Zeszyty Naukowe Wydziału
Elektroniki i Informatyki
Politechniki Koszalińskiej

ISSN:1897-7421
Nr 13, Listopad 2018
Str: 5-25

Damian Giebas
Rafał Wojszczyk
Wydział Elektroniki i Informatyki
Politechnika Koszalińska
ul. J. J. Śniadeckich 2
75-453 Koszalin

Zastosowanie wybranych reprezentacji graficznych do analizy aplikacji wielowątkowych

Słowa kluczowe: reprezentacje graficzne, Sieci Petriego, Control Flow Graph, Systemy Współbieżnych Procesów, aplikacje wielowątkowe

Wstęp

Aplikacje pisane na współczesne komputery są bardzo różnorodne i znajdują zastosowanie w prawie każdej dziedzinie życia. Wiele z tych aplikacji to jednowątkowe programy, które wykonują zadania jedno po drugim. Wraz z rozwojem sprzętu komputerowego i wprowadzeniem na rynek procesorów umożliwiających współbieżne wykonywanie zadań zaczęły pojawiać się aplikacje wielowątkowe. Niektóre z języków programowania jak C i C++ nie były tworzone z myślą o wielowątkowości. Aby uzupełnić te braki dla języka C powstała biblioteka pthreads zgodna z wciąż rozwijanym standardem („ISO/IEC 9945-1:2003 - Information technology -- Portable Operating System Interface (POSIX) -- Part 1: Base Definitions" b.d.). Język C++ otrzymał wsparcie dla wielowątkowości w postaci rozszerzenia biblioteki standardowej wraz z wprowadzeniem standardu C++ 11 (Hinnant i in. 2007).

Języki te wybrano do przedstawiania przykładowych programów wielowątkowych ponieważ istnieje na rynku mnogość oprogramowania stworzonego w tych językach i wciąż wiele takiego oprogramowania powstaje.

Programowanie wielowątkowe w porównaniu do wcześniej stosowanego programowania jednowątkowego posiada szereg zalet jak i szereg wad (Torp 2002).

Najważniejsze z nich zostały przedstawione poniżej:

Zalety

- Responsywność – w przypadku długich zadań w programach z graficznym interfejsem użytkownika programy jednowątkowe ulegają tzw. zamrożeniu (ang. freezing). Problem ten nie występuje w przypadku aplikacji wielowątkowych, gdyż zadania takie mogą zostać oddelegowane do osobnych wątków.
- Współdzielenie zasobów – wątki uruchamiane w ramach jednego procesu współdzielą zasoby komputera. Wszystko dzieje się w ramach jednej przestrzeni adresowej. W przypadku programów jednowątkowych zadania należało delegować do osobnych procesów a komunikacja odbywała się poprzez kopiowanie wartości z jednej przestrzeni adresowej do drugiej.
- Oszczędność – programy wielowątkowe posiadają mniejsze zużycie pamięci niż rozwiązania wykorzystujące kilka jednowątkowych aplikacji.
- Skalowalność – aplikacje wielowątkowe znacznie lepiej wykorzystują możliwości sprzętowe procesorów wspierających wielowątkowość niż zbiór aplikacji jednowątkowych wykonujących wspólnie to samo zadanie. Jednocześnie maszyny stanów aplikacji wielowątkowych są znacznie mniej skomplikowane niż maszyny stanów analogicznego rozwiązania złożonego z aplikacji jednowątkowych.

Wady

- Złożoność kodu aplikacji – każde uruchomienie aplikacji może wyglądać inaczej i jest zależne od aktualnego stanu pozostałych elementów systemu. Programista nigdy nie wie, ile czasu procesora planista przydzieli danemu wątkowi a także nie zna kolejności ich pracy. Taki stan rzeczy posiada więc wpływ na:
  - Debugowanie – debugowanie takich aplikacji jest znacznie utrudnione gdyż sam proces debugowania może wpływać na sposób zachowania aplikacji.
  - Testowanie – testowanie aplikacji jest bardzo trudne gdyż bardzo ciężko jest przewidzieć wszystkie możliwe stany w jakich znajdzie się aplikacja.
- Deadlock – zjawisko nazywany także zakleszczeniem lub blokadą. Sytuacja, w której proces lub wątek w przypadku aplikacji wielowątkowych zamawia dostęp do zasobów i przechodzi w stan oczekiwania. Istnieje możliwość, że oczekujący proces lub wątek nigdy nie zmieni swojego stanu, ponieważ zamawiane przez niego zasoby są

przetrzymywane przez inne czekające procesy (Silberschatz, Galvin, i Gagne 2005).

- Race condition – zjawisko nazywane także szkodliwą rywalizacją. Sytuacja, w której kilka procesów (lub wątków w przypadku aplikacji wielowątkowych) współbieżnie sięga po te same dane i wykonuje na nich działania, wskutek czego wynik tych działań zależy od porządku w jakim następował dostęp do danych (Silberschatz, Galvin, i Gagne 2005).

Zjawiska deadlock i race condition znane były wcześniej gdyż występują one nie tylko w aplikacjach wielowątkowych ale także w rozwiązaniach, w których aplikacje jednowątkowe wykorzystują wspólne zasoby.

Inne znane zjawiska występujące w aplikacjach wielowątkowych to opisany w rozdziale nr 4 atomicity violation i order violation (Lu i in. 2016) nieporuszany w tej pracy. Niniejsza praca skupia się na reprezentacjach graficznych aplikacji wielowątkowych, które pozwolą przede wszystkim na wyeksponowanie miejsc, w których występuje zjawisko typu race condition. Najbardziej znaną reprezentacją graficzną, która pozwoliła na opracowanie metod i zbudowanie narzędzi do detekcji błędów aplikacji wielowątkowych to Control Flow Graph omawiana w rozdziale nr 1. Inną popularną reprezentacją graficzną są Sieci Petriego omawiane w rozdziale nr 2. Wykorzystywane dziś reprezentacje graficzne posiadają pewien szereg ograniczeń, które rzutują na opracowane metody i wykorzystujące je narzędzia. Wśród tych narzędzi znajdują się:

- Helgrind – narzędzie z pakietu Valgrind's Tool Suite[1] do nieinwazyjnego debugowania programów wielowątkowych, pozwalające na wykrycie wszelkiego rodzaju problemów związanych z równoległym dostępem do zasobów. Na stronie twórców znajduje się informacja o tym, że nie gwarantują oni poprawnego działania aplikacji. Mimo wszystkich zalet Helgrind nie posiada możliwości zdalnego debugowania, która to jest niezbędna do pracy w bardzo dużej ilości środowisk gdzie wykorzystywane są języki C i C++ np. w systemach wbudowanych.
- ThreadSanitizer[2] – narzędzie firmy Google bazujące na Helgrind i posiadające także jego ograniczenia. Oba narzędzia wykorzystują algorytm opisany w dokumentacji Helgrind'a. ThreadSanitizer jest narzędziem znajdującym się w pakiecie kompilatorów LLVM/Clang i GCC dla platformy x86. Narzędzie to, podobnie jak Helgrind, jest w fazie beta i jego autorzy nie gwarantują poprawnego działania.

