwe jest przejście do etapu testowania. Testowanie to niezwykle ważny, często jednak zaniedbywany, etap tworzenia oprogramowania. W wielu przypadkach, znaczna część działań związanych ze sprawdzaniem zaimplementowanego produktu skupia się na końcu cyklu wytwarzania, w momencie gdy brakuje już czasu i funduszy na dalsze działania projektowe. Testowanie jest jednak fazą, w której solidnie przeprowadzone działania pozwalają stwierdzić, czy spełnione zostały wymagania, a tym samym czy oprogramowanie można uznać za ukończone.

W przypadku *Coconut* jest jednak nieco inaczej. Ze względu na eksperymentalną postać narzędzia, testowanie ma charakter badania użyteczności tego typu rozwiązań (tzn. opartych na deterministycznym testowaniu przeplotów). Z tego powodu, w niniejszym podrozdziale po przedstawieniu sztucznego przykładu wprowadzającego, skoncentrowano się na zastosowaniu *Coconut* w funkcjonujących projektach. W związku z tym najpierw podjęto próbę zastosowania omawianego rozwiązania dla błędów prezentowanych w rozdziałach 1 i 2, a następnie przedstawiono studium przypadku, w którym wykorzystano *Coconut* do odnalezienia, zdiagnozowania i naprawienia nieznanego wcześniej błędu w projekcie z otwartym kodem źródłowym.

5.4.1. Przykład wprowadzający

W ramach wprowadzenia do narzędzia *Coconut* wykorzystano je do zdiagnozowania prostego wyścigu w programie zaprezentowanym na rysunku 5.3 (standardowo błędny przebieg zaznaczono strzałkami). Oczywiście nie ma gwarancji, że przy uruchomieniu jednocześnie wątków *T1* i *T2* błąd faktycznie wystąpi — jest to zależne od niedeterministycznego planisty. Za pomocą *Coconut* możliwa jest jednak odpowiednia instrumentacja przeplotów tak, aby w każdym uruchomieniu błąd był obserwowalny.



Rysunek 5.3: Prosty błąd wynikający z braku synchronizacji. Strzałki pokazują przebieg, który kończy się awarią.

Na początku do wymuszenia przeplotów pokazanych na rysunku wykorzystano interfejs zdarzeń nazwanych. Odpowiednie wywołania funkcji zostały dodane dokładnie w miejscach wskazywanych przez strzałki. Zdarzenia mają opisowe nazwy tak, aby łatwo było je rozpoznać. Wynikowy kod przedstawiono na rysunku 5.4.

Warto zauważyć, że bezpośrednio przed pobraniem pierwszego elementu z kolejki dodano sprawdzenie czy faktycznie nie jest ona pusta. Przy wymuszonych przez narzędzie przeplotach to założenie nie jest spełnione i na standardowe wyjście błędów zostanie wypisany komunikat: `[main.cpp:void processFirst():41]: Assert failed: requests queue is empty!`.

Dla wyżej rozważonego przykładu stworzono również testy za pomocą interfejsu nazwanych bloków. W omawianym programie wyróżniono cztery bloki związane z: unieważnieniem zapytań, opróżnieniem kolejki, sprawdzeniem unieważnienia i wykorzystaniem pierwszego elementu kolejki. Program podzielony na takie bloki zaprezentowano na rysunku 5.5.

Jeśli przed uruchomieniem wątków wywołana zostanie funkcja wyznaczająca kolejność `c_set_blocks_interleaving()` z argumentem

Wątek T1

```
bool invalidated = false;
std::queue<Request> requests;

void invalidateRequests()
{
  c_wait_event("checked");
  invalidated = true;

  while (!requests.empty())
    requests.pop();
  c_publish_event("cleared");
}
```

Wątek T2

```
void processFirstRequest()
{
  if (invalidated || requests.empty())
    return;
  c_publish_event("checked");

  c_wait_event("cleared");
  c_assert_false(requests.empty(),
                      "requests queue is empty!");
  processRequest(requests.front());
  requests.pop();
}
```

Rysunek 5.4: Instrumentacja wykonania za pomocą zdarzeń nazwanych narzędzia *Coconut*, która sprawia, że błędny przeplot ujawnia się w sposób deterministyczny — przy każdym uruchomieniu.

Wątek T1

```
bool invalidated = false;
std::queue<Request> requests;

void invalidateRequests()
{
  c_begin_block("inv");
  invalidated = true;
  c_end_block();

  c_begin_block("clear");
  while (!requests.empty())
    requests.pop();
  c_end_block();
}
```

Wątek T2

```
void processFirstRequest()
{
  if (c_cond_block(invalidated || requests.empty(),
                                          "check"))
    return;

  c_begin_block("process");
  c_assert_false(requests.empty(),
                        "requests queue is empty!");
  processRequest(requests.front());
  requests.pop();
  c_end_block();
}
```

Rysunek 5.5: Instrumentacja wykonania za pomocą bloków nazwanych narzędzia *Coconut*. Błąd ujawnia się dla kolejności `check`, `inv`, `clear`, `process`.

`"check;inv;clear;process"`, to błąd będzie ujawniał się przy uruchomieniu w sposób deterministyczny, o czym poinformuje ostrzeżenie niespełnionego założenia. Dla przykładu, jeśli ustawiona zostanie kolejność `"inv;check;clear;process"`, to program zadziała zgodnie z intencją programisty.

5.4.2. Badania na zaprezentowanych błędach

Po przedstawieniu działania narzędzia *Coconut* na spreparowanym problemie, warto sprawdzić użyteczność rozwiązania w faktycznie wykorzystywanych aplikacjach. Do tego celu wykorzystano przykłady błędów przedstawione w rozdziałach 1 i 2.

Pierwszym przykładem poddanym analizie jest naruszenie kolejności wynikające z błędnego założenia, że wywołanie zwrotne nie wykona się przed instrukcją następującą bezpośrednio po funkcji, która te wywołanie stworzyła. Błąd ten przedstawiono na rysunku 1.7. Dodanie odpowiedniego zdarzenia ujawniającego tę usterkę jest bardzo proste i odwzorowuje przytoczony wykres strzałkowy. Wynikowy kod zaprezentowano na rysunku 5.6.

Wątek T1

```
int ReadWriteProc(...)
{
  ...
  PBReadAsync(&p);
  c_wait_event("callback set");
  io_pending = TRUE;
  c_publish_event("caller set");
  ...
}
```

Wątek T2

```
void DoneWaiting(...) // ReadWriteProc callback
{
  ...
  io_pending = FALSE;
  c_publish_event("callback set");
  c_wait_event("caller set");
  c_assert_false(io_pending, "io_pending is TRUE!");
  ...
}
```

Rysunek 5.6: Powtarzalny test dla błędu z rysunku 1.7.

Wykorzystanie *Coconut* dla omawianego błędu pozwala programiście w powtarzalny, deterministyczny sposób zaobserwować wystąpienie usterki, a także dokładnie poznać jej genezę. Problematyczne jest oczywiście wymyślenie odpowiedniej aranżacji zdarzeń, która pozwala wymusić wystąpienie usterki przy każdym uruchomieniu. Jest na to jednak prosty sposób. Jeśli programista po raz pierwszy spotykał się z tym błędem, to najprawdopodobniej był w stanie szybko zauważyć, że pomimo wykonania odczytu i wywołania zwrotnego, zmienna `io_pending` ma nieodpowiednią wartość i dlatego program zakleszcza się. Prawdziwym wyzwaniem było zapewne natomiast wymyślenie skąd pochodzi ten niewłaściwy stan zmiennej. Skutecznym sposobem działania w tym przypadku mogłoby być otoczenie każdego zapisu blokiem narzędzia *Coconut*, a następnie przetestowanie każdej sekwencji, nawet teoretycznie niemożliwej. W ten sposób za pomocą mechanizmu asercji łatwo byłoby wykryć i wyizolować sekwencję przeplotów, która prowadzi do błędu.

Wątek T1

```
int ReadWriteProc(...)
{
  ...
  c_begin_block("caller set");
  io_pending = TRUE;
  c_end_block();

  PBReadAsync(&p);

  c_begin_block("assert");
  c_assert_false(io_pending, "io_pending is TRUE!");
  c_end_block();
  ...
}
```

Wątek T2

```
void DoneWaiting(...) // ReadWriteProc callback
{
  ...
  c_begin_block("callback set");
  io_pending = FALSE;
  c_end_block();
  ...
}
```

Rysunek 5.7: Bloki do testów poprawki błędu z rysunku 1.7. Blok `"assert"` został wprowadzony sztucznie i powinien być zawsze wykonywany jako ostatni — służy sprawdzeniu czy po wszystkich przełączeniach wątków, ostateczna wartość `io_pending` równa jest `FALSE`.

Poprawnym sposobem naprawienia opisywanego błędu jest zamiana kolejności wywołania funkcji `PBReadAsync()` z ustawianiem wartości zmiennej `io_pending`. Ta poprawka może budzić pewne wątpliwości, gdyż jest bardzo prosta i nie angażuje żadnych dodatkowych mechanizmów synchronizacji. Może być ona jednak zweryfikowana w sposób analogiczny do opisywanego wyżej. Wszystkie zapisy zmiennej `io_pending` należy otoczyć blokami, a następnie przetestować komplet uszeregowań stworzonych bloków. Niektóre uruchomienia zostaną opatrzone komunikatem

o niemożliwych przeplotach, żadne uruchomienie nie powinno natomiast powodować błędu lub naruszenia założeń. Bloki do testów poprawki zaprezentowano na rysunku 5.7.

