Przegląd metod synchronizacji w jądrach systemów uniksopodobnych oraz implementacja rogatek w systemie operacyjnym Mimiker

(An overview of synchronization methods in Unix-like kernels and an implementation of turnstiles in Mimiker OS)

Julian Pszczołowski

Praca licencjacka

Promotor: dr Piotr Witkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

11 września 2018

Streszczenie

W związku z tym że prawo Moore'a przestaje obowiązywać i pojedyncze procesory nie zwiększają swojej wydajności tak regularnie jak kiedyś, coraz większą uwagę przykłada się do przetwarzania współbieżnego i równoległego. Takie modele obliczeń powodują wiele problemów, głównie związanych z synchronizacją. W tej pracy wytłumaczymy dlaczego odpowiednia synchronizacja jest tak potrzebna. Przypomnimy najbardziej istotne zasady rządzące systemem operacyjnym i przejrzymy blokady dostępne dla programisty przestrzeni użytkownika. Skupimy się jednak na omówieniu środków synchronizacji w jądrach wybranych systemów uniksopodobnych, również pod względem standardowego przypadku użycia czy przykładowej implementacji. Na koniec prześledzimy naszą implementację rogatek w systemie operacyjnym Mimiker, wzorowaną na kodzie źródłowym FreeBSD.

As Moore's Law ends to apply and single processors don't increase their performance as regularly as in the past, parallel and concurrent processing is becoming more and more important today. Such models of computation imply a lot of synchronization-related problems. In this paper we'll explain why proper synchronization is crucial. We'll revise the core concepts of how an operating system works and have a glance at user-space locking. However, the emphasis will be put on the next part which is synchronization in Unix-like kernels. We will cover some details concerning implementation or typical use case. At the end we'll see the implementation of turnstiles in Mimiker OS based on the FreeBSD source.

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Dlaczego synchronizacja jest potrzebna?

W roku 1965 Gordon Moore, jeden ze współzałożycieli firmy Intel, postawił hipotezę mówiącą że liczba tranzystorów w najnowszych procesorach będzie się podwajała co roku [7]. Rzeczywiście, postęp technologiczny i stale zmniejszający się rozmiar tranzystorów pozwalał na tak szybki wzrost wydajności. Dziesięć lat później, w celu dostosowania do realiów, Moore zmienił okres z treści hipotezy z roku na dwa lata. W dzisiejszych czasach nie potrafimy już tak regularnie przyspieszać pojedynczych procesorów, natomiast potrafimy używać wielu procesorów w jednym systemie komputerowym. Przetwarzanie współbieżne i równoległe z roku na rok mają coraz większe znaczenie dla wydajności.

Uwaga. W tej pracy zakładamy, że czytelnik odbył standardowy uniwersytecki kurs systemów operacyjnych, w szczególności jest zaznajomiony z pojęciami takimi jak między innymi: proces, wątek, kwant czasu, czy zasób. Czytelnika chcącego przypomnieć sobie wyżej wymienione pojęcia odsyłamy do [1] lub [2].

Definicja. Przetwarzaniem współbieżnym nazywamy sytuację, w której różne obliczenia wydają się działać w tym samym czasie ([1], s. 176). Można uzyskać taki efekt uruchamiając na zmianę na jednym procesorze każde obliczenie na kwant czasu (ang. *time slice*), na przykład sto milisekund.

Definicja. Przetwarzaniem równoległym nazywamy sytuację, w której rzeczywiście różne obliczenia wykonują się w tym samym czasie ([1], s. 176).

Oczywiście zwykle takie obliczenia są ze sobą połączone – na przykład są to równoległe fazy jednego algorytmu, albo różne algorytmy wykonujące się na jednym komputerze, zatem będą współdzieliły pamięć. A zawsze jeżeli ma miejsce modyfikacja współdzielonej pamięci, potrzebna jest jej synchronizacja.

Przykład (Problemy w przypadku braku synchronizacji)**.** Przeanalizujmy co może się stać jeżeli uruchomimy współbieżnie dwa wątki, każdy z nich będzie wykonywał

funkcję `add50`, zakładając że zmienna `x` jest współdzielona i jej początkowa wartość to zero. [1]

```
1 extern volatile int x; /* Zmienna współdzielona. */
2
3 void add50() {
4   for (int i = 0; i < 50; i++)
5     x++;
6 }
```

Można by oczekiwać, że po skończeniu pracy obu wątków `x` będzie miało wartość sto, w końcu każdy wątek pięćdziesiąt razy wykonał `x++`. Niestety inkrementacja z linii piątej w rzeczywistości będzie wyglądała tak: [2]

```
1 movl x, %eax
2 addl $0x1, %eax
3 movl %eax, x
```

Następuje pobranie wartości zmiennej `x` z pamięci do rejestru `eax`, inkrementacja rejestru, a następnie zapisanie wartości rejestru do pamięci. Zauważmy że po wykonaniu pierwszego etapu, a przed wykonaniem trzeciego, inny wątek może wykonać dowolnie wiele instrukcji (na przykład zwiększyć `x` o kilkanaście), zatem nasza wczytana wartość będzie już nieaktualna.

Poniżej podajemy przeplot, w którym oba wątki wykonały po jednej inkrementacji, natomiast wartość zmiennej zwiększyła się tylko o jeden. Oczywiście każdy wątek ma swój zestaw rejestrów, czyli rejestr `eax` nie jest współdzielony.

```
1      wątek 1      |      wątek 2
2 ------------------+------------------
3  movl x, %eax     |
4                   | movl x, %eax
5  addl $0x1, %eax  |
6                   | addl $0x1, %eax
7  movl %eax, x     |
8                   | movl %eax, x
```

Istnieje również przeplot dwóch wątków, z których każdy pięćdziesiąt razy wykonuje funkcję `add50`, skutkujący ostateczną wartością zmiennej `x` równą dwa – dowód pozostawiamy jako ćwiczenie dla czytelnika.