[1] http://valgrind.org/info/tools.html#others
[2] https://clang.llvm.org/docs/ThreadSanitizer.html

- RacerX – narzędzie wykrywające zjawiska race condition i deadlock opisane w pracy (Engler i Ashcraft 2003) wykorzystujące statyczną analizę kodu (tj. pełna analiza kodu źródłowego aplikacji). Wykrywanie odbywa się poprzez stworzenie Control Flow Graf dla analizowanej aplikacji i wzbogacenie go o spisy wywołań funkcji, użytych zmiennych globalnych, wskaźników do zmiennych przekazanych jako parametr i opcjonalnie o spis wszystkich lokalnych zmiennych. Narzędzie to obecnie nie jest publicznie dostępne dla nikogo[3].
- Relay – narzędzie stworzone na Uniwersytecie Kalifornijskim w San Diego do statycznej analizy kodu celem wykrywania zjawiska race condition. Narzędzie to działało na podobnej zasadzie do RacerX, co zostało opisane w pracy (Voung, Jhala, i Lerner 2007). Narzędzie to posłużyło do analizy kodu jądra Linuksa w wersji 2.6.15. Analiza została wykonana na liczbie 4.5 mln. wierszy kodu i wykazała obecność 53 miejsc, w których występowało zjawisko race condition. Jest to jedyny tak szczegółowy raport dotyczący analizy kodu jądra Linuksa, wykonywany poprzez statyczną analizę kodu pod kątem występowania tego zjawiska. Narzędzie choć dostępne publicznie nie jest rozwijane od 2010 roku.

Dwa pierwsze opisane narzędzia do detekcji błędu wykorzystują techniki dynamiczne tj. pracują ze skompilowanym kodem aplikacji, tymczasem dwa kolejne narzędzia do pracy wymagają kodu źródłowego aplikacji, gdyż wykorzystują techniki statyczne poprzez analizę kodu źródłowego. Wykorzystanie Systemów Współbieżnych Procesów (SWP) do detekcji zjawisk race condition i deadlock jest przykładem podejścia statycznego. Główną zaletą metod dokonujących analizy kodu źródłowego jest fakt, że są one niezależne od platformy na jaką pisany jest kod aplikacji, jednak nie są one w stanie uwzględnić opisywanych zjawisk wywołanych agresywną optymalizacją kompilatora (skutkującą np. zamianą pobrania zawartości zmiennej przez stałą liczbową). Zjawiska wywołane agresywną optymalizacją są wykrywane dzięki technikom dynamicznym, jednak narzędzia wykorzystujące techniki dynamiczne są silnie związane z platformą i tak na przykład wszystkie aspekty Helgrinda można wykorzystać tylko na platformach x86 i AMD64.

Języki C i C++ doczekały się rozszerzeń, które pozwalały na równoległą pracę już wcześniej. Rozszerzenie Cilk („A Brief History of Cilk”, b.d.) dla C i C++ zostało stworzone w 1990 roku na MIT i skomercjalizowane jako Cilk++ a następnie sprzedane firmie Intel, która rozwija je pod nazwą CilkPlus. Rozszerzenie to nie zdobyło większej popularności i będzie utrzymywane tylko do

[3] https://goo.gl/DgYzt5

2018 roku. Firma Intel proponuje migrację z CilkPlus do frameworka OpenMP bądź Intel Threading Building Block (Intel TBB).

Wspomniany framework OpenMP (Bull, Reid, i McDonnell 2012) stworzony został dla języków Fortran i C a potem rozszerzony dla C++98 i jest wspierany przez największe firmy w sektorze IT. Zrównoleglanie pracy programu z OpenMP odbywa się poprzez używanie odpowiednich dyrektyw preprocesora, które powodują wzrost złożoności kodu a także nie współpracują z najnowszymi wersjami języka C++.

Konkurencyjne rozwiązanie dla OpenMP czyli biblioteka Intel TBB („Intel Threading Building Blocks Documentation", b.d.) dla języka C++ jest znacznie lepiej przystosowana do współpracy z najnowszymi wersjami tego języka. Niestety w przypadku zastosowania Intel TBB spora część kodu musi zostać napisana na nowo z wykorzystaniem jej elementów.

Charm++ („Introduction to Charm++ Concepts", b.d.) jest dedykowanym frameworkiem dla języka C++ do tworzenia aplikacji z przetwarzaniem równoległym. Wprowadza on nowy paradygmat tj. zorientowane obiektowo asynchroniczne przesyłanie informacji (ang. object-oriented asynchronous message passing parallel programming paradigm), który dekomponuje program do kontenerów (ang. chares), które komunikują się za pomocą obiektów nazywanych komunikatami. Wady tego rozwiązania zostały przedstawione w prezentacji (Aiken b.d.). Największa z wad to łatwa do pominięcia synchronizacja pracy kontenerów, która to jest wymagana aby uniknąć zjawiska race condition. Inną dużą wadą Charm++ jest przeniesienie na programistę obowiązku zarządzania pamięcią komunikatów. Złe zarządzanie może doprowadzić do bardzo groźnych wycieków pamięci w przypadku, gdy następuje alokacja zasobów a nie są one zwalniane.

Powyższe rozwiązania dla języków C i C++ posiadają jedną niepożądaną cechę tj. wysoki poziom skomplikowania kodu napisanego z ich wykorzystaniem. W przypadku wykorzystania biblioteki pthread czy standardu C++11 kod ten jest znacznie bardziej czytelny.

Dalsza część pracy dotyczy lokalizacji zjawiska race condition znajdującego się w kodzie programu przedstawionego na listingu nr 1, przy pomocy reprezentacji graficznych aplikacji wielowątkowych. Program został napisany w języku C z wykorzystaniem biblioteki pthreads. Celem programu z poniższego listingu jest wykonanie miliona operacji inkrementacji zmiennej *balance* przez każdy z wątków aplikacji. Wynikiem działania powinna być wartość dwóch milionów. Niestety operacje inkrementacji na współdzielonym zasobie nie są synchronizowane, wynikiem czego w programie dochodzi do zjawiska race condition. Synchronizacja powinna zostać wykonana poprzez użycie mechanizmów synchronizacji nazywanych mutexami dostarczonych wraz z biblioteką pthreads. Mutexy są to abstrakcyjne struktury, które wykorzystują mechanizm wzajemnego wykluczania się

celem synchronizacji pracy nad wybranymi zasobami. Słowo mutex pochodzi od angielskich słów mutual exclusion.

Mimo iż możliwe jest pisanie w języku C programów wielowątkowych to brak natywnego wsparcia języka powoduje, że w programach tych często występują zjawiska race condition czy deadlock. Poniższy kod aplikacji zawierający zjawisko race condition będzie przekształcany do kolejnych reprezentacji graficznych, w których zjawisko to powinno zostać wyeksponowane, gdyż reprezentacje graficzne nie posiadają ograniczeń języka C i są lepiej przystosowane do przedstawiania wysokopoziomowych idei.

```
#include <stdio.h>
#include <pthread.h>

static volatile int balance = 0;

void* deposit(void *param) {
	// Block B
	char *who = (char*)param;

	int i;
	printf("%s: begin\n", who);
	for (i = 0; i < 1000000; i++)
	// Block C
	{
		balance = balance + 1; // Place with uncontrolled access. Race condition
place.
	}
	// Endblock C
	printf("%s: done\n", who);
	return NULL;
	// Endblock B
}

int main() {
	// Block A
	pthread_t p1, p2;
	char a[] = "A";
	char b[] = "B";
	printf("main() starts depositing, balance = %d\n", balance);

	pthread_create(&p1, NULL, deposit, a);
	pthread_create(&p2, NULL, deposit, b);
	// Endblock A

	// Block D
	pthread_join(p1, NULL);
	pthread_join(p2, NULL);
	printf("main() A and B finished, balance = %d\n", balance);
	return 0;
	// Endblock D
}
```

Listing 1. Kod aplikacji wielowątkowej zawierający zjawisko race condition

Zastosowanie metod graficznych wynika z potrzeby posiadania uniwersalnego narzędzia, które pozwoli na analizę kodu i wykrycie miejsca występowania omawianych zjawisk. Jak już wcześniej było wspomniane graficzne metody są lepsze do przedstawiania wysokopoziomowych idei niż język programowania jakim jest C. Dodatkowo przekształcanie kodu źródłowego do reprezentacji graficznej jest rozwiązaniem niezależnym od platformy, na którą kod będzie skompilowany. Taka transformacja kodu źródłowego do reprezentacji graficznej jest elementem statycznej analizy kodu.