Narzędzie *Coconut* może zostać również wykorzystane do powtarzalnego odtwarzania błędów naruszenia niepodzielności. Przykład takiego użycia zaprezentowano na przykładzie błędu z rysunku 1.3, w którym wskaźnik mógł zostać wyzerowany po sprawdzeniu jego wartości. Aby wyizolować ten błąd zdefiniowano dwa zdarzenia — wykonania sprawdzenia i wyzerowania wskaźnika. Wynikowy kod przedstawiono na rysunku 5.8.

Wątek T1

```
void innobase_mysql_print_thd(...)
{
  ...
  if (thd->proc_info)
  {
    ...
    c_publish_event("after check");
    c_wait_event("nulled");
    c_assert_true(thd->proc_info != 0,
                          "proc_info is null!");
    fputs(thd->proc_info, f);
  }
  ...
}
```

Wątek T2

```
bool do_command(...)
{
  ...
  c_wait_event("after check");
  thd->proc_info = 0;
  c_publish_event("nulled")
  ...
}
```

Rysunek 5.8: Powtarzalny test dla błędu z rysunku 1.3.

Wykorzystanie interfejsu zdarzeń nazwanych dla błędów naruszenia niepodzielności jest tak samo proste, jak dla błędów naruszenia kolejności. Niestety, diagnoza tego typu błędów jest zdecydowanie trudniejsza. Ciężko jest *a priori* określić sekwencję przeplotów, która może doprowadzić do wystąpienia usterki. Dogłębnego badania wymagają fragmenty kodu, w których wykonywane warunkowo są operacje na zmiennych współdzielonych. Tego typu analiza może być jednak wydatnie uproszczona przy użyciu interfejsu bloków, który dostosowany jest również do błędów naruszenia niepodzielności poprzez makro `c_cond_block()`.

Najprostszy schemat działania dla wyszukiwania i testowania błędów typu *TOCTOU* polega na otoczeniu blokiem: operacji sprawdzenia, operacji modyfikującej oraz operacji następujących po sprawdzeniu i podjęciu próby wymuszenia takiej właśnie kolejności. Dla pozostałych błędów naruszenia niepodzielności należy natomiast próbować tworzyć bloki w sekcjach, które powinny wykonywać się w sposób niepodzielny i narzucać przeploty z konfliktującymi operacjami w innych wątkach.

Ostatnim analizowanym przykładem jest błąd naruszenia kolejności przedstawiony na rysunku 1.8, w wyniku którego dochodziło do zwolnienia pamięci przed ostatnim jej wykorzystaniem. Stworzenie powtarzalnego środowiska testowego za pomocą interfejsu zdarzeń jest bardzo proste. Zostało to przedstawione na rysunku 5.9.

Takie podejście, tzn. próba odwrócenia kolejności krytycznych operacji (np. zwolnienia i użycia pamięci, inicjalizacji i wykorzystania zmiennej), może zostać wykorzystane do tworzenia ślepych testów. Programista nie musi wtedy dogłębnie analizować programu, a może po prostu sprawdzić czy nieprawidłowa sekwencja

Wątek T1

```
void js_DestroyContext(...)
{
  ...
  js_UnpinPinnedAtoms(&rt->atomState);
  c_publish_event(”after free”);
  ...
}
```

Wątek T2

```
void js_DestroyContext(...)
{
  ...
  c_wait_event(”after free”);
  js_MarkAtomState(&rt->atomState, ...);
  ...
}
```

Rysunek 5.9: Powtarzalny test dla błędu z rysunku 1.8.

istotnych operacji może wystąpić. Oczywiście nie ma gwarancji, że dodanie wyłącznie jednego zdarzenia nazwanego do programu zagwarantuje powtarzalne ujawnianie się każdego błędu naruszenia kolejności, ale w znacznej części przypadków tak właśnie będzie (zob. podrozdział 1.2.5).

Na zakończenie, dla drugiej poprawki do omawianej usterki przygotowano bloki umożliwiające dokładne jej przetestowanie. Wykorzystano najprostszą strategię doboru wielkości pojedynczego bloku — po jednym dla każdej operacji na zmiennych współdzielonych. Wynikowy kod przedstawiono na rysunku 5.10.

Wątek T1

```
void js_DestroyContext(...)
{
  ...
  c_begin_block(”state set”);
  state = LANDING;
  c_end_block();

  c_begin_block(”gcLevel wait”);
  while (gcLevel > 0);
  c_end_block();

  c_begin_block(”free”);
  js_UnpinPinnedAtoms(&rt->atomState);
  c_end_block();
  ...
}
```

Wątek T2

```
void js_DestroyContext(...)
{
  ...
  c_begin_block(”gcLevel set”);
  gcLevel = 1;
  c_end_block();

  if (c_cond_block(state == LANDING, ”state check”))
  {
    c_begin_block(”gcLevel unset”);
    gcLevel = 0;
    c_end_block();
    return;
  }

  c_begin_block(”use”);
  js_MarkAtomState(&rt->atomState, ...);
  c_end_block();
  gcLevel = 0;
  ...
}
```

Rysunek 5.10: Podział kodu poprawki błędu z rysunku 1.8 na bloki umożliwiające dokładne przetestowanie.

Wyodrębnienie aż siedmiu bloków może sprawiać wrażenie, że testowanie będzie bardzo czasochłonne. Warto jednak pamiętać, że interesujące są tylko porządki, w których po bloku ”`free`” następuje (niekoniecznie bezpośrednio) blok ”`use`”, co istotnie zmniejsza rozmiar przestrzeni przeplotów. Dodatkowo warto zauważyć, że blok ”`gcLevel unset`” wystąpi jedynie, gdy wcześniej ”`state check`” wystąpił po ”`state set`”. Istnieją zatem proste sposoby na zmniejszenie liczby przeplotów do przetestowania, czyniąc tę metodę atrakcyjnym sposobem weryfikacji poprawek.

Podczas badań sprawdzono możliwość stworzenia powtarzalnych testów ujawniających usterkę dla wszystkich błędów zaprezentowanych w rozdziałach 1 i 2, dla których przytoczono kod źródłowy. Spośród łącznie szesnastu błędów, w przypadku dziewięciu dało się z łatwością uzyskać właściwe testy. Dla czterech usterek było to możliwe tylko przy dodatkowym wsparciu (ew. instrumentacji) ze strony systemu

operacyjnego, ponieważ błędy te wynikały z niewłaściwego wykorzystania wywołań systemowych. Opisywana w niniejszym rozdziale metoda okazała się bezużyteczna w przypadku trzech błędów, ze względu na to, że przyczyna usterki występowała we wspólnej dla wszystkich wątków ścieżce wykonania. Jeśli wątki wykonują identyczny kod, to praktycznie nie istnieje możliwość modelowania przeplotów na poziomie kodu źródłowego, gdyż jest on nierozróżnialny. Otrzymane wyniki, a także zaistniałe problemy rozważono dokładniej w podrozdziale 5.5.

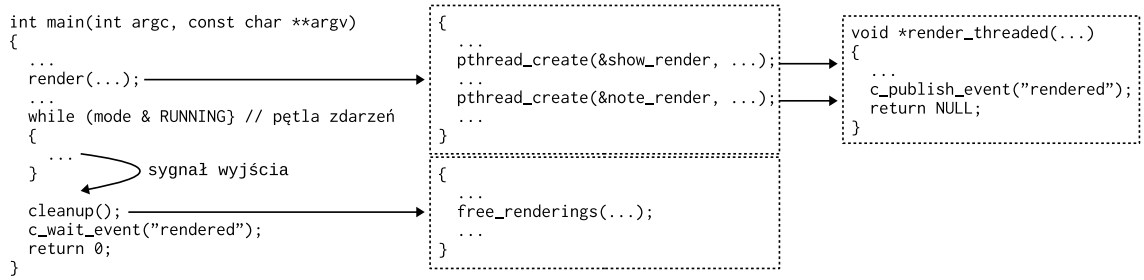
5.4.3. Przykładowe wdrożenie: projekt Slider

Testy na wcześniej znanych, dobrze opisanych błędach nie pokazują w pełni użyteczności narzędzia *Coconut*. Z tego powodu postanowiono omawiane rozwiązanie wdrożyć do projektu typu *open source*, który rozwijany jest przez otwartą społeczność programistów. Do tego testu poszukiwano programu według następujących kryteriów:
— ogólnodostępny kod źródłowy,
— rozwijany w ciągu ostatniego miesiąca,
— niewielki (do 10000 linii kodu),
— wielowątkowy,
— napisany w *C* lub *C++*,
— kod źródłowy nieznany wcześniej autorowi niniejszej pracy.
Do znalezienia projektu spełniającego wyżej opisane wymagania wykorzystano wyszukiwarkę popularnego serwisu *GitHub* [34], który udostępnia miejsce dla rozwiązań *open source*. Na drodze losowania wybrano projekt *Slider* [36], który służy do wyświetlania prezentacji zapisanych w formacie *PDF* w sposób wspomagający prezentującego.