Pisząc program wykonujący się w przestrzeni użytkownika można łatwo zapobiec opisanemu powyżej problemowi – wystarczy zastosować dostępny w naszym języku środek synchronizacji (w skrócie: blokadę), który umieści instrukcję `x++` w sekcji krytycznej. Przykłady blokad przypomnimy w podrozdziale 2.1. Gdybyśmy jednak

[1] Przykład został zainspirowany zadaniem z kursu *Systemy operacyjne (zaawansowane)* prowadzonego przez Krystiana Bacławskiego w Instytucie Informatyki Uniwersytetu Wrocławskiego.

[2] Użyty został kompilator gcc 7.3 z opcją `-Og` pod systemem Linux na architekturze x86-64.

tworzyli kod jądra, wybranie odpowiedniego mechanizmu synchronizacji wymagałoby o wiele większej uwagi, czego dowiemy się w rozdziałach 3. i 4. Natomiast sam problem możemy sklasyfikować jako sytuację wyścigu.

Definicja. Sekcja krytyczna to fragment kodu, który korzysta ze współdzielonych zasobów i w danej chwili może być wykonywany tylko przez jeden wątek ([1], s. 201).

Definicja. Sytuacja wyścigu (ang. *race condition*) to sytuacja w której więcej niż jeden wątek czyta i pisze do współdzielonego fragmentu pamięci, a ostateczny rezultat tych działań zależy od względnego czasowego przeplotu wykonania instrukcji przez te wątki ([1], s. 201).

1.2. Podstawowe fakty o funkcjonowaniu jądra systemu operacyjnego

Zacznijmy od omówienia modelu systemu operacyjnego, jaki będziemy rozważali w tej pracy. Wiedzę opieramy głównie na Linuksie i FreeBSD.

W systemie znajdują się wątki jądra oraz wątki użytkownika, każdy z nich posiada swój priorytet. Wątki użytkownika poprzez wywołania systemowe mogą zlecić do wykonania zadanie, które wykona się jako kod w przestrzeni jądra.

W jądrze również znajduje się planista, który decyduje o tym, jaki wątek należy uruchomić jako następny. Planista jest uruchamiany między innymi wtedy, gdy aktualnie wykonującemu się wątkowi minie przydzielony kwant czasu.

Jądro utrzymuje w pamięci wiele struktur danych, w tym struktury wątków, gdzie znajdziemy informacje o obecnym stanie wątku, priorytecie, czy też flagę mówiącą o tym, że dany wątek należy zatrzymać i zlecić uruchomienie innego.

W systemie występują przerwania sprzętowe – jest to sposób komunikacji zewnętrznych urządzeń z systemem. Przerwanie powoduje wstrzymanie obecnie działającego kodu i uruchomienie procedury obsługi przerwania (ang. *interrupt handler*).

Teraz przyjrzymy się bliżej niektórym tematom, a dokładniej:

1. Wejściu do jądra.
2. Kontekstom wykonywania kodu.
3. Wywłaszczaniu.
4. Dolnej i górnej połówce obsługi przerwania.
5. Dolnej i górnej połówce jądra.

Dlaczego akurat tym zagadnieniom? Wybór właściwych mechanizmów synchronizacji polega na wyeliminowaniu potencjalnych sytuacji wyścigu, które mogą być

spowodowane współbieżnym dostępem do współdzielonych danych z różnych kontekstów wykonania. Musimy zrozumieć w jakich momentach może zacząć wykonywać się kod jądra (1.), kod wątku (3.) lub kod obsługi przerwania (4.). Ponadto, jak dowiemy się w dalszej części pracy, podczas doboru blokad istnieją ograniczenia wynikające z rodzaju aktualnego kontekstu wykonania (2. i 5.).

1.2.1. Wejście do jądra

Procesor może zacząć wykonywać kod z przestrzeni jądra z trzech powodów ([3], s. 85-86).

- Przerwanie sprzętowe (ang. *hardware interrupt*).
- Pułapka sprzętowa (ang. *hardware trap*).
- Pułapka zainicjowana przez oprogramowanie (ang. *software-initiated trap*).

Przerwania sprzętowe to sposób na komunikację zewnętrznych urządzeń z systemem operacyjnym. Niektóre przerwania są niezbędne do prawidłowego działania systemu, na przykład te generowane przez zegar, pozwalające na odliczanie czasu i sprawiedliwy podział zasobów procesora pomiędzy wątki. Przerwania mają asynchroniczną naturę, zatem kod obsługi przerwania niekoniecznie jest powiązany z kodem, który aktualnie się wykonywał (i został przerwany).

Pułapki sprzętowe są natomiast powiązane z wątkiem, podczas wykonywania którego wystąpiły. Przykładami takich pułapek mogą być dzielenie przez zero, wystąpienie błędu strony podczas odwołania się do pamięci, czy wykonanie instrukcji `syscall` (architektura x86-64) służącej do zainicjowania wywołania systemowego.

Pułapka zainicjowana przez oprogramowanie służy do wymuszenia pilnej reakcji jądra nasze żądanie, na przykład dotyczące potrzeby ponownego uruchomienia planisty czy przetworzenia pakietu sieciowego. Będąc w przestrzeni użytkownika uzyskujemy taki efekt poprzez użycie wywołania systemowego (zatem wywołanie systemowe można zaliczyć zarówno do kategorii pułapki sprzętowej, jaki i pułapki inicjowanej przez oprogramowanie). Natomiast w przestrzeni jądra możemy ustawić odpowiednią flagę – będzie ona sprawdzona podczas próby wyjścia z jądra. Jeżeli flaga została ustawiona, zamiast wychodzić wykona się kod obsługi.

1.2.2. Konteksty wykonywania kodu

Konteksty, w jakich wykonuje się kod na procesorze pod kontrolą uniksopodobnego systemu operacyjnego, można podzielić na trzy rodzaje ([4], s. 5-6).

- Wykonywanie kodu w przestrzeni użytkownika, w kontekście wątku (ang. *thread context*) użytkownika.

- Wykonywanie kodu w przestrzeni jądra, w kontekście wątku, czyli w imieniu konkretnego wątku użytkownika lub jądra.