1. Control Flow Graph

Control Flow Graph (CFG) to nic innego jak graf skierowany, który jest jedną z możliwych reprezentacji graficznych aplikacji wielowątkowej. CFG przedstawiony w pracy (Allen 1970) składa się z węzłów i krawędzi, które odpowiadają kolejnym blokom kodu i determinują kolejność ich wykonywania.

CFG zakłada istnienie 3 rodzajów węzłów. Pierwszym rodzajem węzła jest węzeł wejścia (ang. entry node), który cechuje się tym, że nie posiada on przodka natomiast posiada potomków.

Drugi rodzaj węzła to węzeł wyjścia (ang. exit node), który analogicznie do węzła wejścia nie posiada potomków, ale posiada przodków.

Trzeci rodzaj węzłów to węzły posiadające zarówno przodków jak i potomków. Węzły te mogą posiadać przynajmniej jednego przodka i przynajmniej jednego potomka. Przodkowie jak i potomkowie mogą być węzłami bezpośrednimi jak i pośrednimi.

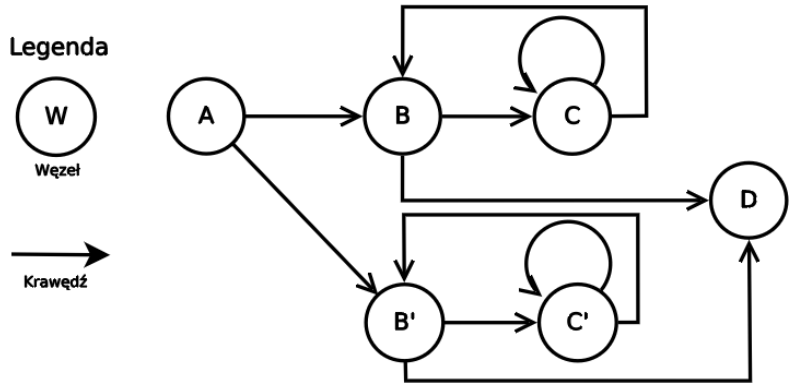
Innymi słowy CFG to graf skierowany G będący parą (B, E), gdzie B jest zbiorem węzłów $\{b_1,b_2,b_3,\ldots b_n\}$ natomiast E jest podzbiorem zbioru wszystkich możliwych krawędzi $\{(b_1,b_2),(b_1,b_3),\ldots,(b_m,b_n)\}$ występującymi między tymi węzłami.

Na rysunku nr 1 przedstawiono CFG dla aplikacji, której kod znajduje się na listingu nr 1. Kod podzielony jest na 4 logiczne bloki, które pozwalają na łatwe jego przekształcenie w CFG. Blok A jest fragmentem kodu przygotowującym aplikację do pracy na wątkach, natomiast blok D jest fragmentem kodu kończącym pracę na wątkach i kończącym pracę aplikacji. Bloki B i C są fragmentem aplikacji wykonywanym równolegle dlatego aby podkreślić ten aspekt aplikacji na CFG, dla jednego wątku zostały one oznaczone jako B i C a dla drugiego jako B' i C'. Dodatkowo blok C zawiera się w bloku B i jego praca jest powtarzana milion razy.

Z wyjątkiem funkcji main każdy logiczny blok tj. dowolna inna funkcja, ciało pętli, ciało instrukcji sterujących lub inny dowolny blok zawierający się w nawiasach klamrowych będzie posiadał swoje odzwierciedlenie w postaci węzła.

Funkcja main w językach C i C++ jest miejscem początku i końca pracy aplikacji dlatego rozbita została na wyżej omówione bloki A i D.

Rysunek nr 1 przedstawia CFG aplikacji, której kod znajduje się na listingu nr 1. Diagram zaczyna się od węzła A znajdującego się w funkcji main. Poprzedza on utworzenie dwóch wątków aplikacji, które przedstawione są jako węzły B i B'. Węzły C i C' są węzłami odpowiadającymi ciału pętli for, a więc dopóki warunek pętli będzie spełniony wciąż będzie wykonywać się wskazany blok, na co wskazuje obecność krawędzi (*C*,*C*) i krawędzi (*C'*,*C'*). Po zakończeniu pracy pętli kontrola wraca do głównego ciała funkcji, a więc do bloków B i B'. Program posiada już tylko blok D odpowiadający za zakończenie pracy a odpowiadający mu węzeł D kończy graf.



Rys. 1. Control Flow Graph aplikacji z listingu nr 1

Utworzony CFG odzwierciedla dokładnie kolejność wykonywanych bloków kodu, jednakże nie można wyczytać z niego informacji dotyczących operacji na współdzielonych zasobach. Operacje te, dziejące się w bloku C aplikacji, nie posiadają swojej reprezentacji graficznej i CFG będzie identyczny zarówno dla prawidłowo działającej aplikacji jak i tej, w której znajduje się zjawisko race condition. Wykorzystanie wyłącznie CFG nie jest wystarczająco dobrą notacją pozwalającą na wykrycie zjawisk race condition i deadlock.

Inną niedogodnością jest fakt, że CFG nie pozwala pokazywać zagnieżdżeń bloków. Bez dokładnego opisu można odnieść wrażenie, że po wyjściu z bloku C i powrotu do bloku B można ponownie wrócić do bloku C, co nie jest możliwe do wykonania w aplikacji.

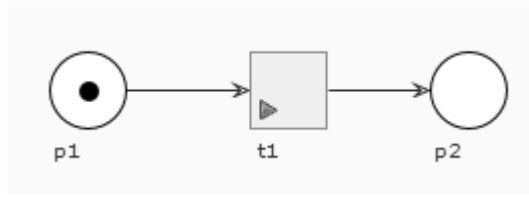
Fakt, że dany blok jest wykonywany w dwóch niezależnych wątkach pozwala domniemać, że jest to miejsce, w którym może wystąpić zjawisko race condition. W przypadku systemów, gdzie wątki nie współdzielą między sobą żadnych zasobów należałoby również sprawdzić wszystkie takie miejsca. W sytuacji gdy zasób jest

współdzielony przez dwa wątki, które nie posiadają wspólnych bloków, mechanizm ten jest niewystarczający do zlokalizowania zjawiska race condition.

Control Flow Graf jest wykorzystywane w narzędziach do detekcji omawianych zjawisk np. w narzędziu RacerX, a każdy z węzłów CFG tworzonych przez to narzędzie dodatkowo wzbogacany jest o spisy wywołań funkcji, użytych zmiennych globalnych, wskaźników do zmiennych przekazanych jako parametr i opcjonalnie o spis wszystkich lokalnych zmiennych. Dopiero w sytuacji, gdy istnieje komplet tych wszystkich informacji możliwe jest wykrycie zjawiska race condition.