W ramach niniejszego studium przypadku postanowiono przeprowadzić wstępne rozpoznanie kodu źródłowego aplikacji, zidentyfikować miejsca potencjalnie błędne, a następnie przetestować je za pomocą *Coconut*. W przypadku znalezienia problemów zdecydowano o wsparciu twórcy w ich rozwiązaniu.

Podczas przeglądu kodu źródłowego zauważono, że poza wątkiem głównym tworzone są dwa dodatkowe wątki renderujące slajdy oraz notatki prezentera, które wykorzystują pamięć alokowaną dynamicznie. Dalsza analiza pokazała, że pamięć ta zwalniana jest w wątku głównym przed zamknięciem programu. Postanowiono więc sprawdzić, czy możliwa jest sytuacja, w której pamięć zostanie zwolniona w trakcie działania wątków, co mogłoby doprowadzić do groźnych błędów bezpieczeństwa typu *use after free*. W tym celu stworzono ślepy test, w którym po zwolnieniu pamięci zatrzymywano wątek główny w oczekiwaniu na zakończenie wątków renderujących. Zarys kodu testu przedstawiono na rysunku 5.11.

Podczas testowania wykorzystywano duże pliki *PDF*, których czas renderowania był odpowiedni dłuższy. W trakcie procesu renderowania żądano wyłączenia aplikacji, aby uruchomić kod zwalniający pamięć. Zgodnie z oczekiwaniami prowadziło to do błędów *use after free*, które objawiały się otrzymaniem sygnału *SEGV* i zamknięciem aplikacji. Dalsza analiza kodu źródłowego pozwoliła odkryć bezpośrednią przyczynę usterki — autor kodu projektu *Slider* nie przerywał i nie oczekiwał na zakończenie pracy wątków, gdy użytkownik żądał wczesnego zakończenia aplikacji. Zgodnie z założeniami złożono raport o odkrytym błędzie [35]. Do zgłoszenia dołączono sugestię poprawki — wprowadzenie flagi przerywającej i oczekiwanie na zakończenie pracy wątków za pomocą funkcji `pthread_join()`.



Rysunek 5.11: Test ujawniający błąd *use after free* przy przedwczesnym wyjściu z aplikacji *Slider*.

Po zgłoszeniu błędu, opiekun projektu nie był w stanie odtworzyć błędu. Operacja zwalniania pamięci następowała zwykle na chwilę przed całkowitym zamknięciem aplikacji, a więc bardzo często zdarzało się, że błąd nie ujawniał się i aplikacja zostawała zamknięta poprawnie. Podano więc kroki, które pozwoliły w sposób powtarzalny obserwować przebieg błędu. Raport usterki spowodował, że wkrótce do projektu dodana została poprawka eliminująca opisywany błąd. Niestety, wprowadzona łata wykorzystała inny niż sugerowany, niepoprawny mechanizm obsługi przedwczesnego zakończenia, co doprowadziło do powstania poważniejszego błędu — zakleszczenia przy poprawnym zamknięciu.

Autor bardzo szybko cofnął zastosowaną poprawkę i zaproponował inną, opierającą się na funkcji `pthread_cancel()` i `pthread_join()`. Wykorzystanie `pthread_cancel()` należy uznać za bardzo ryzykowne działanie, gdyż skutek wywołania tej procedury jest trudny do przewidzenia — w zależności od implementacji, a także od ustawień, anulacja wywołana za pomocą `pthread_cancel()` może wystąpić w zupełnie nieoczekiwanym miejscu. Postanowiono więc po raz kolejny wykorzystać *Coconut* do przeprowadzenia testów.

Zauważono, że wewnątrz funkcji renderującej występują lokalne alokacje i dealokacje pamięci. Pojawiło się zatem podejrzenie, że jeśli przerwanie wykonania nastąpi po alokacji, ale przed zwolnieniem, to dojdzie do wycieku pamięci. Zaaranżowano więc odpowiednie zdarzenia i uruchomiono *Slider* pod nadzorem aplikacji *Valgrind*. Ponownie zgodnie ze spostrzeżeniami błąd wystąpił — przerwanie mogło powodować wycieki pamięci. Przy okazji badania tego problemu zaobserwowano również, że jeśli przerwanie wystąpi wewnątrz niektórych funkcji bibliotecznych, to użyte tam zamki mogą pozostać w stanie niespójnym, co doprowadza do zakleszczenia przy ponownym ich wywołaniu. Oba problemy zostały ponownie zgłoszone opiekunowi projektu *Slider*.

Wdrożenie *Coconut* do aplikacji *Slider* zakończyło się pełnym sukcesem. Wykorzystując funkcjonalność omawianej biblioteki, w projekcie odnaleziono trzy istotne błędy i co ważne, podano sposób ich odtworzenia. Oprócz opisanych wyżej testów, które wykazały istnienie pewnych usterek, przeprowadzono również szereg eksperymentów, zarówno za pomocą interfejsu zdarzeń jak i bloków, które pozwoliły uzyskać pewność, że konkretne fragmenty kodu nie zawierają błędów wynikających z naruszenia kolejności lub niepodzielności.

Przy okazji opisywanego wdrożenia na uwagę zasługuje również łatwość integracji biblioteki *Coconut* z projektem *Slider*. Rozpoczęcie testowania wymagało wyłącznie zmodyfikowania jednej linijki w pliku *Makefile*, a także dodania odpowiedniego

nagłówka do plików z kodem źródłowym. Po tych prostych zabiegach możliwe było swobodne wykorzystywanie biblioteki w testowanym projekcie.

5.5. Ocena narzędzia

Po dokonaniu wnikliwych badań możliwe jest podjęcie próby rzetelnej oceny przedstawionego narzędzia i wykorzystanej techniki testowania. W pierwszej części niniejszej sekcji zweryfikowano stopień realizacji wymagań projektowych. Następnie podsumowano wyniki wykonanych testów, rozważając przy tym użyteczność biblioteki *Coconut*. W kolejnej części rozważono problemy, z którymi spotkano się podczas użytkowania narzędzia, a także zaproponowano sposoby ich rozwiązania. Na koniec przedstawiono perspektywy rozwoju biblioteki *Coconut*, a kluczowe wnioski zebrano w podsumowaniu.

5.5.1. Ocena realizacji wymagań

Najważniejszym celem, który został zdefiniowany jeszcze przed implementacją biblioteki *Coconut*, było stworzenie rozwiązania, które umożliwi wygodne i rzetelne testowanie programów współbieżnych. Cechy te mają charakter jakościowy i dlatego bardzo trudno określić stopień ich realizacji. Wymagania projektowe dla rozważanego przedsięwzięcia miały zatem za zadanie wyznaczyć najistotniejsze formalne i weryfikowalne składowe osiągnięcia celu głównego. Z tego powodu w całościowej ocenie uzyskanego narzędzia nie może zabraknąć analizy realizacji sformułowanych założeń projektowych.

Wymagania funkcjonalne

Narzędzie *Coconut* umożliwia modelowanie dowolnych przeplotów wątków wybranych przez użytkownika (WF–1). W tym celu udostępnia dwa różne interfejsy — zdarzeń i bloków, które nieco różnią się w wykorzystaniu, pozwalają jednak wpływać na pracę planisty i ograniczać współbieżność. Niestety, nie udało się stworzyć mechanizmów, które umożliwiałyby definiowanie przeplotów w wątkach wykorzystujących te same ścieżki wykonania. Problem ten rozważono dokładniej w podrozdziale 5.5.3.

Za pomocą interfejsu bloków, omawiane rozwiązanie pozwala tworzyć testy różnych przeplotów (WF–2). Użytkownik narzędzia po rozmieszczeniu nazwanych bloków w kodzie jest w stanie z łatwością zdefiniować interesujące go przeploty do sprawdzenia. Takie podejście umożliwia tworzenie automatycznych testów jednostkowych kodu współbieżnego.

Narzędzie *Coconut* implementuje zaawansowany mechanizm wykrywania niemożliwych przeplotów (WF–3). Zastosowane podejście ma charakter heurystyczny, ale przy odpowiednim skonfigurowaniu (zarejestrowaniu wszystkich wątków i ustawieniu cyklu wątku czuwającego) nadaje się do użycia w każdym projekcie.

Interfejs biblioteki *Coconut* umożliwia sprawdzanie podstawowych warunków i założeń, a zbiór funkcji realizujących to może być z łatwością rozszerzony (WF–4). Działanie *Coconut* może być tymczasowo wyłączone poprzez usunięcie funkcji inicjującej z kodu lub zdefiniowanie odpowiedniej zmiennej środowiskowej (WF–5). W przypadku kompilacji produkcyjnej udostępniony został odpowiedni symbol preprocesora, którego zdefiniowanie zamienia wszelkie wywołania biblioteczne na instrukcje puste (WF–6).

Wymagania niefunkcjonalne

Narzędzie *Coconut* zostało wykonane w sposób możliwy do kompilacji metodą zdefiniowaną w wymaganiach niefunkcjonalnych i wykorzystuje jedynie bibliotekę *pthread* (WNF–1, WNF–2). Efektem prac implementacyjnych jest biblioteka statyczna, która może być konsolidowana z praktycznie każdym projektem (WNF–3). Użycie *Coconut* nie różni się zatem niczym od wykorzystywania innych bibliotek, co umożliwia powszechne i bezproblemowe wdrożenie (WNF–5). Omawiane rozwiązanie wykorzystuje dozwoloną ingerencję w kod testowanej aplikacji (WNF–4). W podrozdziale 5.5.3 opisano jednak sposób, który niweluje tę konieczność.