- Wykonywanie kodu (obsługi przerwania) w przestrzeni jądra, w kontekście przerwania (ang. *interrupt context*), czyli nie w imieniu żadnego konkretnego wątku. Przerwany mógł zostać dowolny wątek jądra lub użytkownika. Co będzie istotne z punktu widzenia dalszej części pracy, kod w kontekście przerwania nie może się zablokować, to znaczy, nie może oddać sterowania. Skoro do kodu nie jest przypisany żaden wątek, system nawet nie miałby jak nas obudzić ([4], s. 122). Przypomnijmy, że planista systemowy operuje na wątkach, ponieważ to właśnie one mają zapisany stan rejestrów, w tym informację o numerze kolejnej instrukcji do wykonania (ang. *instruction pointer*).

1.2.3. Wywłaszczanie

Gdyby przez dłuższy odcinek czasu prześledzić jaki kod jest wykonywany na procesorze, okazałoby się że (poza asynchronicznymi procedurami obsługi przerwań) system operacyjny najpierw zleca na pewien krótki czas wykonywanie jednego wątku, następnie po upłynięciu tego czasu zmienia wątek na inny, po kolejnej chwili znowu na inny. Każdemu wątkowi regularnie zostaje przydzielony kwant czasu (być może zależny od priorytetu), podczas którego wątek posuwa obliczenia do przodu.

Definicja. Wywłaszczeniem (ang. *preemption*) nazywamy zabranie wątkowi zasobu, którego jeszcze nie skończył używać ([1], s. 117). W tej pracy będziemy używać terminu wywłaszczenie w kontekście odebrania procesora. Może się to zdarzyć na przykład z powodu upłynięcia przydzielonego czasu lub pojawienia się w systemie wątku o wyższym priorytecie.

Powyższa definicja ma zastosowanie do systemów, które implementują wielozadaniowość poprzez wywłaszczanie (ang. *preemptive multitasking*), czyli dla wszystkich uniksopodobnych i większości współczesnych systemów operacyjnych ([4], s. 41).

Linux i FreeBSD w początkowych wersjach umożliwiały wyłącznie wywłaszczenie wątku wykonującego się w przestrzeni użytkownika – dowolny kod, który zaczął się wykonywać w przestrzeni jądra, musiał wykonać się do końca, czyli do powrotu do przestrzeni użytkownika lub do momentu zablokowania się (w oczekiwaniu na przykład na zdarzenie wejścia-wyjścia). Obecnie wątki mogą zostać wywłaszczone również podczas gdy wykonują kod znajdujący się w jądrze, co zwiększa responsywność systemu ([4], s. 63) [8].

W podrozdziale 3.5. zobaczymy, że wyłączenie wywłaszczania jest jednym ze środków synchronizacji w jądrze. Dowiemy się (na podstawie systemu Linux) kiedy dokładnie dochodzi do wywłaszczenia wątku użytkownika lub jądra, jak jest to zaimplementowane, i w jakim celu można wyłączyć ten mechanizm.

1.2.4. Dolna i górna połówka obsługi przerwania

W tym podrozdziale opiszemy w jaki sposób obsługa przerwania podzielona jest na dwa etapy (dwie połówki). W kolejnym użyjemy tej wiedzy do wprowadzenia podziału jądra na dolną i górną połówkę (ang. *bottom half, top half*). Bazujemy na terminologii FreeBSD (która jest istotnie różna od Linuxowej).

Gdy w systemie występuje przerwanie sprzętowe, na przykład od karty sieciowej – sygnalizujące pojawienie się nowego pakietu, przerywany jest obecnie wykonywany kod i uruchamiana jest procedura obsługi przerwania. Na czas jej wykonania przerwania z danej linii (ang. *interrupt line*) wyłączane są na wszystkich procesorach ([4], s. 134) [3], zatem kolejne pakiety od karty sieciowej nie dochodzą do systemu. Widzimy, że zbyt długie (pod względem czasu wykonania) procedury obsługi mogą skutkować utratą danych. W jaki sposób szybko przyjąć informację o nowym pakiecie, przekopiować go do systemu, a następnie dostarczyć czekającemu procesowi?

Pełna obsługa przerwania dzielona jest na dwie połowy. Pierwsza („dolna") obejmuje najistotniejsze rzeczy, które trzeba zrobić od razu. W naszym przykładzie byłoby to przekopiowanie danych z wewnętrznej pamięci karty sieciowej do systemu, a następnie ponowne włączenie mechanizmu przyjmowania przerwań, by umożliwić przychodzenie kolejnych pakietów. Natomiast przetworzeniem odebranych danych, czyli rzeczą która nie musi być wykonana w tym momencie, system zajmuje się w drugiej („górnej") połówce pełnej obsługi, zwykle w niedalekiej przyszłości.

W systemie FreeBSD za obsługę górnej, mniej krytycznej czasowo połówki obsługi przerwania odpowiedzialny jest mechanizm przerwań programowych (ang. *software interrupt*) ([3], s. 93). Dolna (przerwanie sprzętowe) połówka obsługi przerwania tworzy kolejkę zadań do wykonania przez górną (przerwanie programowe), czyli przez wątki jądra działające z wyższym priorytetem niż jakiekolwiek wątki użytkownika.

Zatem jeżeli występuje przerwanie sprzętowe, system obsługuje je od razu by skrócić czas podczas którego wyłączone jest przyjmowanie kolejnych przerwań i by nie gubić informacji od urządzeń. Jeżeli urządzenia akurat nie zgłaszają przerwań, a w systemie znajdują się możliwe do uruchomienia wątki przerwań programowych, procesor zostaje przydzielony temu o największym priorytecie. Jeśli nie występują przerwania sprzętowe i cała oddelegowana praca została już przetworzona (nie ma wątku przerwania sprzętowego, który można by uruchomić), procesor zostaje przydzielony wątkowi użytkownika o najwyższym priorytecie.

Dzięki temu, gdy do systemu dociera przez pewien czas dużo pakietów, przekopiowanie danych z karty sieciowej następuje praktycznie od razu po zgłoszeniu przerwania (w tej fazie przyjmowanie nowych przerwań sprzętowych jest wyłączone), przetwarzanie danych następuje w przerwach pomiędzy kopiowaniem (w tej fazie

[3] Kolejne przerwania tego samego typu wciąż można kolejkować, a brak zagnieżdżonych procedur obsługi znacząco upraszcza ich implementację ([4], s. 119).

przyjmowanie nowych przerwań sprzętowych jest włączone), a gdy wszystko już będzie zrobione, procesor zostanie przydzielony wątkowi użytkownika, który zobaczy gotowe pakiety.