2. Sieć Petriego

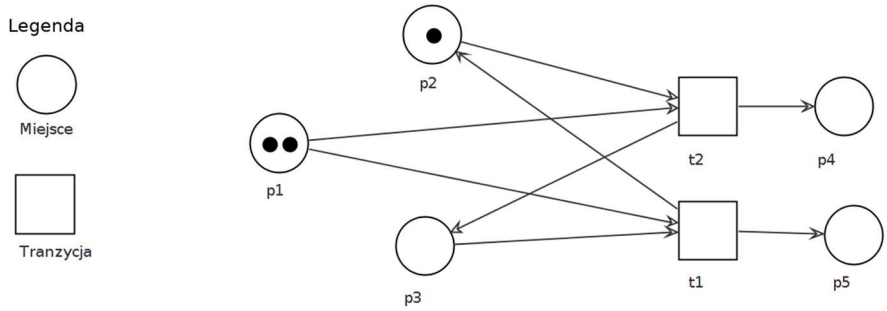
Sieć Petriego (SP) to formalny model przepływu informacji zaprojektowany do opisywania systemów asynchronicznych, w których zadania wykonywane są równolegle.



Rys. 2. Przykład Sieci Petriego

Sieci Petriego składają się z miejsc i tranzycji połączonych krawędziami skierowanymi (Peterson 1977). Przepływ informacji jest wykazywany poprzez przesuwanie żetonów między miejscami poprzez przejście po krawędziach. Na krawędziach znajdują się tranzycje, które odpowiadają za pozwolenie dokonania przejścia, co następuje gdy na wszystkich miejscach wejściowych tranzycji znajdują się żetony. Najprostszy przykład SP przedstawia rysunek nr 2. Znajdują się na niej dwa miejsca p1 i p2 i jedna tranzycja t1. Żeton znajdujący się w miejscu p1 zostanie przeniesiony po krawędziach do miejsca p2 ponieważ jest spełniony warunek przejścia tj. miejsce p1 jest jedynym miejscem wejściowym tranzycji t1 i posiada żeton.

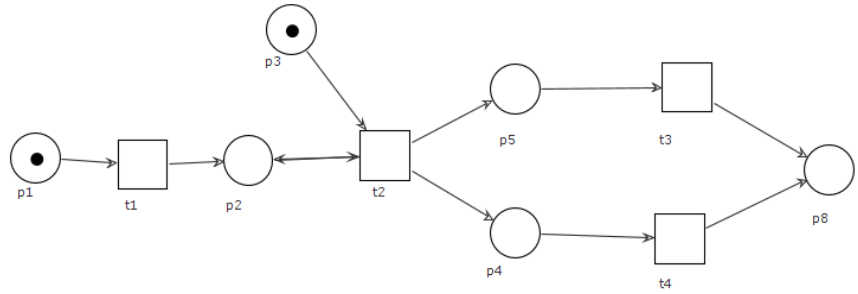
W przeciwieństwie do CFG, SP nie buduje się tylko na logicznych blokach kodu. Przy budowie należy uwzględnić takie rzeczy jak stan początkowy niektórych elementów tj. miejsce odzwierciedlające licznik pętli powinno posiadać tyle żetonów ile iteracji wykona pętla, czy też informacje o kolejności wykonywania poszczególnych zadań w przypadku gdy mogą one zostać zrobione równolegle. Przykładem może być sieć z rysunku nr 3 przedstawiająca zastosowanie mechanizmu wzajemnego wykluczania, który narzuca kolejność przesunięć żetonów z miejsca p1 przez tranzycję t1 i t2.



Rys. 3. Przykładowa Sieć Petriego z mechanizmem wzajemnego wykluczania

Możliwość wykorzystania mechanizmów wzajemnego wykluczania (w powyższym przykładzie składa się on z tranzycji t1, t2 i miejsc p2, p3) pozwala na kontrolę przesunięć żetonów w sieci i symulować wielowątkowość aplikacji. Jednakże nie jest to realne odzwierciedlenie. W przypadku aplikacji wielowątkowych, których głównym celem jest szybkość wykonania operacji, programista nie narzuca ich kolejności wykonywania. To planista decyduje, który wątek w danym momencie pracuje i sytuacja naprzemiennej pracy wątków (jak zostało to pokazane na rysunku powyżej) jest mało prawdopodobna.

Na rysunku numer 4 przedstawiona została Sieć Petriego dla rozważanej aplikacji. Sieć ta jest zbudowana z 6 miejsc i 4 tranzycji. Miejsce p1 odpowiada blokowi A wybranej aplikacji i oznacza jej uruchomienie. Miejsce p2 jest odpowiednikiem momentu uruchomienia obu wątków aplikacji. W przypadku Sieci Petriego możemy symulować działanie pętli for, a więc blok C w tym przypadku będzie składał się z miejsc (p3, p5, p8) i tranzycji (t2, t3) dla pierwszego wątku, a także z miejsc (p3, p4, p8) i tranzycji (t2, t4) dla drugiego wątku. Miejsce p3 jest licznikiem pętli, które powinno mieć milion żetonów, gdyż tyle iteracji wykonuje każda z pętli w wątkach. Umożliwi to każdej gałęzi sieci wykonać się milion razy, tak jak w każdym wątku wykonywane jest milion operacji na wspólnym zasobie.



Rys. 4. Sieć Petriego aplikacji wielowątkowej z listingu nr 1

Miejsce p8 jest odpowiednikiem bloku D aplikacji i kończy ono całą sieć, a ilość żetonów odpowiada wartości zmiennej balance. Jeśli na miejscu p3 znajdowałoby się milion żetonów (tj. tyle ile wynosi maksymalna ilość iteracji pętli w bloku C) to po wykonaniu symulacji w miejscu p8 znajdzie się ich 2 miliony.

Konstrukcja sieci nie pozwala na wystąpienie sytuacji, w której zachodzi zjawisko race condition, tak więc wynik działania sieci będzie zgodny z oczekiwanym wynikiem działania aplikacji ale nie z jej realnym działaniem.

Dodatkową wadą takiej reprezentacji jest to, że do jednego i tego samego kodu aplikacji można zbudować wiele modeli sieci. Sytuacja ta powoduje, że gdy stworzony zostanie model sieci dla aplikacji nigdy nie ma pewności, że będzie można odczytać z niego wszystkie niezbędne informacje pozwalające zlokalizować poszukiwane informacje.

W przypadku zastosowania w sieci mechanizmów wzajemnego wykluczania należy zawsze określić, która tranzycja będzie posiadała priorytet, wynikiem czego jest z góry ustalona kolejność działania tranzycji. Sytuacja taka nie ma miejsca w aplikacjach. Programista nigdy nie ma pewności, który wątek pierwszy otrzyma dostęp do zasobu, gdyż praca wątków nastawiona jest na jak najszybsze wykonywanie zadań i są one wykonywane od razu w momencie, gdy planista przydzieli wątkowi czas procesora. Także w przeciwieństwie do sieci mechanizmy wzajemnego wykluczenia dostarczane z językiem C nie wymuszają kolejności pracy wątków.

3. Sformułowanie problemu

Przedstawiona analiza dwóch powszechnie znanych reprezentacji graficznych aplikacji wielowątkowych pozwala stwierdzić, że za ich pomocą zlokalizowanie zjawiska race condition jest bardzo złożone i w wielu przypadkach wymaga stosowania dodatkowych (nadmiarowych) mechanizmów kontrolnych.

Posiadany jest kod aplikacji wielowątkowej napisany w języku C z wykorzystaniem biblioteki pthreads.

Ograniczeniami jest składnia języka C, jego gramatyka a także fakt, że obliczenia muszą być wykonywane równolegle.