5.5.2. Ocena użyteczności narzędzia

Obiektywna ocena użyteczności biblioteki *Coconut* jest zadaniem bardzo trudnym. Przydatność dowolnego narzędzia można rzeczowo określić dopiero po pewnym czasie istnienia na rynku, gdy zostanie ono wdrożone w różnych przedsięwzięciach i przez różne osoby. W związku z eksperymentalnym charakterem omawianego rozwiązania nie jest to możliwe i w ocenie bazuje się jedynie na przeprowadzonych wcześniej testach.

Dla błędów omówionych w rozdziałach 1 i 2 narzędzie *Coconut* udało się zastosować w 75% przykładów. Pozostałe 25% błędów to sytuacje, w których usterka występowała we wspólnej ścieżce wykonania wątków. Ten wynik oznacza, że w wielu przypadkach omawiana biblioteka może pomóc w odszukaniu i naprawieniu błędu. Warto jednak pamiętać, że statystyka ta dotyczy sytuacji, w których źródło, przebieg i skutek błędu były znane i dobrze opisane i nie można przenieść z dużą pewnością wyniku tej analizy na pole nieodkrytych dotąd błędów.

Z tego powodu dużo bardziej miarodajne jest przykładowe wdrożenie, które przeprowadzono dla projektu *Slider*. W tej aplikacji, przy pomocy biblioteki *Coconut*, wykryto trzy poważne błędy związane ze współbieżnością i podano powtarzalny sposób na ich odtworzenie. Można zatem powiedzieć, że dla projektu *Slider* omawiane rozwiązanie okazało się użyteczne. Istnieje zatem realna szansa, że biblioteka *Coconut* znalazłaby zastosowanie również w innych przedsięwzięciach.

5.5.3. Problemy i sposoby ich rozwiązania

Jak już wielokrotnie wspomniano, narzędzie *Coconut* i koncepcja strukturalnego testowania programów współbieżnych nie są idealne — posiadają wady. Warto zatem zastanowić się na ile poważne są te niedoskonałości i czy istnieją sposoby, aby je złagodzić.

Najpoważniejszym problemem związanym z zastosowaniem przedstawionej w niniejszym rozdziale koncepcji testowania programów współbieżnych jest właściwe wybranie przeplotów do testowania. W ogólnym przypadku, dla bardzo dużego programu ciężko jest określić, jaka sekwencja wykonania wątków może być niebezpieczna. Istnieją jednak narzędzia, takie jak np. *Valgrind*, które potrafią wskazywać potencjalnie niebezpieczne operacje na danych współdzielonych. Informacje uzyskane na podstawie takiej dynamicznej analizy mogą być z powodzeniem wykorzystane do stworzenia testów w narzędziu takim jak *Coconut*. Problem ten nie jest zatem aż tak poważny, jak mogłoby się wydawać.

Narzędzie *Coconut* do wymaga ingerencji w kod testowanej aplikacji. Można uznać, że jest to rozwiązanie nieeleganckie i przez wielu programistów byłoby nie

do zaakceptowania. Ten problem można rozwiązać na kilka sposobów. Pierwsze dwa opisano przy okazji dyskusji architektury rozwiązania (zob. podrozdział 5.2.2), jednak ze względu na obecny kształt narzędzia *Coconut*, najciekawszym podejściem wydaje się być stworzenie dynamicznego wstrzykiwacza kodu wywołań bibliotecznych. Kwestię tę opisano dokładniej w kolejnym podrozdziale.

Istotną przeszkodą w powtarzalnym odtworzeniu wielu błędów okazały się wątki współdzielące ten sam kod. Odróżnienie zdarzeń i bloków w sekcjach wspólnego kodu wymaga dołączania pewnego identyfikatora wątku. Zaprojektowanie i wykorzystanie interfejsu udostępniającego taką funkcjonalność jest jednak bardzo trudne, bo często nie wiadomo przed uruchomieniem, ile wątków będzie się wykonywało i co dokładnie będą robić (np. w wielowątkowym serwerze). W związku z tym ciężko jest wskazać właściwe rozwiązanie tego problemu i być może należy poprzestać na tym, że to programista sam musi zadbać o odpowiednie przetwarzanie etykiet zdarzeń i bloków, np. poprzez konkatenację ich z identyfikatorem wątku. Podobny problem można zaobserwować, gdy testowany jest kod zawierający pętle. W tym wypadku uniwersalnym identyfikatorem dla zdarzeń i bloków może być jednak iterator, od którego wartości można uzależnić wykonanie konkretnych funkcji bibliotecznych.

Tak jak wspomniano przy okazji analizy błędów opisanych w rozdziałach 1 i 2, wiele testów, aby powtarzalnie ujawniać błędy, wymaga wsparcia ze strony systemu operacyjnego. Nie jest to jednak związane z wybraną koncepcją testowania ani z samą implementacją narzędzia. Problem ten bardzo często pojawia się także przy tworzeniu zwykłych testów sekwencyjnych. Powszechnie stosuje się wtedy różne techniki makietowania (ang. *mocking*), które z powodzeniem mogą być stosowane także dla testów programów współbieżnych.

Przyjęte założenia sprawiają, że *Coconut* testuje współbieżność widoczną z poziomu kodu w językach *C* i *C++*. Z tego powodu nie istnieje możliwość diagnozowania za pomocą tego narzędzia błędów, które wynikają ze sposobu w jaki kompilator generuje kod maszynowy. Przede wszystkim bloki i zdarzenia biblioteki *Coconut* pozwalają oddzielić co najwyżej pojedyncze instrukcje języków *C*/*C++*, które mogą generować kilka rozkazów procesora — ziarnistość jest więc ograniczona. Dodatkowo zdarza się, że wyścig na danych wprowadzany jest przez agresywne optymalizacje, które wykonuje kompilator, takie jak np. zamiana kolejności instrukcji [42]. Do diagnozowania tego typu błędów *Coconut* nie powinien być wykorzystywany, gdyż może je maskować ze względu na dodatkową synchronizację, którą wprowadza.

5.5.4. Perspektywy rozwoju

Najciekawszym kierunkiem rozwoju narzędzia *Coconut* jest zmiana sposobu, w którym wywołania biblioteczne zostają umieszczone w kodzie. Tak jak wspomniano, modyfikowanie kodu testowanego oprogramowanie jest nieeleganckie i może zostać uznane za zaśmiecanie projektu. Można byłoby zatem wykorzystać debugger, który w trakcie uruchomienia modyfikowałby wykonanie wstrzykując odpowiednie wywołania funkcji. Alternatywną do tego rozwiązania, mniej inwazyjną, lecz nieco trudniejszą w implementacji metodą jest modyfikacja binarnej postaci kodu. Niezależnie od wybranego sposobu realizacji konieczne byłoby stworzenia języka, który definiowałby punkty, w których należałoby umieścić wywołania odpowiednich funkcji. Taki język mógłby pozwalać na tworzenie testów w sposób bardziej deklaratywny — prostszy dla programisty i bardziej odporny na pomyłki.

Dodatkowym wzbogaceniem tej koncepcji mógłby być edytor graficzny, który pozwalałaby testerowi w prosty sposób tworzyć testy. Aplikacja ta operowałaby na kodzie źródłowym i pozwalała m.in. na oznaczanie bloków kodu, tworzenie diagramów strzałkowych interesujących przeplotów, a także dodawanie odpowiednich założeń i uruchamianie konkretnych testów. W ten sposób programista całkowicie zostałby odizolowany od konieczności obcowania z surową biblioteką, co dodatkowo ułatwiłoby wykorzystanie opisywanego narzędzia.

Ciekawym zagadnieniem w kwestii rozwoju biblioteki *Coconut* jest integracja z innymi narzędziami. W szczególności atrakcyjna wydaje się możliwość współpracy z rozwiązaniami takimi jak *Valgrind* czy *ThreadSanitizer*, które sugerują miejsca prawdopodobnych wyścigów. Warto również rozważyć zintegrowanie *Coconut* z innymi bibliotekami oferującymi testowanie jednostkowe tak, aby rozwiązanie to stało się standardowym elementem zestawu narzędzi programisty *C/C++*.

Omawiając perspektywy rozwoju narzędzia *Coconut* warto zauważyć, że podobne rozwiązanie da się zaimplementować dla każdego języka programowania, który wykorzystuje jawne blokowanie, czyli dla praktycznie każdego języka imperatywnego. Wiele popularnych platform umożliwia także tworzenie opakowań (ang. *wrapper*) dla kodu napisanego w *C*, a więc istnieje możliwość bezpośredniego przeniesienia biblioteki *Coconut* do innych języków programowania.

5.6. Podsumowanie

Zarówno biblioteka *Coconut* jak i sama koncepcja strukturalnego testowania programów współbieżnych poprzez deterministyczną weryfikację konkretnych przeplotów wydają się być interesującymi zagadnieniami w kwestii narzędzi wspierających programowanie współbieżne. Jak pokazały testy przeprowadzone na przykładach prezentowanych w rozdziałach 1 i 2, takie podejście może pomóc w wykryciu, wyizolowaniu i poprawieniu znacznej liczby błędów. Przykładowe wdrożenie w projekcie *Slider* uwidacznia natomiast fakt, że narzędzie *Coconut* może zostać z łatwością wdrożone do istniejących projektów i może wydatnie ułatwić wyszukiwanie nowych, nieznanych wcześniej błędów. Biblioteka *Coconut* nie jest rozwiązaniem wolnym od problemów, ale bogate, atrakcyjne perspektywy rozwoju pozwalają wyeliminować większość z nich.