Warto dodać, że programista jądra może w dowolnym momencie wyłączyć mechanizm przyjmowania nowych przerwań, zaburzając opisaną powyżej hierarchię. Jest to jeden ze środków synchronizacji, który opiszemy dokładniej w podrozdziale 3.3.

1.2.5. Dolna i górna połówka jądra

Podział jądra FreeBSD na dwie połówki jest następujący: dolna połówka to kody obsługi przerwań sprzętowych, a górna połówka to cała reszta, w tym przerwania programowe czy funkcje uruchamiane gdy użytkownik zainicjuje wywołanie systemowe ([3], s. 86-87).

Dolna połówka jest wykonywana w kontekście przerwania, a górna w kontekście wątku, zatem może na przykład zablokować się w oczekiwaniu na jakieś zdarzenie.

Warto też przypomnieć z poprzedniego podrozdziału, że w takim razie dolna połówka jądra wykonuje się z (przynajmniej częściowo) wyłączonym mechanizmem przyjmowania kolejnych przerwań sprzętowych, a w górnej połówce mechanizm ten jest włączony.

Rozdział 2.

Przestrzeń użytkownika

Zanim przejdziemy do meritum pracy, czyli omówienia mechanizmów synchronizacji w przestrzeni jądra, opiszmy pokrótce co z punktu widzenia synchronizacji dzieje się w przestrzeni użytkownika.

2.1. Przypomnienie środków synchronizacji dostępnych w przestrzeni użytkownika

Pisząc program w systemie zgodnym ze standardem wątków POSIX (ang. *POSIX threads* albo *pthreads*), przykładowo w języku C pod Linuxem, mamy do dyspozycji kilka rodzajów blokad, między innymi:

- Muteks (ang. *mutex*) [9].
- Semafor (ang. *semaphore*) [10].
- Blokada współdzielona (ang. *rwlock* albo *read/write lock*) [11].
- Zmienna warunkowa (ang. *condition variable*) [12].
- Bariera (ang. *barrier*) [13].
- Blokada wirująca (ang. *spinlock*) [14].

Muteks to najprostszy mechanizm wprowadzenia sekcji krytycznej. Jest to obiekt, który może znajdować się w jednym z dwóch stanów: zablokowanym, gdy muteks jest posiadany przez pewien wątek, i odblokowanym, gdy muteks nie ma właściciela. Wchodząc do sekcji krytycznej wystarczy że wywołamy funkcję `pthread_mutex_lock`, która albo przydzieli nam muteks od razu, albo zablokuje obecny wątek do czasu aż muteks będzie wolny. Wychodząc z sekcji krytycznej wołamy `pthread_mutex_unlock`. Widzimy, że nie może wystąpić sytuacja, w której dwa wątki znajdują się w sekcji krytycznej.

Semafor reprezentowany jest przez liczbę naturalną (włącznie z zerem). Możemy albo zwiększyć tę liczbę o jeden (wołając `sem_post`) albo zmniejszyć o jeden (wołając `sem_wait`). Próba zmniejszenia wartości semafora poniżej zera skutkuje zablokowaniem obecnego wątku do momentu aż dekrementacja będzie możliwa (jego wartość będzie dodatnia). Czym zatem różni się semafor binarny (semafor z wartością początkową równą jeden) od muteksu? W przypadku semafora zmniejszenie i zwiększenie wartości może zostać wykonane przez dowolny wątek. Muteks natomiast musi być odblokowany przez wątek, który go zajął.

Blokada współdzielona wprowadza nowy rodzaj sekcji krytycznej: w jej środku może znajdować się albo jeden wątek piszący, albo wiele wątków czytających. Wątek chcący czytać, wywołując `pthread_rwlock_rdlock`, wejdzie do sekcji krytycznej lub zostanie zablokowany do czasu aż w sekcji nie będzie wątku piszącego. Wątki chcące pisać wywołują `pthread_rwlock_wrlock`. Wyjście z sekcji krytycznej realizowane jest przez `pthread_rwlock_unlock`. Faworyzowanie wątków piszących lub czytających w dostępie do blokady jest zależne od aktualnej polityki systemowej albo od implementacji, w przypadku braku takiej polityki.

Zmienne warunkowe służą do oczekiwania na pewien warunek. Najlepiej zobrazuje to przykład: wątek A podczas działania wiele razy modyfikuje wartość zmiennej `x`, a wątek B czeka na moment gdy zmienna `x` będzie miała wartość zero. Możemy stworzyć zmienną warunkową `x_is_zero`. Wątek B zablokuje się na niej za pomocą funkcji `pthread_cond_wait`. W momencie gdy wątek A zmieni wartość `x` na zero, może wywołać `pthread_cond_signal` by obudzić jeden, lub `pthread_cond_broadcast`, by obudzić wszystkie wątki czekające na `x_is_zero`. Zmiennych warunkowych należy używać w sekcji krytycznej chronionej przez muteks, by uniknąć sytuacji wyścigu, w której wątek B widzi, że `x` ma niezerową wartość, zatem przygotowuje się do zablokowania, a w międzyczasie A zasygnalizuje `x_is_zero`, czego B już nie zauważy i zostanie zablokowany na dowolnie długi czas.

Bariery pozwalają na synchronizowanie wielu wątków naraz. Przy tworzeniu bariery ustalamy jej wartość `count`. Wywołaniem `pthread_barrier_wait` blokujemy wątek, dopóki łącznie `count` wątków nie zawoła `pthread_barrier_wait`. Ten mechanizm może się przydać na przykład gdy implementujemy iteracyjny algorytm, którego każda faza wykonywana jest przez kilka wątków, i chcemy się upewnić że nie przejdziemy do kolejnej fazy zanim wszystkie wątki nie skończą obliczeń – wtedy `count` ustawiamy na liczbę wątków i każdy z nich po zakończeniu fazy woła `pthread_barrier_wait`.