Zatem pytanie jest następujące. Czy kod aplikacji jest poprawny tj. nie występują w nim zjawiska:

- deadlock,
- race condition?

Dwie wcześniej omówione reprezentacje graficzne nie pozwoliły na stwierdzenie czy kod znajdujący się na listingu nr 1 jest wolny od tych zjawisk. W punkcie nr 4 przedstawiona zostanie reprezentacja wykorzystująca modele systemów współbieżnych procesów do tego celu. Przedstawione zostaną dwie reprezentacje,

z których na pierwszej zjawisko race condition będzie widoczne, a na drugiej zostanie przedstawione rozwiązanie eliminujące to zjawisko.

4. Model SWP dla aplikacji wielowątkowych

System procesów jest to zbiór procesów $P=\{P_i|i=1\ldots ln\}$ realizujących operacje w oparciu o zbiór wspólnie wykorzystywanych zasobów $R=\{R_k|k=1\ldots lm\}$. Współbieżne wykonanie procesów oznacza, że każda kolejna operacja jednego procesu rozpoczyna się przed zakończeniem operacji innego procesu i związane jest z ograniczonym dostępem procesów do współdzielonych zasobów (Banaszak, Majdzik, i Wójcik 2008). Specyficznym przypadkiem są systemy, w których procesy są realizowane cyklicznie (tzn. operacje procesów są powtarzane wielokrotnie w stałych odcinkach czasu). W tym ujęciu przez System Współbieżnych Procesów Cyklicznych (SWPC) rozumie się jako zbiór wykonujących się współbieżnie procesów cyklicznych, które są związane ze sobą poprzez korzystanie ze wspólnych zasobów (Bocewicz, Banaszak, i Wójcik b.d.).

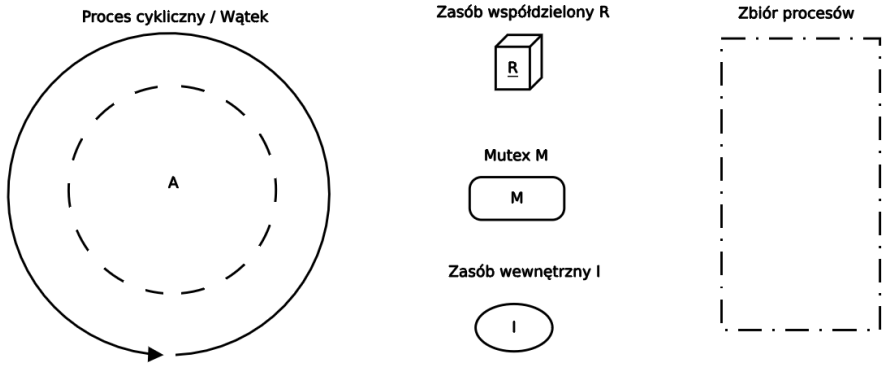
Gdy mowa o SWPC należy wspomnieć o konfliktach żądań zasobowych, które są konsekwencją wystąpienia m.in. takich zjawisk jak zagłodzenie i blokada. Podobne zjawiska można spotkać w aplikacjach wielowątkowych. Zagłodzenie (ang. starvation) występuje w momencie, gdy jeden z wątków aplikacji przez cały okres jej działania nie zwalnia określonego zasobu i tym samym uniemożliwia dostęp do niego innym wątkom. Blokada (ang. deadlock) natomiast występuje wtedy, gdy dwa wątki (lub więcej) próbują otrzymać dostęp do wzajemnie zajmowanych przez siebie zasobów i powstaje tzw. cykl żądań zasobowych. Sytuacja taka powoduje, że każdy z wątków czeka aż pozostałe zwolnią swoje zasoby, co nigdy nie następuje.

Kolejnym specyficznym zjawiskiem aplikacji wielowątkowych jest zjawisko race condition czyli sytuacja, w której stan współdzielonego zasobu (np. wartość zmiennej reprezentowanej przez ten zasób) jest zmieniany przez jeden z wątków w momencie, gdy inne wątki dokonują operacji z już nieaktualną wartością zasobu. Konsekwencją takiego zjawiska jest możliwość uzyskiwania różnych wyników aplikacji (często trudnych do przewidzenia) w zależności od kolejności dostępu wątków do współdzielonych zasobów.

Analogicznie jak omawiane w poprzednich punktach modele CFG i SP systemy współbieżnych procesów cyklicznych mogą również być wykorzystywane do reprezentacji aplikacji wielowątkowych. W tym celu wykorzystuje się zbiór elementów graficznych (Rys. 5) składający się z:

- zasobów współdzielonych reprezentujących instancję dowolnego typu, która jest współdzielona między wątkami np. poprzez wskaźnik lub jako zmienna globalna,

- zasobów wewnętrznych wątków, które podobnie jak zasoby współdzielone są instancjami dowolnego typu, a ich okres życia trwa tak długo jak okres życia wątków,
- procesów cyklicznych reprezentujących wątki aplikacji,
- mechanizmu synchronizacji (mutexu) zapewniającego wzajemne wykluczanie procesów na zasobach (w języku C mutex jest algorytmem implementowanym w postaci obiektu, na którym mogą być wykonywane operacje blokowania i zwolnienia).

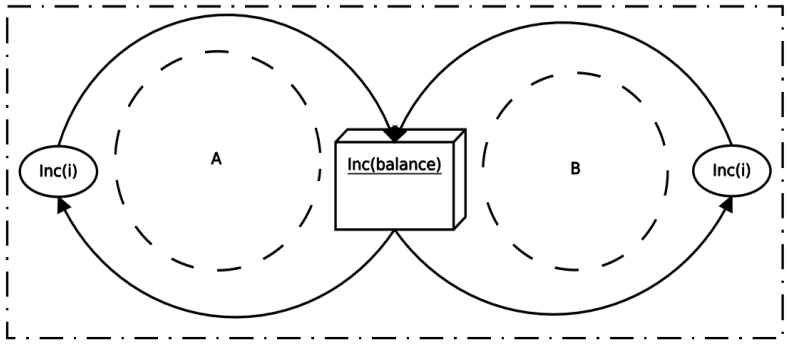


Rys. 5. Elementy SWPC zastosowane modelowania aplikacji wielowątkowych

Zarówno na zasobach jak i mutexach mogą być wykonywane operacje procesów cyklicznych (nazwy tych operacji są podawane wewnątrz zasobu). Są to między innymi:

- Inc – operacja inkrementacji zasobu,
- Lock – operacja założenia blokady na obiekcie mutex'u,
- Unlock – operacja zwolnienia blokady z obiektu mutex'u.

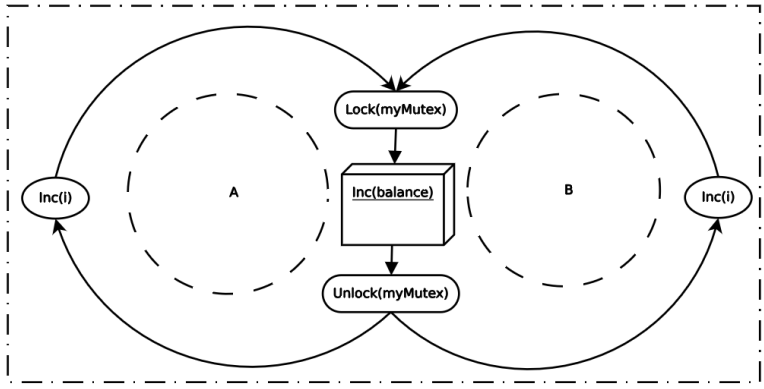
Proponowany model SWPC (wykorzystujący zaproponowany zestaw elementów), w przeciwieństwie do Sieci Petriego i CFG, ukrywa wiele szczegółów implementacyjnych. Uwydatnione na nim zostaną tylko te cechy aplikacji, które są istotne do oceny jej poprawności (pod względem występowania zjawisk prowadzących do konfliktów żądań zasobowych). Takie podejście powinno pozwolić na dokładne odtworzenie aplikacji z modelu i jednocześnie wskazać miejsca, w których może występować zjawisko race condition lub deadlock.