Na podstawie powyższego podsumowania można zatem stwierdzić, że przystępując do testowania aplikacji współbieżnych warto rozważyć możliwość zastosowania techniki deterministycznego weryfikowania konkretnych przeplotów, która została zrealizowana w bibliotece *Coconut*. Sama idea jak i implementacja oferowana przez wspomniane narzędzie warte są natomiast dalszych badań i ciągłego rozwoju.

Podsumowanie

W niniejszej pracy dogłębnie rozważono zagadnienia niezawodności i bezpieczeństwa w programowaniu współbieżnym. Pierwszym elementem badania była analiza błędów w powszechnie wykorzystywanych aplikacjach współbieżnych. Potwierdziła ona, że współbieżność jest zagadnieniem niezwykle trudnym, zarówno ze względów społecznych, jak i technologicznych. Programiści tworzący systemy współbieżne mają trudności z przystosowaniem się do dynamicznego rozwoju tej dziedziny, a wykorzystywane obecnie techniki programowania są często nieadekwatne do rozwiązywanych problemów. Skutkuje to tym, że programy współbieżne są nierzadko zawodniejsze od programów sekwencyjnych.

Kolejną rozważaną kwestią było bezpieczeństwo aplikacji współbieżnych. Badanie znanych podatności w oprogramowaniu pokazało, że niewłaściwa obsługa współbieżności może powodować poważne błędy bezpieczeństwa. Dokładne rozpoznanie wspomnianych podatności i porównanie z typowymi usterkami pokazało z kolei, że potencjalnie każdy błąd we współbieżności może stanowić zagrożenie dla bezpieczeństwa systemu informatycznego. Wszystkie błędy we współbieżności należy zatem traktować z należytą ostrożnością. W ramach tego badania zaobserwowano również, że znane z programowania sekwencyjnego techniki wykrywania podatności jak i obrony przed atakami są istotnie osłabione w aplikacjach współbieżnych.

Następnie, w ramach poszukiwań rozwiązania zaobserwowanych problemów technologicznych, dokonano przeglądu nowoczesnych technik programowania współbieżnego. Analiza obejmowała szerokie spektrum propozycji — od najlepszych praktyk poprzez zaawansowane mechanizmy aż do wyrafinowanych języków programowania. Podczas rozpoznania zauważono, że wszystkie te dziedziny rozwijają się niezwykle dynamicznie, ale nie dotyczy to narzędzi wspierających programowanie współbieżne. W celu poznania przyczyn takiego stanu rzeczy, dokładnie zbadano istniejące rozwiązania w tym obszarze. Podczas analizy zidentyfikowano konkretne zapotrzebowania na nowe narzędzia.

W związku z powyższym, w ramach praktycznej części pracowni, opracowano koncepcyjnie, zaprojektowano i zaimplementowano innowacyjne narzędzie do testowania i próbnego uruchamiania programów współbieżnych. Rozwiązanie to opiera się na deterministycznej weryfikacji przeplotów, co zapewnia powtarzalność i ułatwia obserwowanie błędów. W celu sprawdzenia skuteczności narzędzia zastosowano je do błędów omówionych w pierwszej części pracy. Wykorzystanie rozwiązania w tym przypadku pozwoliło w powtarzalny sposób obserwować ujawnianie się 75% błędów. Przydatność narzędzia potwierdziło przykładowe wdrożenie w wybranym projekcie z otwartym kodem źródłowym. Za pomocą opisywanego rozwiązania udało się odnaleźć i odtworzyć trzy istotne błędy we współbieżności, a także stworzyć testy poprawnościowe znacznej części projektu.

Wykorzystana w narzędziu koncepcja deterministycznej weryfikacji przeplotów jest atrakcyjnym tematem dalszych badań. W pracy zaprezentowano możliwe kierunki rozwoju przedstawionego rozwiązania, które mogłyby zwiększyć jego skuteczność i ułatwić masowe wdrożenie. Dalsze prace nad narzędziem tego typu mogłyby doprowadzić do stworzenia interesującego, współbieżnego odpowiednika testów jednostkowych, które są powszechnym i sprawdzonym sposobem testowania programów sekwencyjnych. Jest to działanie warte rozważenia, gdyż przyspieszenie rozwoju narzędzi dla współbieżności mogłoby uczynić programowanie procesem łatwiejszym, a co za tym idzie, programy mogłyby stać się bezpieczniejsze i niezawodniejsze.

A. API biblioteki Coconut

W niniejszym dodatku przedstawiono kompletne *API* narzędzia *Coconut*. Funkcje, makra, zmienne środowiskowo i symbole preprocesora podzielono na kategorie dotyczące: uruchamiania, różnych interfejsów instrumentacji, a także wyświetlania i sprawdzania założeń.

A.1. Uruchamianie

`void c_init()` — funkcja inicjalizująca. Powinna być wywołana przed jakąkolwiek inna funkcją biblioteki *Coconut*, w przeciwnym razie wszystkie inne wywołania będą ignorowane.

`void c_free()` — funkcja zwalniająca zasoby. Powinna być wywołana jako ostatnia funkcja biblioteki *Coconut*.

`void c_set_watchdog_tick(unsigned int tick)` — funkcja ustawiająca długość cyklu wątku typu *watchdog* na `tick` sekund. Ma wyższy priorytet od zmiennej środowiskowej `C_SET_WATCHDOG_TICK`.

`C_SET_WATCHDOG_TICK` — zmienna środowiskowa wyrażająca długość cyklu wątku typu *watchdog* w sekundach. Ma niższy priorytet niż funkcja `c_set_watchdog_tick()`.

`C_DISABLE` — zmienna środowiskowa, która w przypadku wartości różnej od `0` wyłącza całkowicie działanie biblioteki.

`NCOCONUT` — symbol preprocesora, którego zdefiniowanie podczas kompilacji całkowicie usuwa mechanizm z kodu.

A.2. Interfejs zdarzeń nazwanych

`void c_wait_event(const char *event)` — funkcja blokująca wywołujący wątek do momentu opublikowania zdarzenia o nazwie `event`. Jeśli zdarzenie zostało opublikowane przed wywołaniem tej funkcji, to wątek nie zostaje zablokowany.

`void c_publish_event(const char *event)` — funkcja publikująca zdarzenie o nazwie `event`. Odblokowuje wszystkie wątki oczekujące na zdarzenie `event`.

`bool c_is_event_published(const char *event)` — funkcja sprawdzająca, czy zdarzenie `event` zostało opublikowane. Zwraca wartość `true` jeśli publikacja miała miejsce, `false` w przeciwnym przypadku.

A.3. Interfejs bloków nazwanych

`void c_set_blocks_interleaving(const char *interleaving)` — funkcja ustalająca porządek bloków. Nazwy bloków do wykonania sekwencyjnego powinny być oddzielone za pomocą `';'`. Bloki oddzielone za pomocą `','` będą uruchomione równolegle.

`void c_begin_block(const char *block)` — funkcja rozpoczynająca blok o nazwie `block` (nazwa nie powinna zawierać `','` ani `';'`). Powoduje zablokowanie wątku wywołującego aż do momentu, gdy wszystkie bloki poprzedzające zakończą się.

`void c_end_block()` — funkcja kończąca ostatnio rozpoczęty blok. Informuje następników o zakończeniu danego bloku.

`bool c_is_{before, during, after}_block(const char *block)` — funkcje sprawdzające, czy blok o nazwie `block` jest aktualnie rozpoczęty/w trakcie wykonania/zakończony.

`c_cond_block(COND, BLOCK)` — makro pozwalające stworzyć blok o nazwie `BLOCK` otaczający sprawdzenie warunku logicznego `COND`. Zwraca wartość ewaluacji wyrażenia logicznego `COND`. Makro przydatne do tworzenia bloków wewnątrz wyrażeń `if (...)`. Punkty sekwencyjne gwarantują właściwą kolejność wykonania i jednokrotną ewaluację `COND`.

A.4. Wyświetlanie i sprawdzanie założeń

`void c_output(const char *format, ...)` — funkcja wyświetlająca w synchronizowany sposób wiadomość `format` z argumentami (analogicznie jak dla funkcji `printf`) na standardowym wyjściu błędów.

`c_out(FMT, ...)` — makro wyświetlające za pomocą `c_output()` wiadomość `FMT` z argumentami, która poprzedzona jest informacjami o pliku, funkcji i linii w której makro zostało wykorzystane.

`c_assert_{true, false}(COND, FMT, ...)` — makra sprawdzające prawdziwość/fałszywość warunku `COND`. W przypadku niespełnienia założenia wyświetlona zostaje wiadomość `FMT` z argumentami w sposób analogiczny do tego z `c_out()`.

`c_assert_{before, after}_event(EVENT, FMT, ...)` — makra sprawdzające wystąpienie/niewystąpienie zdarzenia `EVENT`. W przypadku niespełnienia założenia wyświetlona zostaje wiadomość `FMT` z argumentami w sposób analogiczny do tego z `c_out()`.