Blokada wirująca w użyciu podobna jest do muteksu, to znaczy pozwala nam stworzyć sekcję krytyczną dostępną wyłącznie dla jednego wątku. Analogicznie wołamy funkcję `pthread_spin_lock` przy wejściu, a `pthread_spin_unlock` przy wyjściu z sekcji. Różnica polega na tym, że gdy blokada nie jest dostępna, zamiast zablokować się (zatem zostać wywłaszczonym), aktywnie czekamy (ang. *busy waiting*), zużywając zasoby procesora. Co ciekawe, mechanizm blokady wirującej w przestrzeni użyt-

kownika jest opcjonalny (jest opcjonalną częścią standardu wątków POSIX [15]). Natomiast, jak dowiemy się w podrozdziale 3.4., jest niezbędny do implementacji jądra systemu.

2.2. Uwagi na temat implementacji

2.2.1. Operacje atomowe

Aby przybliżyć sposób w jaki zostały zaimplementowane blokady w przestrzeni użytkownika, najpierw musimy wprowadzić pojęcie operacji atomowej.

Definicja. Operacją atomową nazywamy ciąg instrukcji, który wydaje się być niepodzielny, to znaczy inne wątki nie mogą zobaczyć stanu pośredniego całej operacji ani jej przerwać. Operacja atomowa może albo wykonać się cała, albo nie wykonać się wcale, nie wpływając na stan systemu ([1], s. 201).

Niektóre problemy wynikające z braku atomowości inkrementacji w języku C opisaliśmy w podrozdziale 1.1.

Zazwyczaj nie jesteśmy w stanie sami zaimplementować danej akcji atomowo i potrzebujemy do tego wsparcia sprzętowego. W architekturze x86-64 do tego celu służy prefiks `LOCK`. Można go stosować do instrukcji, których operandem wyjściowym jest komórka pamięci. Na przykład `LOCK SUB` atomowo wykona odejmowanie [16].

Istnieją specjalne typy instrukcji atomowych wykorzystywanych do implementacji mechanizmów synchronizacji, między innymi:

- Compare-and-swap (w skrócie CAS),
- Test-and-set (w skrócie TAS),
- Load-link/store-conditional (w skrócie LL/SC).

Semantyka operacji CAS jest następująca:

```
1 bool compare_and_swap(int *ptr, int old, int new) {
2   if (*ptr != old)
3     return false;
4
5   *ptr = new;
6   return true;
7 }
```

Pseudokod operacji TAS to:

```
1 int test_and_set(int *ptr) {
2     int old = *ptr;
```

```
3       *ptr = 1;
4       return old;
5   }
```

Natomiast LL/SC jest zestawem dwóch instrukcji, używanych po sobie. Instrukcja LL zwraca zawartość komórki pamięci, natomiast SC zapisuje do tej komórki nową, podaną przez nas wartość, jeżeli od czasu wykonania LL komórka pamięci nie była modyfikowana.

Oczywiście gdyby skompilować podany wyżej kod, otrzymane funkcje nie byłyby atomowe. Na szczęście sprzęt dostarcza nam niektóre z tych operacji w formie atomowej. Przykładowo CAS jest dostępne w architekturze x86-64 w postaci instrukcji `LOCK CMPXCHG` (ang. *compare and exchange*, inna nazwa na *compare and swap*). Pisząc kod w C, zamiast używać wstawki asemblerowej, możemy wykorzystać gotową nakładkę GCC w postaci funkcji `__sync_bool_compare_and_swap` [21]. Zestaw instrukcji LL/SC znajdziemy między innymi w architekturze MIPS32 [19].

Zauważmy, że w przestrzeni użytkownika możemy używać operacji atomowych. Nie jest wymagane wywołanie systemowe i przejście do przestrzeni jądra.

2.2.2. Futeks

Wszystkie opisane w tym rozdziale metody synchronizacji przestrzeni użytkownika, oprócz blokad wirujących, czyli muteksy, semafory, zmienne warunkowe, bariery i blokady współdzielone, zaimplementowane są w systemie Linux za pomocą wywołania systemowego `futex` (skrót od ang. *fast user-space mutex*) [17]. Jest to wywołanie specyficzne dla Linuxa, dodane w wersji 2.6 ([6], s. 692), choć zostało zaimplementowane również w systemie FreeBSD, by umożliwić uruchamianie bez zmian programów napisanych i skompilowanych na Linuxa [18]. Jako ciekawostkę można dodać, że futeks jest wymieniony jako jeden z głównych konceptów jądra Zircon systemu Fuchsia, rozwijanego od niedawna przez firmę Google [20].

Skąd *fast* w nazwie? Zaimplementowanie mechanizmów synchronizacji za pomocą futeksa znacznie przyspiesza ich działanie w przypadku, w którym nie jest wymaganie zablokowanie wątku. Zademonstrujemy to za pomocą poniższego pseudokodu operacji `mutex_lock` i `mutex_unlock`:

```
1   /* Stan muteksu:
2    *  1 - wolny,
3    *  0 - zajęty. */
4   int state = 1;
5
6   void mutex_lock() {
7     while (true) {
8       /* Czy muteks jest wolny? */
9       if (__sync_bool_compare_and_swap(&state, 1, 0))
```

```
10        break; /* Tak. */
11      else
12        futex(&state, FUTEX_WAIT); /* Nie, zablokuj się. */
13    }
14  }
15
16  void mutex_unlock() {
17    if (__sync_bool_compare_and_swap(&state, 0, 1))
18      futex(&state, FUTEX_WAKE); /* Obudź czekające wątki. */
19  }
```

Widzimy, że operacja `mutex_lock` w przypadku, gdy muteks jest wolny, nie używa żadnego wywołania systemowego (zatem wątek nie wchodzi do przestrzeni jądra, co generowałoby narzut czasowy) – atomowo zmieniamy wartość zmiennej `state` z jedynki na zero. Jeżeli muteks jest zajęty, używamy wywołania `futex` – tego nie można uniknąć, ponieważ bezpośrednio w przestrzeni użytkownika nie mamy możliwości zablokowania wątku.

Rozdział 3.

Niskopoziomowe środki synchronizacji w jądrze

Omówimy teraz niskopoziomowe mechanizmy synchronizacji, które dostępne są dla programistów jądra. Ich implementacja zazwyczaj jest w dużej mierze zależna od konkretnej architektury. Na tych mechanizmach opierać się będą bardziej wysokopoziomowe środki, które umówimy w rozdziale 4. Warto dodać, że niskopoziomowa synchronizacja jest kluczowa i szeroko wykorzystywana w systemie Mimiker, o którym dowiemy się więcej w rozdziale 5.