Rys. 6. Model SWPC aplikacji wielowątkowej z listingu nr 1

Rysunek nr 6 przedstawia SWPC dla aplikacji z listingu nr 1. Różni się on znacznie od sieci Petriego i CFG. System zawiera parę procesów (A, B) odpowiadających obu wątkom rozważanej aplikacji. Procesy A i B znajdują się w ramach jednego zbioru, tak samo jak oba wątki pracują w ramach jednej aplikacji. Oba procesy wykonują operację zwiększenia wartości współdzielonego zasobu o nazwie *balance* lub zwiększają wartość swoich wewnętrznych zasobów, analogicznie do wątków przykładowej aplikacji. Pozostałe elementy aplikacji tj. wyświetlanie informacji na standardowe wyjście, inicjalizacja zmiennych czy zakończenie pracy wątków zostają ukryte, gdyż są one zbędne w procesie wykrywania zjawiska race condition.

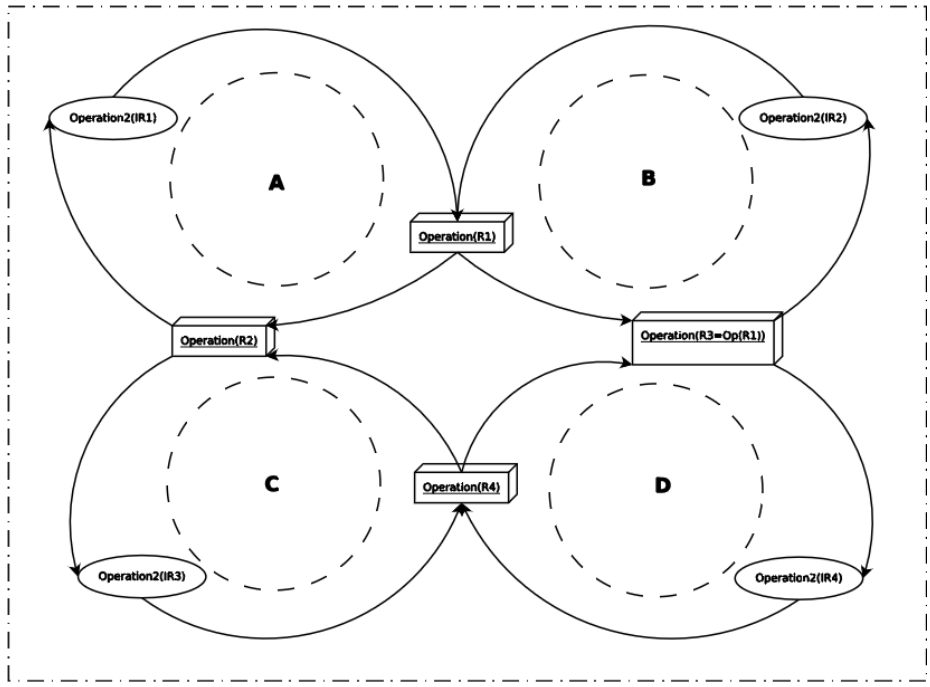
Prezentowany SWPC choć bardzo ogólny, posiada niezbędne informacje dotyczące odtworzenia rozważanej aplikacji. Programista otrzymuje komplet informacji pozwalających odtworzyć jej kod. Na rysunku łatwo dostrzec, że praca na współdzielonym zasobie nie jest synchronizowana tj. brakuje mutexu zapewniającego wzajemne wykluczanie procesów na zasobie współdzielonym. Oznacza to, że na tym zasobie może dochodzić do zjawiska race condition.



Rys. 7. Model SWPC aplikacji z listingu nr 1 bez błędu race condition

Ukrycie zbędnych detali dotyczących realizowanych w aplikacji wątków czyni model bardzo czytelnym. Pominięcie szczegółów implementacyjnych nie wpływa na ocenę poprawności aplikacji. W przeciwieństwie do SP i CFG, model SWPC uwydatnia wrażliwe elementy aplikacji, co przekłada się na lepsze przedstawienie sposobu działania aplikacji i na zlokalizowanie miejsc, gdzie mogą wystąpić potencjalne błędy.

Wyeliminowanie błędu wynikającego z wystąpienia zjawiska race condition jest możliwe w wyniku dodania elementów synchronizacji. Na rysunku nr 7 przedstawiono model SWPC z mutexami, które eliminują zjawisko race condition. W przedstawionym rozwiązaniu procesy przed zajęciem zasobu współdzielonego blokują do niego dostęp (Lock(myMutex)), a następnie go zwalniają (Unlock(myMutex)) po zakończeniu operacji na tym zasobie.

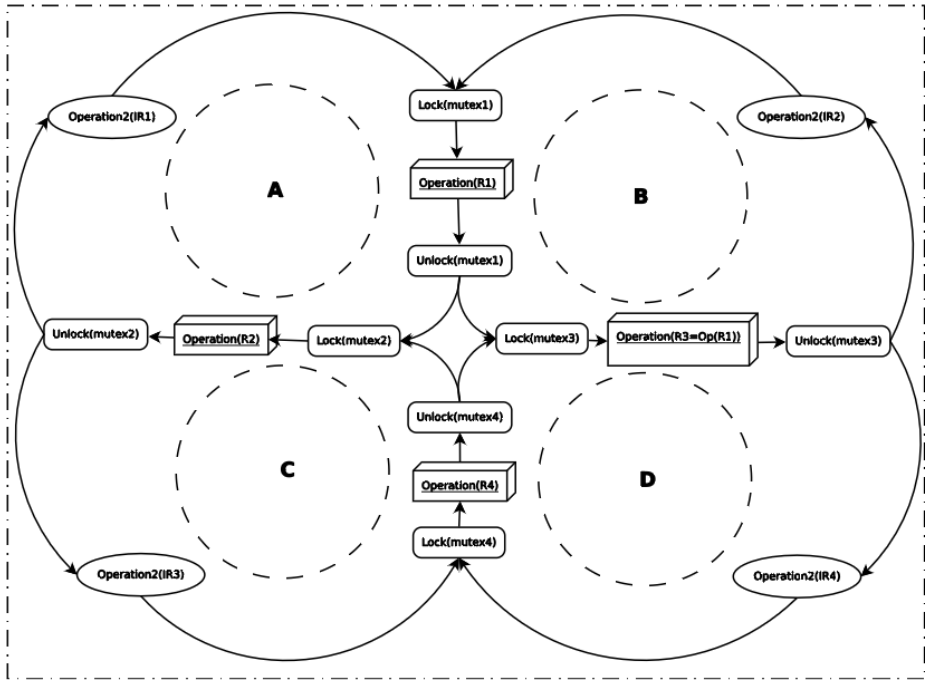


Rys. 8. Przykładowy model SWPC przykładowej aplikacji bez elementów synchronizujących

Aplikacja z listingu nr 1 jest przykładem, dla którego zbudowany SWPC nie jest skomplikowany. Na rysunku nr 8 znajduje się model SWPC dla hipotetycznej aplikacji posiadającej cztery wątki. W aplikacji znajdują się cztery współdzielone zasoby R1, R2, R3 i R4, a każdy z wątków pracuje z dwoma z nich i z własnym zasobem wewnętrznym. Dodatkowo w wątku B operacja na zasobie R3 jest zależna

od nowej wartości zasobu R1 (zależność ta jest wyrażona poprzez równanie R3=Op(R1) wpisane w element graficzny) ustawianej właśnie przez wątek B. Z rysunku łatwo można wyczytać, że operacje realizowane na zasobach współdzielonych nie są synchronizowane, a więc niewątpliwie może dochodzić do zjawiska race condition. Poza race condition w aplikacji zachodzi także zjawisko atomicity violation. Zjawisko to jest konsekwencją związku między zasobem R1 i R3. Stan zasobu R1 wpływa na stan zasobu R3. Zanim proces B wykona operację na zasobie R3, stan zasobu R1 może zostać zmieniony przez proces A.