`c_assert_{before, after, during}_block(BLOCK, FMT, ...)` — makra sprawdzające czy blok o nazwie `BLOCK` jest aktualnie rozpoczęty/w trakcie wykonania/zakończony. W przypadku niespełnienia założenia wyświetlona zostaje wiadomość `FMT` z argumentami w sposób analogiczny do tego z `c_out()`.

B. Instrukcja obsługi biblioteki Coconut

W niniejszym dodatku przedstawiono kompletną instrukcję obsługi — od ściągnięcia kodu źródłowego poprzez kompilację aż do prostego przykładu użycia.

B.1. Ściągnięcie i kompilacja

Ściągnięcie i kompilacja odbywają się za pomocą standardowych narzędzi programistycznych *git* i *make*.

```
$ git clone git://github.com/luksow/Coconut.git
$ cd Coconut/src/
$ make
```

Wydruk B.1: Sekwencja poleceń wymagana do ściągnięcia i skompilowania narzędzia *Coconut*.

W wyniku tych działań narzędzie *Coconut* jest gotowe do działania. Do instrumentacji wykonania wykorzystywane są dwa pliki: biblioteka statyczna `libcoconut.a`, która powstała w wyniku kompilacji oraz plik nagłówkowy `coconut_pub.h` zawierający odpowiednie deklaracje.

B.2. Przykład użycia

Do demonstracji użycia wykorzystany został przykładowy program `extended_events.cpp` z katalogu `examples`, którego działanie zostało szczegółowo omówione w podrozdziale 5.4.1.

```
#include <thread>
#include <queue>
#include <iostream>

#include "coconut.h"

struct Request
{
};

bool invalidated;

std::queue<Request> requests;

void processRequest(Request& r)
{
}

void invalidateRequests()
```

```
20 {
21   c_wait_event("checked");
22   invalidated = true;
23
24   while (!requests.empty())
25     requests.pop();
26   c_publish_event("cleared");
27 }
28
29 void processFirstRequest()
30 {
31   if (invalidated || requests.empty())
32     return;
33   c_publish_event("checked");
34
35   c_wait_event("cleared");
36   c_assert_false(requests.empty(), "requests queue is empty!");
37   processRequest(requests.front());
38   requests.pop();
39 }
40
41 int main()
42 {
43   c_init();
44
45   invalidated = false;
46   requests.push(Request());
47   auto t1 = new std::thread(invalidateRequests);
48   auto t2 = new std::thread(processFirstRequest);
49   t1->join();
50   t2->join();
51   delete t1;
52   delete t2;
53
54   c_free();
55   return 0;
56 }
```

Wydruk B.2: Kompletny program demonstrujący użycie interfejsu zdarzeń nazwanych biblioteki *Coconut*.

Do skompilowania powyższego programu potrzebna będzie skompilowana biblioteka *Coconut* oraz jej plik nagłówkowy. Kompilacja i uruchamianie przebiegają w standardowy sposób.

```
1 $ cd ../examples
2 $ cp ../src/libcoconut.a ./
3 $ cp ../src/coconut_pub.h ./coconut.h # uproszczona nazwa
4 $ g++ --std=c++11 -lpthread extended_events.cpp libcoconut.a
5 $ ./a.out
6 [extended_events.cpp:void processFirstRequest():36]: Assert failed: requests queue is emp
```

Wydruk B.3: Kompilacja i uruchamianie programu demonstracyjnego przy pomocy biblioteki *Coconut*.

Po uruchomieniu, zgodnie z przewidywaniami, program demonstracyjny narusza zdefiniowane założenie, o czym informuje odpowiedni komunikat.

Bibliografia

[1] R. P. Abbot, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, D. A. Webb. Security analysis and enhancements of computer operating systems. Raport instytutowy, April 1976.

[2] Ada Resource Association. *Ada Resource Association Home*. Dostęp internetowy: http://www.adaic.org/ [02.06.2013].

[3] Adapteva. *Adapteva Home*. Dostęp internetowy: http://www.adapteva.com/ [02.06.2013].

[4] Tony Adrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, Yichen Xie. Zing: A Model Checker for Concurrent Software. Raport instytutowy, 2004. Dostęp internetowy: http://research.microsoft.com/en-us/projects/zing/tool.pdf [02.06.2013].

[5] Ross Anderson. *Security Engineering. A Guide to Building Dependable Distributed Systems. Second Edition*. Wiley Publishing, Inc., 2008.

[6] The Apache Software Foundation. *Apache OpenOffice Bugzilla*. Dostęp internetowy: https://issues.apache.org/ooo/ [02.06.2013].

[7] The Apache Software Foundation. *ASF Bugzilla*. Dostęp internetowy: https://issues.apache.org/bugzilla/ [02.06.2013].

[8] Boris Beizer. *Software Testing Techniques, 2nd Edition*. International Thomson Computer Press, 1990.

[9] Bell Laboratories, Lucent Technologies. *Verisoft Home*. Dostęp internetowy: http://cm.bell-labs.com/who/god/verisoft/ [02.06.2013].

[10] Bell Labs. *Spin Home*. Dostęp internetowy: http://spinroot.com/ [02.06.2013].

[11] Mordechai Ben-Ari. *Podstawy programowania współbieżnego i rozproszonego*. Wydawnictwa Naukowo–Techniczne, Warszawa, 2009.

[12] Richard Bisbey, Dennis Hollingworth. Protection Analysis: Final Report. Raport instytutowy, May 1978.

[13] Chandrasekhar Boyapati, Robert Lee, Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2002. Dostęp internetowy: http://www.st.informatik.tu-darmstadt.de/~ostermann/oopsla02.pdf [02.06.2013].

[14] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, Shmuel Ur. Applications of Synchronization Coverage. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, June 2005. Dostęp internetowy: http://pdf.aminer.org/000/548/312/applications_of_synchronization_coverage.pdf [02.06.2013].

[15] John Carmack. Static Code Analysis. *#AltDevBlogADay*, December 2011. Dostęp internetowy: http://www.altdevblogaday.com/2011/12/24/static-code-analysis/ [02.06.2013].

[16] CheckThread. *CheckThread Home*. Dostęp internetowy: http://www.checkthread.org/example-racecondition.html [02.06.2013].

[17] Kees Cook. *fs: add link restrictions*. Dostęp internetowy: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=800179c9b8a1e796e441674776d11cd4c05d61d7 [02.06.2013].

[18] Kees Cook. Link restrictions released in Linux 3.6. *codeblog: code is freedom — patching my itch*, October 2012. Dostęp internetowy: http://www.outflux.net/blog/ [02.06.2013].

[19] Jonathan Corbet. The kernel lock validator. *LWN.net*, May 2006. Dostęp internetowy: http://lwn.net/Articles/185666/ [02.06.2013].

[20] Jonathan Corbet. User–space lockdep. *LWN.net*, February 2013. Dostęp internetowy: http://lwn.net/Articles/536363/ [02.06.2013].

[21] Coverity Development Testing. *Coverity Home*. Dostęp internetowy: http://www.coverity.com/ [02.06.2013].

[22] Russ Cox. Bell labs and csp threads. Dostęp internetowy: http://swtch.com/~rsc/thread/ [02.06.2013].

[23] CVE Details. *CVE Details: The ultimate security vulnerability datasource*. Dostęp internetowy: http://www.cvedetails.com/ [02.06.2013].

[24] Roman Dementiev. Exploring Intel Transactional Synchronization Extensions with Intel Software Development Emulator. June 2012. Dostęp internetowy: http://software.intel.com/en-us/blogs/2012/11/06/exploring-intel-transactional-synchronization-extensions-with-intel-software [02.06.2013].

[25] The Economist. Parallel bars. June 2011. Dostęp internetowy: http://www.economist.com/node/18750706 [02.06.2013].

[26] Dawson Engler, Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003. Dostęp internetowy: http://ftp.cs.rochester.edu/meetings/sosp2003/papers/p200-engler.pdf [02.06.2013].

[27] Chris Evans. Very interesting traceroute flaw. 2000. Dostęp internetowy: http://lwn.net/2000/1012/a/traceroute.php3 [02.06.2013].

[28] Eitan Farchi, Yarden Nir, Shmuel Ur. Concurrent Bug Patterns and How to Test Them. IBM Research Lab in Haifa, Haifa University Campus, Mount Carmel, Haifa 31905, Israel. Dostęp internetowy: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1213511 [02.06.2013].

[29] Cormac Flanagan, Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004. Dostęp internetowy: http://www.cs.ucsc.edu/~cormac/papers/popl04.ps [02.06.2013].

[30] A Serializability Violation Detector for Shared-Memory Server Programs. Min Xu and Rastislav Bodík and Mark D. Hill. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language Design and Implementation*, 2005. Dostęp internetowy: http://www.cs.berkeley.edu/~bodik/research/pldi05-svd.pdf [02.06.2013].

[31] Free Software Foundation. *GDB: The GNU Project Debugger*. Dostęp internetowy: http://www.gnu.org/software/gdb/ [02.06.2013].

[32] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*. Helion, Gliwice, 2010.

[33] Gimpel Software. *PC–Lint Home*. Dostęp internetowy: http://www.gimpel.com/html/pcl.htm [02.06.2013].

[34] GitHub Inc. *GitHub Home*. Dostęp internetowy: https://github.com/ [02.06.2013].

[35] GitHub Inc. *Slider issues*. Dostęp internetowy: https://github.com/TrilbyWhite/Slider/issues/7 [02.06.2013].