3.1. Operacje atomowe

Operacje atomowe zostały umówione w podrozdziale 2.2.1., ponieważ były potrzebne do implementacji blokad w przestrzeni użytkownika, lecz dla kompletności tego rozdziału wspominamy o nich również tutaj.

3.2. Bariery pamięciowe i bariera kompilatora

Większość współczesnych procesorów obsługuje wykonywanie kodu poza kolejnością (ang. *out-of-order execution*). Jest to technika pozwalająca wykorzystać większą część zasobów procesora, na przykład jednostek funkcyjnych (ang. *functional unit*), poprzez przetwarzanie instrukcji w kolejności niekoniecznie takiej, jaką zlecił programista. Ta optymalizacja jest niewidoczna dla wykonywanego programu, jednakże może być widoczna dla zewnętrznego obserwatora, śledzącego stan modyfikowanej pamięci. W przypadku architektury wieloprocesorowej (ang. *symmetric multiprocessing*) takim obserwatorem może być inny procesor.

Prześledźmy przykładowe równoległe wykonanie dwóch programów na dwóch procesorach. Początkowe wartości zmiennych `a` i `b` są równe odpowiednio jeden i dwa.

```
 wątek 1 | wątek 2
---------+---------
 a = 3;  |   ---
 b = 4;  | c = b;
  ---    | d = a;
```

Na niektórych architekturach z wykonywaniem poza kolejnością możliwe jest że w wątku drugim zmienna `c` otrzyma nową wartość `b`, czyli cztery, a zmienna `d` starą wartość `a`, czyli jeden ([4], s. 204).

Aby zapobiec takiemu scenariuszowi, procesory oferują nam specjalne instrukcje zwane barierami pamięciowymi. W jądrze Linux dostępne są między innymi następujące wywołania:

- `rmb()`, czyli bariera odczytu (ang. *read memory barrier*),
- `wmb()`, czyli bariera zapisu (ang. *write memory barrier*),
- `mb()`, czyli bariera zarówno odczytu jak i zapisu (ang. *memory barrier*).

Bariera odczytu zapewnia, że żadne odczyty pamięci nie zostaną przeniesione przez procesor przez barierę. To znaczy, żaden odczyt znajdujący się w kodzie programu przed wywołaniem `rmb()` nie zostanie wykonany po tym wywołaniu, ani żaden odczyt znajdujący się po wywołaniu `rmb()` nie zostanie wykonany przed tym wywołaniem. Bariera zapisu działa analogicznie.

Dodajmy do kodu dwóch wątków odpowiednie bariery.

```
 wątek 1 | wątek 2
---------+---------
 a = 3;  |   ---
 mb();   |   ---
 b = 4;  | c = b;
  ---    | rmb();
  ---    | d = a;
```

Teraz, zgodnie z naszymi oczekiwaniami, jeżeli zmienna `c` otrzyma już nową wartość wartość `b` (cztery), to zmienna `d` również otrzyma nową wartość `a` (trzy).

Poza problemami związanymi z dynamiczną zmianą kolejności wykonania instrukcji przez procesor, istnieje również problem statycznej zmiany kolejności, którą ma prawo dokonać kompilator w celu optymalizacji. Możliwa jest zamiana miejscami odczytów i zapisów do pamięci tak aby nie wpływało to na efekt działania kodu w języku C. Jednakże kompilator nie jest świadomy zjawisk, które dzieją się w innym kontekście niż obecnie poddawany translacji. Czasami zatem zmiana kolejności instrukcji jest niepożądana, na przykład gdy współdzielimy dane z kodem obsługi przerwania ([4], s. 205) – taki kod może przerwać nasze wykonanie i sprawdzić

stan pamięci w dowolnym momencie, dlatego czasem ważne jest w jakiej kolejności wykonamy teoretycznie dwie niezależne od siebie instrukcje `x = 42;` i `y = 43;`.

Aby zapobiec zamianie miejscami zapisów i odczytów przez kompilator, jako programista jądra Linux możemy użyć wywołania `barrier()`. Działa ono na podobnej zasadzie jak `mb()`, czyli nie pozwala aby zapis lub odczyt umiejscowiony przed wywołaniem `barrier()` został przeniesiony po tym wywołaniu, i na odwrót.

Wymienione przez nas wcześniej bariery pamięciowe zawierają w sobie barierę kompilatora [22], zatem w wielu przypadkach nie trzeba jej bezpośrednio używać.

3.3. Wyłączanie przerwań

Mechanizm wyłączania przerwań służy do synchronizowania danych współdzielonych na jednym procesorze przez dolną i górną połówkę jądra.

Kod wykonujący się w dolnej połówce jądra (kod obsługi przerwania) zawsze działa z wyłączonym mechanizmem przyjmowania nowych przerwań. Natomiast kod wykonujący się w górnej połówce, jeżeli współdzieli dane z dolną połówką, nie może pozwolić by nagłe wywołanie przerwania zastało te dane w niespójnym stanie. Dlatego też kod w górnej połówce jądra, na czas modyfikacji współdzielonych danych, musi wyłączyć w procesorze mechanizm przerwań (procesor będzie ignorował przychodzące przerwania).

Implementacja tego mechanizmu jest zależna od konkretnej architektury. W procesorach x86-64 można wejść do sekcji ignorowania przerwań wywołując instrukcję `CLI` (ang. *clear interrupt flag*), a wyjść z niej instrukcją `STI` (ang. *set interrupt flag*) [16].

Warto dodać, że jeżeli rozważamy system wieloprocesorowy, wyłączenie przerwań może nie być wystarczające do wyeliminowania sytuacji wyścigu pomiędzy dolną i górną połówką jądra – nie zabezpiecza nas to przed scenariuszem w którym inny procesor będzie równocześnie modyfikował współdzielone dane, jako że przerwania wyłączamy tylko lokalnie.

3.4. Blokada wirująca

W jądrze, w przeciwieństwie do przestrzeni użytkownika, blokada wirująca jest niezbędna. Służy do synchronizacji danych pomiędzy procesorami.