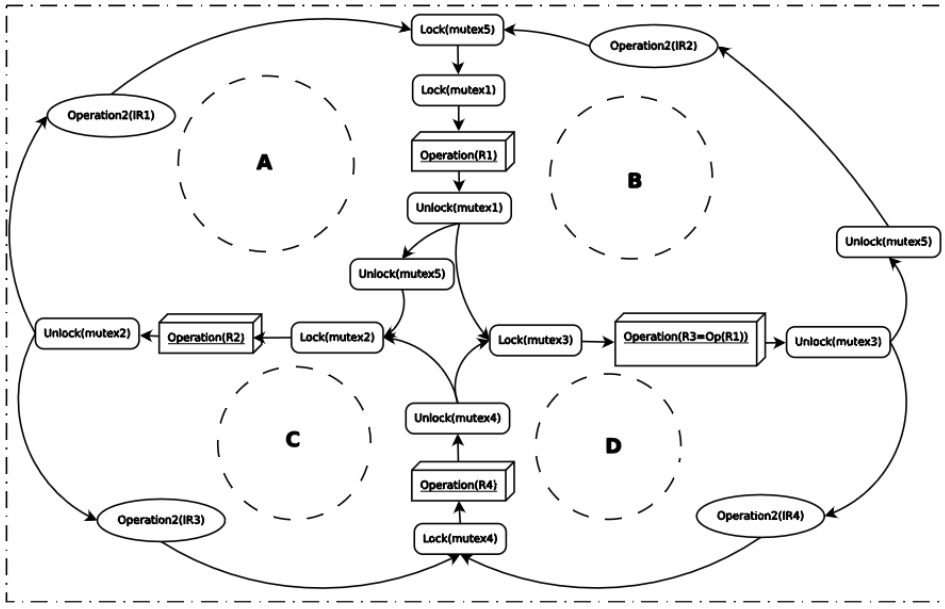
Wyeliminowanie zjawiska race condition sprowadza się do umieszczenia w aplikacji 4 mutexów mutex1, mutex2, mutex3, mutex4 celem zapewnienia wzajemnego wykluczania procesów na zasobach współdzielonych - odpowiedni SWPC jest przedstawiony na rysunku nr 9. Przed każdą operacją na współdzielonym zasobie dokonywana jest akcja blokady na odpowiednim mutexie, a po jej wykonaniu mutex ten jest zwalniany.



Rys. 9. Przykładowy model SWPC przykładowej aplikacji z atomicity violation

Niestety takie podejście nie eliminuje zjawiska atomicity violation. Zjawisko to wciąż jest obecne, gdyż wątek B po zwolnieniu mutex1 przechodzi do operacji blokowania mutex3. W tej sytuacji niezabezpieczony przez wątek B zasób R1 może zostać zmieniony przez wątek A.

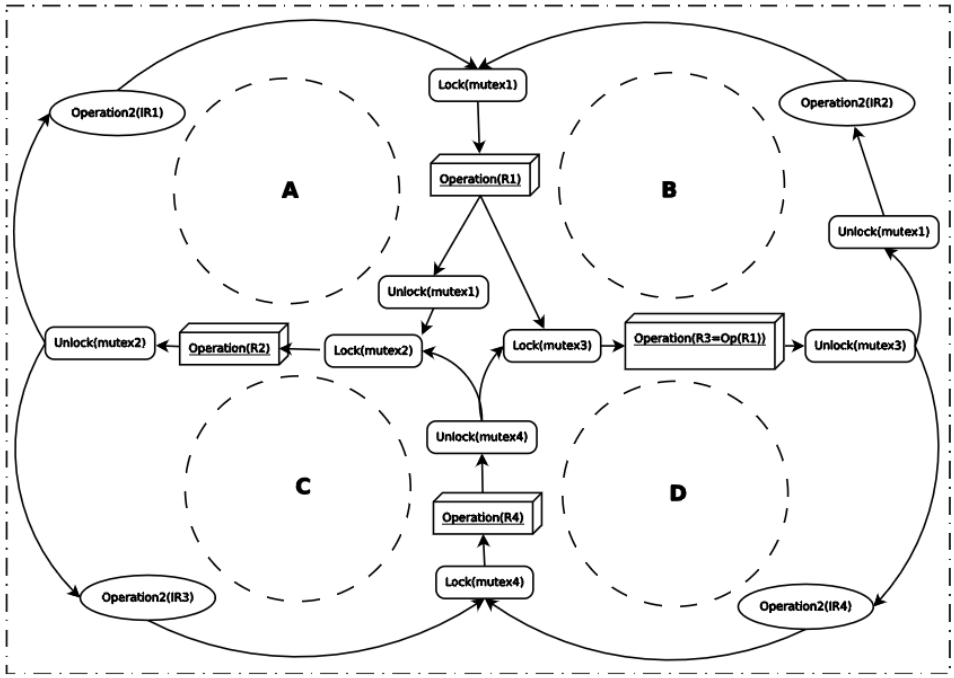
Jednym z dopuszczalnych sposobów wyeliminowania tego zjawiska jest wprowadzenie dodatkowego mutexu, który w wątku B będzie strzegł pracy na obu zasobach, a w wątku A tylko operacji na zasobie R1. Model z takim mutexem został przedstawiony na rysunku nr 10. Wyeliminowanie zjawiska poprzez dodanie kolejnego mutexu zwiększa ryzyko wystąpienia blokady jednakże, istnieje rozwiązanie lepsze tj. bez dodawania mutex5.



Rys. 10. Przykładowy model SWPC przykładowej aplikacji z rozwiązanym atomicity violation poprzez nadmiarowy mutex

Rozwiązanie, które pozwoli na eliminacje zjawiska atomicity violation bez dodawania mutex5 zostało przedstawione na rysunku nr 11. Rolę elementu synchronizującego pracę wątku B otrzymał mutex1, dzięki czemu można było usunąć nadmiarowy mutex. W momencie, gdy wątek B zaczyna pracę blokuje on możliwość pracy na współdzielonych zasobach wątkom A i D, aż do momentu gdy skończy pracę.

Zaprezentowane modele dla hipotetycznej aplikacji pokazują, że dzięki SWP w bardzo prosty sposób można zlokalizować nie tylko zjawisko race condition, ale także zjawisko atomicity violation. Dodatkowym atutem SWP jest jego czytelność, co pozwoliło na optymalizację polegającą na usunięciu nadmiarowego mutexu. Operacja ta wpłynie na szybkość aplikacji, ponieważ do wykonania jest mniej operacji blokowania i odblokowania, które potrafią być bardzo kosztowne.