[36] GitHub Inc. *Slider repository*. Dostęp internetowy: https://github.com/TrilbyWhite/Slider [02.06.2013].

[37] Patrice Godefroid. Model checking for programming languages using VeriSoft. *Proceedings of the 24th ACM SIGPLAN–SIGACT symposium on Principles of programming languages*, 1997. Dostęp internetowy: http://cm.bell-labs.com/who/god/verisoft/popl97.ps [02.06.2013].

[38] Brian Goetz. Double–checked locking: Clever, but broken. *JavaWorld.com*, September 2001. Dostęp internetowy: http://www.javaworld.com/jw-02-2001/jw-0209-double.html [02.06.2013].

[39] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. *Java Concurrency in Practice*. Addison Wesley, 2006.

[40] Google Inc. *The Go Programming Language*. Dostęp internetowy: http://golang.org/ [02.06.2013].

[41] Google Inc. *The Go Programming Language FAQ*. Dostęp internetowy: http://golang.org/doc/faq#csp [02.06.2013].

[42] Google Inc. *ThreadSanitizer, a data race detector for C/C++ and Go*. Dostęp internetowy: https://code.google.com/p/thread-sanitizer/ [02.06.2013].

[43] Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. Composable Memory Transactions. Microsoft Research, Cambridge, August 2006. Dostęp internetowy: http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/stm.pdf [02.06.2013].

[44] Klaus Havelund, Thomas Pressburger. Model checking java programs using java pathfinder, 1998. Dostęp internetowy: http://www.havelund.com/Publications/jpf-sttt.ps.Z [02.06.2013].

[45] Maurice Herlihy, J. Eliot B. Moss. Transactional memory: architectural support for lock–free data structures. *Proceedings of the 20th annual international symposium on computer architecture*, 1993. Dostęp internetowy: http://cs.brown.edu/~mph/HerlihyM93/herlihy93transactional.pdf [02.06.2013].

[46] Carl Hewitt, Peter Bishop, Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence, 1973. Dostęp internetowy: http://ijcai.org/Past%20Proceedings/IJCAI-73/PDF/027B.pdf [02.06.2013].

[47] C.A.R. Hoare. Communicating Sequential Processes, 2004. Dostęp internetowy: http://www.usingcsp.com/cspbook.pdf [02.06.2013].

[48] Luke Hutchison. The multicore dilemma (in the big data era) is worse than you think. Dostęp internetowy: http://www.flowlang.net/2012/05/multicore-dilemma-in-big-data-era-is.html [02.06.2013].

[49] Luke Hutchison. The Flow Programming Language: An implicitly–parallelizing, programmer–safe language. Dostęp internetowy: http://www.flowlang.net/ [02.06.2013].

[50] IBM Research. *ConTest Home*. Dostęp internetowy: https://www.research.ibm.com/haifa/projects/verification/contest/ [02.06.2013].

[51] Intel Corp. *Threading Building Blocks Home*. Dostęp internetowy: http://threadingbuildingblocks.org/ [02.06.2013].

[52] Andrzej Jaszkiewicz. *Inżynieria oprogramowania*. Helion, Gliwice, 1997.

[53] Simon Peyton Jones. Programming in the Age of Concurrency: Software Transactional Memory. Channel 9. Dostęp internetowy: http://channel9.msdn.com/Shows/Going+Deep/Programming-in-the-Age-of-Concurrency-Software-Transactional-Memory [02.06.2013].

[54] Simon Peyton Jones. Beautiful Concurrency. April 2012. Dostęp internetowy: https://www.fpcomplete.com/school/beautiful-concurrency [02.06.2013].

[55] Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. Concurrent Haskell. *23rd ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida*, 1996. Dostęp internetowy: http://research.microsoft.com/pubs/67077/concurrent-haskell.ps.gz [02.06.2013].

[56] Kickstarter Inc. *Parallella: A Supercomputer For Everyone*. Dostęp internetowy: http://www.kickstarter.com/projects/adapteva/parallella-a-supercomputer-for-everyone [02.06.2013].

[57] Andi Kleen. Lock elision in the GNU C library. *LWN.net*, January 2013. Dostęp internetowy: http://lwn.net/Articles/534758/ [02.06.2013].

[58] Tom Knight. An Architecture for Mostly Functional Languages. The M.I.T. Artificial Intelligence Laboratory, 1986. Dostęp internetowy: http://web.mit.edu/mmt/Public/Knight86.pdf [02.06.2013].

[59] Jeff Knupp. Python's Hardest Problem. *Hackers Gonna Hack*, March 2012. Dostęp internetowy: http://www.jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/ [02.06.2013].

[60] Yossi Kreinin. checkedthreads: bug–free shared memory parallelism. *Proper Fixation: a substitute for anaesthesia*, April 2013. Dostęp internetowy: http://www.yosefk.com/blog/checkedthreads-bug-free-shared-memory-parallelism.html [02.06.2013].

[61] Christoph Lameter. Effective Synchronization on Linux/NUMA Systems. May 2005. Dostęp internetowy: http://www.lameter.com/gelato2005.pdf [02.06.2013].

[62] The Linux Foundation. *Linux Kernel Bug Tracker*. Dostęp internetowy: https://bugzilla.kernel.org/ [02.06.2013].

[63] The Linux Foundation. *Linux Kernel Documentation*. Dostęp internetowy: http://lxr.linux.no/linux/Documentation/ [02.06.2013].

[64] The Linux Foundation. *The Linux man-pages*. Dostęp internetowy: http://www.kernel.org/doc/man-pages/ [02.06.2013].

[65] LMAX Ltd. *LMAX Disruptor: High Performance Inter-Thread Messaging Library*. Dostęp internetowy: http://lmax-exchange.github.io/disruptor/ [02.06.2013].

[66] Robert Love. Kernel Locking Techniques. *Linux Journal*, August 2002. Dostęp internetowy: http://www.linuxjournal.com/article/5833 [02.06.2013].

[67] Shan Lu, Weihang Jiang, Yuanyuan Zhou. A Study of Interleaving Coverage Criteria. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, September 2007. Dostęp internetowy: http://www.cs.wisc.edu/~shanlu/paper/fp051-lu.pdf [02.06.2013].

[68] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, Yuanyuan Zhou. MUVI: Automatically Inferring Multi–Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. *Proceedings of the Twenty–First ACM SIGOPS Symposium on Operating Systems Principles*, 2007. Dostęp internetowy: http://web.mit.edu/ralucap/www/MUVI.pdf [02.06.2013].

[69] Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. Department of Computer Science, University of Illinois at Urbana Champaign, Urbana, IL 61801. Dostęp internetowy: http://www.cs.columbia.edu/~junfeng/09fa-e6998/papers/concurrency-bugs.pdf [02.06.2013].

[70] Shan Lu, Joseph Tucek, Feng Qin, Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006. Dostęp internetowy: http://www.cs.wisc.edu/~shanlu/paper/asplos062-lu.pdf [02.06.2013].

[71] Brandon Lucia, Luis Ceze. Finding Concurrency Bugs with Context–Aware Communication Graphs. *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009. Dostęp internetowy: http://homes.cs.washington.edu/~luisceze/publications/micro09-lucia.pdf [02.06.2013].

[72] Simon Marlow. Parallel and Concurrent Programming in Haskell. Microsoft Research Ltd., Cambridge, U.K. Dostęp internetowy: http://community.haskell.org/~simonmar/par-tutorial.pdf [02.06.2013].

[73] Bill McCloskey, Feng Zhou, David Gay, Eric Brewer. Autolocker: Synchronization Inference for Atomic Sections. *Conference Record of the 33rd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 2006. Dostęp internetowy: http://www.barnowl.org/research/pubs/06-popl-autolocker.pdf [02.06.2013].

[74] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Linux Technology Center, IBM Beaverton, 2012. Dostęp internetowy: https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html [02.06.2013].

[75] Microsoft Corp. *Asynchronous Programming with Async and Await*. Dostęp internetowy: http://msdn.microsoft.com/en-us/library/vstudio/hh191443.aspx [02.06.2013].

[76] Microsoft Corp. *CHESS Home*. Dostęp internetowy: http://research.microsoft.com/en-us/projects/chess/ [02.06.2013].

[77] Microsoft Corp. *General Best Practices in the Concurrency Runtime*. Dostęp internetowy: http://msdn.microsoft.com/en-us/library/ff601929.aspx [02.06.2013].

[78] Microsoft Corp. *Parallel Programming in the .NET Framework*. Dostęp internetowy: http://msdn.microsoft.com/en-us/library/dd460693.aspx [02.06.2013].

[79] Microsoft Corp. *Zing Home*. Dostęp internetowy: http://research.microsoft.com/en-us/projects/zing/ [02.06.2013].

[80] The MITRE Corporation. *Common Vulnerabilities and Exposures: The Standard for Information Security Vulnerability Names*. Dostęp internetowy: http://cve.mitre.org/ [02.06.2013].

[81] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics Magazine 38 (8)*, April 1965. Dostęp internetowy: http://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf [02.06.2013].

[82] The Mozilla Foundation. *Mozilla Bugzilla*. Dostęp internetowy: https://bugzilla.mozilla.org/ [02.06.2013].

[83] Mozilla Foundation. *The Rust Programming Language*. Dostęp internetowy: http://www.rust-lang.org/ [02.06.2013].