Wątek, który spróbuje wziąć zajętą blokadę wirującą, przechodzi w stan aktywnego czekania, zużywając zasoby procesora aż blokada będzie dostępna. Dlaczego w jądrze potrzebujemy tego mechanizmu? Poza oczywistym wytłumaczeniem, że służy on do budowy bardziej wysokopoziomowych blokad typu muteks, jest on je-

dynym wyjściem w przypadku gdy synchronizujemy dane pomiędzy dolną połówką jądra działającą na jednym procesorze, a jądrem działającym na drugim procesorze. Przyczyną jest brak możliwości uśpienia wątku znajdującego się w dolnej połówce jądra – pozostaje aktywnie czekać, aż drugi procesor wyjdzie z sekcji krytycznej. W przestrzeni użytkownika natomiast zawsze możemy przejść w stan uśpienia.

Jeżeli jest możliwość, że wątek A wykonujący kod zabezpieczony blokadą wirującą zostanie przerwany przez kod, który również spróbuje wziąć tę samą blokadę wirującą, należy dodatkowo zastosować wyłączenie przerwań. W przeciwnym wypadku kod obsługi przerwania zacząłby aktywnie czekać aż A zwolni blokadę, co oczywiście nigdy się nie stanie ([4], s. 185). Zauważmy, że musimy wyłączyć przerwania jedynie na lokalnym procesorze. Próba wzięcia zajętej blokady wirującej przez przerwanie na innym procesorze nie prowadzi do zakleszczenia, ponieważ właściciel blokady (nasz procesor) niedługo ją zwolni.

Synchronizacja w postaci blokady wirującej i wyłączenia przerwań jest zawsze wystarczająca [23], choć oczywiście zwykle nadmiarowa. Powinniśmy używać takiej pary tylko gdy jest to naprawdę potrzebne, jako że zbyt długi okres z wyłączonymi przerwaniami obniża responsywność systemu.

Blokadę wirującą implementuje się głównie za pomocą języka asemblera, mocno zależnego od konkretnej architektury ([5], s. 354), zwykle używając operacji atomowych i barier pamięciowych [24] [25].

3.5. Wyłączanie wywłaszczania

Jak już wspomnieliśmy w podrozdziale 1.2.3., wywłaszczanie to zabranie wątkowi procesora, pomimo że jeszcze nie skończył go używać. Prześledźmy na podstawie systemu Linux, kiedy wątek użytkownika lub jądra może zostać wywłaszczony ([4], s. 62-64). Dodajmy, że podobnie zostało to zaimplementowane w systemie FreeBSD [26]. Podczas pracy systemu, obecnie działający wątek może zostać oznaczony flagą `need_resched`, jeżeli:

- funkcja `scheduler_tick()`, wywoływana przez przerwania zegarowe, wykryje że skończył się kwant czasu,
- pojawił się w systemie możliwy do uruchomienia wątek o wyższym priorytecie.

Gdy jądro wykryje obecność flagi `need_resched`, nastąpi wywłaszczenie obecnego wątku, wybranie nowego i zmiana kontekstu. Sprawdzenie tej flagi odbywa się w kilku szczególnych miejscach, opisanych poniżej.

Do wywłaszczenia wątku użytkownika może dojść gdy jądro próbuje wejść do przestrzeni użytkownika wracając z procedury obsługi przerwania lub wracając z procedury obsługi wywołania systemowego. Tuż przed powrotem do wątku

system sprawdza obecność `need_resched` i jeżeli flaga jest ustawiona, wątek zostaje wywłaszczony.

Do wywłaszczenia wątku jądra może dojść gdy:

- wątek bezpośrednio o to poprosi wywołując `schedule()`,
- wracamy z procedury obsługi przerwania i ustawiona jest flaga `need_resched`,
- wątek staje się znów wywłaszczalny (można wyłączyć na pewien czas możliwość wywłaszczenia wątku) i ustawiona jest flaga `need_resched`,
- wątek w jądrze się zablokuje.

Wątek jądra może wyłączyć możliwość bycia wywłaszczonym ustawiając odpowiednią zmienną w swojej strukturze. W takim wypadku obecność flagi `need_resched` nie spowoduje zmiany kontekstu. Flaga zostanie sprawdzona automatycznie przy włączeniu wywłaszczania.

Samo wyłączanie wywłaszczania służy za kolejny niskopoziomowy środek synchronizacji. Należy go używać gdy dane współdzielone są przez wątki (zatem kod górnej połówki jądra) działające na jednym procesorze, a bardziej wysokopoziomowe blokady typu muteks nie są dostępne. [1]

Istnieją również inne przypadki użycia omawianego mechanizmu [27]. Na architekturze wieloprocesorowej, podczas korzystania ze struktur indywidualnych dla każdego z procesorów, może wydarzyć się taka sytuacja:

```
1 struct_t per_cpu[NR_CPUS];
2
3 per_cpu[smp_processor_id()] = x;
4 /* wywłaszczenie */
5 y = per_cpu[smp_processor_id()];
```

Jeżeli dopuścimy do wywłaszczenia w zaznaczonym miejscu, może okazać się, że drugie wywołanie funkcji `smp_processor_id()` zwróci inną wartość, jako że nasz wątek zostanie uruchomiony na innym procesorze.

3.6. Kolejki uśpionych wątków

W tym podrozdziale zaczerpniemy terminologię z systemu FreeBSD. Kolejki uśpionych wątków (ang. *sleepqueue*) implementują w jądrze stan nieograniczonego uśpienia wątków ([3], s. 132).

Definicja. Stanem nieograniczonego uśpienia wątku nazywamy stan, w którym wątek czeka na zewnętrzne zdarzenie [28].

[1] Użycie muteksu nie jest możliwe na przykład w kodzie implementującym muteks.

Z każdą kolejką związany jest kanał oczekiwania (ang. *wait channel*), identyfikujący tę kolejkę. Wszystkie wątki, które przechodzą w stan nieograniczonego uśpienia używając tego samego kanału, trafiają do jednej kolejki. Aby wybudzić jeden albo wszystkie wątki z kolejki, należy znać jej kanał (zwykle jest to liczba lub pewien wskaźnik konwertowany na liczbę).