Rys. 11. Przykładowy model SWPC przykładowej aplikacji z rozwiązaniem dla atomicity violation z minimalną ilością mutexów

Podsumowanie

Wszystkie 3 przedstawione reprezentacje posiadają swoje wady i zalety. W temacie aplikacji wielowątkowych CFG powinien być używany w momencie kiedy obiektem zainteresowania jest ilość bloków logicznych i kolejność ich wykonywania. Niestety CFG jest bardzo ogólną graficzną reprezentacją i nie nadaje się do analizy relacji między wątkami, bez dodatkowych informacji o poszczególnych blokach kodu, które są przedstawiane jako węzły.

Sieci Petriego są narzędziem dużo bardziej wyrafinowanym. Pozwalają pokazać mechanizm wzajemnego wykluczania i przepływ informacji. Jednakże poziom skomplikowania sieci będzie rósł wraz z poziomem skomplikowania aplikacji, a próba jej optymalizacji może spowodować ukrycie istotnych szczegółów. Dla każdej aplikacji wielowątkowej możliwe jest również stworzenie wielu różnych SP. Każda z sieci może działać dokładnie tak jak zakłada idea aplikacji wielowątkowej, natomiast żadna z nich nie będzie działać tak jak realna aplikacja, gdy w aplikacji występuje race condition.

Metoda wykorzystująca modele SWP wydaje się znacznie lepszym rozwiązaniem niż dwie poprzednie metody. SWP ukrywa większość szczegółów

implementacyjnych uwydatniając te miejsca, w których może wystąpić błąd race condition, atomicity violation czy deadlock, który podobnie do atomicity violation jest zjawiskiem wynikającym z niepoprawnego ustawienia mutexów. Interpretacja modelu SWP jest znacznie prostsza niż w przypadku SP czy CFG, a rozszerzenie notacji pozwoliło na zlokalizowanie miejsca wystąpienia błędu race condition w przykładowej aplikacji. Dodatkowo niewielka zmiana w modelu SWP pokazała jak należy rozwiązać problem race condition w przykładowej aplikacji czy atomicity violation w hipotetycznej aplikacji w rozdziale nr 4. Na niekorzyść metody wykorzystującej modele SWP przemawia fakt, że nie były one tworzone z myślą o aplikacja wielowątkowych, więc na potrzeby niniejszego artykułu należało rozszerzyć standardową notację, aby można było za jej pomocą wyrazić wszystkie niezbędne elementy aplikacji wielowątkowej.

Literatura

1. „A Brief History of Cilk”. b.d. https://www.cilkplus.org/cilk-history.
2. Aiken, Alex. b.d. „Charm++”. Udostępniono 28 październik 2017. https://web.stanford.edu/class/cs315b/lectures/lecture11.pdf.
3. Allen, Frances E. 1970. „Control Flow Analisys”. http://sumanj.info/secure_sw_devel/p1-allen.pdf.
4. Banaszak, Zbigniew, Paweł Majdzik, i Robert Wójcik. 2008. *Procesy współbieżne. Modele efektywności funkcjonowania*.
5. Bocewicz, Grzegorz. 2013. *Modele multimodalnych procesów cyklicznych*.
6. Bocewicz, Grzegorz, Zbigniew Banaszak, i Robert Wójcik. b.d. „Harmonogramowanie pracy wózków samojezdnych w warunkach ograniczonego dostępu do współdzielonych zasobów ESW”. Udostępniono 28 październik 2017. https://s3.amazonaws.com/academia.edu.documents/40428772/Harmonogramowane_pracy_wzkw_samojezdnych20151127-13023-1hjyvu1.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1509226185&Signature=dw0FK%2FEGMwDVjXYuoovI%2FpsG%2F9Q%3D&response-content-disposition=inline%3B%20filename%3DHarmonogramowane_pracy_wozkow_samojezdny.pdf.
7. Bull, J. Mark, Fiona Reid, i Nicola McDonnell. 2012. „OpenMP Application Programming Interface Examples”. W *International Workshop on OpenMP*, 271–274. Springer. https://link.springer.com/chapter/10.1007/978-3-642-30961-8_24.

8. Engler, Dawson, i Ken Ashcraft. 2003. „RacerX: effective, static detection of race conditions and deadlocks". W *ACM SIGOPS Operating Systems Review*, 37:237–252. ACM. http://dl.acm.org/citation.cfm?id=945468.
9. Hinnant, Howard E., Beman Dawes, Lawrence Crowl, Jeff Garland, i Anthony Williams. 2007. „Multi-threading Library for Standard C++". 24 czerwiec 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2320.html.
10. „Intel Threading Building Blocks Documentation". b.d. https://software.intel.com/en-us/tbb-documentation.
11. „Introduction to Charm++ Concepts". b.d. http://charmplusplus.org/tutorial/CharmConcepts.html.
12. „ISO/IEC 9945-1:2003 - Information technology -- Portable Operating System Interface (POSIX) -- Part 1: Base Definitions". b.d. Udostępniono 10 wrzesień 2017. https://www.iso.org/standard/38789.html.
13. Lu, Shan, Soyeon Park, Eunsoo Seo, i Yuanyuan Zhou. 2016. „Concurrency Testing Topics". kwiecień 6. http://www.it.uu.se/edu/course/homepage/conctest/vt16/testing-module1-lecture3.pdf.
14. Peterson, James L. 1977. „Petrie Nets". http://cs.rpi.edu/academics/courses/spring04/dci/peterson.pdf.
15. Silberschatz, Abraham, Piter B. Galvin, i Greg Gagne. 2005. *Postawy systemów operacyjnych*.
16. Torp, Kristian. 2002. „Multithreading in Java". listopad 19. http://people.cs.aau.dk/~torp/Teaching/E02/OOP/handouts/multithreading.pdf.
17. Voung, Jan Wen, Ranjit Jhala, i Sorin Lerner. 2007. „RELAY: static race detection on millions of lines of code". W *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 205–214. ACM. http://dl.acm.org/citation.cfm?id=1287654.

Streszczenie

Artykuł zawiera zestawienie reprezentacji graficznych, do których możliwa jest transformacja kodu źródłowego aplikacji wielowątkowych. Zestawienie powszechnie wykorzystywanych reprezentacji, jakimi są Control Flow Graph i Sieci Petriego, pozwoliło na analizę tych reprezentacji, pod kątem przydatności do znajdywania popularnych i niepożądanych zjawisk w aplikacjach wielowątkowych. Jako alternatywa dla Control Flow Graph i Sieci Petriego przedstawiono reprezentację Systemów Współbieżnych Procesów. Wszystkie trzy reprezentacje zostały wykorzystane do reprezentacji przykładowej aplikacji napisanej w języku C, zawierającej zjawisko race condition. W podsumowaniu dokonana została ocena,

która zależała od tego czy dana reprezentacja pozwoli odnaleźć wspomniane zjawisko.

Summary

The article contains a list of graphical representations to which it is possible to transform the source code of multithreaded applications. Comparison of commonly used representations, such as Control Flow Graph and Petri Network, allowed to analyze these representations in terms of their usefulness in finding popular and undesirable phenomena in multithreaded applications. As an alternative to Control Flow Graph and Petri Network, the representation of Concurrent Processing Systems is presented. All three representations were used to represent a sample C-language application containing race condition. In conclusion, an assessment was made, which depended on whether the representation would allow to find the phenomenon.