[84] MTI Technology Vulnerability Research Team. *Samsung Galaxy S3 partial screen-lock bypass*. Dostęp internetowy: http://seclists.org/bugtraq/2013/Feb/120 [02.06.2013].

[85] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, April 1989. Dostęp internetowy: http://embedded.eecs.berkeley.edu/Research/hsc/class.F03/ee249/discussionpapers/PetriNets.pdf [02.06.2013].

[86] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball. CHESS: A Systematic Testing Tool for Concurrent Software. Raport instytutowy, 2007. Dostęp internetowy: http://research.microsoft.com/pubs/70509/tr-2007-149.pdf [02.06.2013].

[87] Pramod Nagaraja. Concurrency: Best Practices. *IndicThreads.com Conference On Java Technology*, 2009. Dostęp internetowy: http://j09.indicthreads.com/wp-content/uploads/2009/12/IndicThreads-Java-Concurrency_Best_Practices_Pramod.pdf [02.06.2013].

[88] Satish Narayanasamy, Gilles Pokam, Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. 2005. Dostęp internetowy: http://cseweb.ucsd.edu/~calder/papers/ISCA-05-BugNet.pdf [02.06.2013].

[89] NASA Ames Research Center. *Java PathFinder Home*. Dostęp internetowy: http://babelfish.arc.nasa.gov/trac/jpf [02.06.2013].

[90] Nicholas Nethercote, Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007. Dostęp internetowy: http://valgrind.org/docs/valgrind2007.pdf [02.06.2013].

[91] The Open Group. *The Open Group Home*. Dostęp internetowy: http://www.opengroup.org/ [02.06.2013].

[92] The Open Web Application Security Project. *Double Free Vulnerability*. Dostęp internetowy: https://www.owasp.org/index.php/Double_Free [02.06.2013].

[93] The Open Web Application Security Project. *OWASP Top Ten Project*. Dostęp internetowy: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project [02.06.2013].

[94] OpenMP Architecture Review Board. *The OpenMP API specification for parallel programming*. Dostęp internetowy: http://openmp.org/ [02.06.2013].

[95] Oracle Corporation. *MySQL Bugs*. Dostęp internetowy: http://bugs.mysql.com/ [02.06.2013].

[96] Paul Starzetz. *Linux kernel uselib() privilege elevation*. Dostęp internetowy: http://sebug.net/paper/Exploits-Archives/2005-exploits/0501-exploits/isec-0021-uselib.txt [02.06.2013].

[97] Mathias Payer, Thomas R. Gross. Protecting Applications Against TOCTTOU Races by User-Space Caching of File Metadata. *In Proceedings of the Eighth International Conference on Virtual Execution Environments (VEE '12)*, 2012. Dostęp internetowy: http://www.lst.ethz.ch/research/publications/VEE_2012/VEE_2012.pdf [02.06.2013].

[98] Milos Prvulovic, Josep Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-30)*, June 2003. Dostęp internetowy: http://iacoma.cs.uiuc.edu/iacoma-papers/reenact.pdf [02.06.2013].

[99] William Pugh, Nathaniel Ayewah. Unit Testing Concurrent Software. *IEEE/ACM International Conference on Automated Software Engineering*, November 2007. Dostęp internetowy: http://www.cs.umd.edu/projects/PL/multithreadedtc/pubs/ASE07-pugh.pdf [02.06.2013].

[100] Shaz Qadeer, Dinghao Wu. KISS: Keep It Simple and Sequential. *PLDI 2004: ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004. Dostęp internetowy: http://faculty.ist.psu.edu/wu/papers/kiss.pdf [02.06.2013].

[101] Srinivasan Raghunathan, Ashutosh Prasad, Birendra K. Mishra, Hsihui Chang. Open Source Versus Closed Source: Software Quality in Monopoly and Competitive Markets. *IEEE Transactions on Systems, Man, and Cybernetics — Part A: System and Humans, Vol. 35, No. 6*, November 2005.

[102] Rusty Russell. *Unreliable Guide To Locking*. Dostęp internetowy: https://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/ [02.06.2013].

[103] Nir Shavit, Dan Touitou. Software Transactional Memory. Tel–Aviv University. Dostęp internetowy: http://www.cse.ohio-state.edu/~agrawal/788-su08/Papers/week4/shavit95software.pdf [02.06.2013].

[104] Łukasz Sowa. Nowoczesne mechanizmy bezpieczeństwa w systemie Linux. Politechnika Warszawska, 2012.

[105] Diomidis Spinellis. A Tale of Four Kernels. Department of Management Science and Technology Athens University of Economics and Business, 2008. Dostęp internetowy: http://www.dmst.aueb.gr/dds/pubs/conf/2008-ICSE-4kernel/html/Spi08b.html [02.06.2013].

[106] Scott Stender, Alex Vidergar. Concurrency Attacks in Web Applications. iSEC Partners. Dostęp internetowy: http://www.hakim.ws/BHUSA08/speakers/Stender_Vidergar_Concurrency_Attacks/BH_US_08_Stender_Vidergar_Concurrency_Attacks_in_Web_Applications_Presentation.pdf [02.06.2013].

[107] Jan Strelau. *Psychologia. Podręcznik akademicki. Podstawy psychologii*. Gdańskie Wydawnictwo Psychologiczne, 2007.

[108] Michael Suess. Recursive locks — a blessing or a curse? *Thinking Parallel: A Blog on Parallel Programming and Concurrency by Michael Suess*, September 2006. Dostęp internetowy: http://www.thinkingparallel.com/2006/09/27/recursive-locks-a-blessing-or-a-curse/ [02.06.2013].

[109] Minyoung Sung, Soyoung Kim, Sangsoo Park, Naehyuck Chang, Heonshik Shin. Comparative performance evaluation of java threads for embedded applications: Linux thread vs. green thread. *Inf. Process. Lett.*, November 2002.

[110] Herb Sutter. C++ and Beyond 2012: Herb Sutter — C++ Concurrency. Channel 9. Dostęp internetowy: http://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Herb-Sutter-Concurrency-and-Parallelism [02.06.2013].

[111] Andrew Tanenbaum, Maarten Van Steen. *Distributed Systems: Principles and Paradigms, 2nd Edition*. Prentice Hall, 2006.

[112] Andrew S. Tanenbaum. *Systemy operacyjne. Wydanie III*. Helion, Gliwice, 2010.

[113] Richard N. Taylor, David L. Levine, Cheryl D. Kelly. Structural Testing of Concurrent Programs. *IEEE Transactions on Software Engineering, Vol. 18, No. 3*, March 1992.

[114] Dan Tsafrir, Tomer Hertz, David Wagner, Dilma Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. *In Sixth USENIX conference on File and Storage Technologies(FAST '08)*, 2008. Dostęp internetowy: http://www.cs.berkeley.edu/~daw/papers/tocttou-fast08.pdf [02.06.2013].

[115] Eugene Tsyrklevich, Bennet Yee. Dynamic Detection and Prevention of Race Conditions in File Accesses. *In Proceedings of the 12th USENIX Security Symposium*, 2003. Dostęp internetowy: http://static.usenix.org/event/sec03/tech/full_papers/tsyrklevich/tsyrklevich.pdf [02.06.2013].

[116] University of Maryland. *MultithreadedTC Home*. Dostęp internetowy: http://www.cs.umd.edu/projects/PL/multithreadedtc/overview.html [02.06.2013].

[117] University of Twente. *MoonWalker Home*. Dostęp internetowy: http://fmt.cs.utwente.nl/tools/moonwalker/ [02.06.2013].

[118] Valgrind Developers. *Valgrind Home*. Dostęp internetowy: http://valgrind.org/ [02.06.2013].

[119] Arjan van de Ven. How to NOT write kernel driver. Red Hat Inc. Dostęp internetowy: http://www.fenrus.org/how-to-not-write-a-device-driver-paper.pdf [02.06.2013].

[120] Robin Ward. How our users exploited concurrency and how we fixed it. *Evil Trout*, January 2013. Dostęp internetowy: http://eviltrout.com/2013/01/10/exploiting-concurrency.html [02.06.2013].

[121] Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. Dostęp internetowy: http://www.watson.org/~robert/2007woot/2007usenixwoot-exploitingconcurrency.pdf [02.06.2013].

[122] Robert N.M. Watson. Concurrency and security. Computer Laboratory, University of Cambridge. Dostęp internetowy: http://www.cl.cam.ac.uk/teaching/1213/SecurityII/2013-security2-5-watson.pdf [02.06.2013].

[123] Jinpeng Wei, Calton Pu. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. *In Fourth USENIX conference on File and Storage Technologies(FAST '05)*, 2005. Dostęp internetowy: http://static.usenix.org/events/fast05/tech/full_papers/wei/wei.pdf [02.06.2013].

[124] Junfeng Yang, Ang Cui, Sal Stolfo, Simha Sethumadhavan. Concurrency Attacks. Department of Computer Science, Columbia University. Dostęp internetowy: http://www.cs.columbia.edu/~junfeng/papers/con-tr.pdf [02.06.2013].

[125] Yuan Yu, Tom Rodeheffer, Wei Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, 2005. Dostęp internetowy: http://research.microsoft.com/pubs/65170/sosp05-racetrack.pdf [02.06.2013].

[126] Michał Zalewski. Delivering Signals for Fun and Profit. BindView Corp., 2001. Dostęp internetowy: http://lcamtuf.coredump.cx/signals.txt [02.06.2013].

[127] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, Thomas Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.