W jądrze FreeBSD za pomocą kolejek uśpionych wątków zaimplementowane są między innymi zmienne warunkowe i wywołania `sleep(9)`, `wakeup(9)`, `pause(9)`. Podręcznik systemowy nie zaleca wykorzystywania kolejek uśpionych wątków w innych miejscach jądra, jako że jest to zbyt niskopoziomowy mechanizm [29].

W interfejsie jądra FreeBSD dostępne są parametry nieograniczonego stanu uśpienia. Możemy wybrać, czy stan uśpienia wątku A ma być przerywalny (ang. *interruptible sleep*), czyli czy inny wątek będzie mógł bezpośrednio wybudzić wątek A (na przykład wysyłając sygnał SIGKILL) nie odwołując się do kolejki i jej kanału oczekiwania, oraz czy stan uśpienia ma mieć limit czasowy, po którym wątek zostanie automatycznie wybudzony.

3.7. Rogatki

W tym podrozdziale również zaczerpniemy terminologię z systemu FreeBSD. Rogatki (ang. *turnstile*) implementują w jądrze stan ograniczonego uśpienia wątków ([3], s. 130).

Definicja. Stanem ograniczonego uśpienia wątku (w oczekiwaniu na blokadę) nazywamy stan, w którym jedyną rzeczą potrzebną do wznowienia działania jest przyznanie pewnej ilości czasu procesora dla wątku, który jest w posiadaniu tej blokady [28].

Rogatki to mechanizm bardzo zbliżony do kolejek uśpionych wątków. Jest wykorzystywany w implementacji bardziej krytycznych czasowo blokad, w których wiemy że wątek znajdujący się w posiadaniu blokady niedługo ją zwolni, a nie czeka na zajście zewnętrznego zdarzenia asynchronicznego, na przykład na wciśnięcie przycisku na klawiaturze.

W jądrze FreeBSD za pomocą rogatek zaimplementowane są między innymi: muteks, blokada współdzielona typu RW i blokada współdzielona typu RM, o których dowiemy się więcej w rozdziale 4.

Zauważmy, że w takim razie wątek używający rogatek (na przykład będąc w posiadaniu muteksu) nie może oczekiwać na zewnętrzne zdarzenie. Jednakże w teorii istnieje scenariusz w którym nawet pomimo trzymania się tej reguły, czas oczekiwania na zwolnienie muteksu może być dowolnie długi. Wyobraźmy sobie sytuację w której w systemie istnieje wątek H o wysokim priorytecie, wątek L o niskim priorytecie będący w posiadaniu muteksu, oraz duża liczba wątków o średnim priorytecie. Jeżeli wątek H spróbuje wejść w posiadanie tego samego muteksu, zablokuje się w

oczekiwaniu na jego zwolnienie przez wątek L. Jednakże wątek L nie zostanie uruchomiony dopóki w systemie istnieją wątki o średnim priorytecie, które zatem zostaną uruchomione jako pierwsze.

Definicja. Odwróceniem priorytetów (ang. *priority inversion*) nazywamy sytuację w której okoliczności zachodzące w systemie wymuszają na wątku oczekiwanie na inny wątek o niższym priorytecie ([1], s. 458).

Na szczęście rogatki są odporne na odwrócenie priorytetów. Mechanizm, który zapobiega odwróceniu priorytetów, nazywa się dziedziczeniem priorytetów – używając poprzedniej notacji, jeżeli wątek H wejdzie w stan uśpienia w oczekiwaniu na muteks posiadany przez wątek L, priorytet H zostanie tymczasowo pożyczony wątkowi L. Wtedy wątek L szybko dokończy działanie, wyjdzie z sekcji krytycznej, tracąc pożyczony priorytet i wybudzając H. Zauważmy, że w takim scenariuszu wątki o średnim priorytecie zostaną uruchomione po wątku H.

Dziedziczenie priorytetów może odbyć się łańcuchowo. Jeżeli wątek H zablokuje się w oczekiwaniu na muteks A, którego właściciel jest zablokowany w oczekiwaniu na muteks B, którego właściciel jest zablokowany w oczekiwaniu na muteks C, wątek H pożyczy swój wysoki priorytet właścicielom muteksów A, B i C.

Aby dziedziczyć priorytety rogatki muszą posiadać informację o obecnym właścicielu blokady. Jest to główna różnica pomiędzy zawartością struktur kolejki uśpionych wątków i rogatki (w rogatkach tak samo jak w kolejkach obecny jest identyfikator zwany kanałem oczekiwania).

Dlaczego kolejki uśpionych wątków nie przeprowadzają dziedziczenia priorytetów? Jeżeli wątek uśpiony na kolejce oczekuje na zewnętrzne zdarzenie, zwiększenie mu priorytetu w systemie nie spowoduje, że to zdarzenie wydarzy się szybciej.

Rozdział 4.

Wysokopoziomowe środki synchronizacji w jądrze

> The kernel synchronization primitives interact and have a number of rules regarding how they can and can not be combined. There are too many for the average human mind and they keep changing. (if you disagree, please write replacement text) : -)
>
> FreeBSD 8.0 Kernel Developer's Manual, `locking(9)`

> **BUGS** There are too many locking primitives to choose from.
>
> FreeBSD 11.2 Kernel Developer's Manual, `locking(9)`

Większość środków synchronizacji opisanych w tym rozdziale działa na podobnej zasadzie jak odpowiadające im środki z przestrzeni użytkownika, które opisaliśmy w podrozdziale 2.1. Główna różnica pomiędzy nimi jest taka, że:

- implementacja blokad w jądrze jest znacznie trudniejsza – w przestrzeni użytkownika duża część czynności delegowana jest do systemu za pomocą wywołań systemowych,
- podczas używania blokad w jądrze można popełnić dużo błędów, które w przestrzeni użytkownika błędami nie są – na przykład błędem jest jeżeli wątek posiadający muteks przejdzie w stan nieograniczonego uśpienia.

Blokady z tego rozdziału, pochodzące z jądra systemu FreeBSD, możemy podzielić na:

- zaimplementowane za pomocą rogatek – wspierają dziedziczenie priorytetów; aby nie dopuścić do odwrócenia priorytetów podczas posiadania takiej blokady nie należy przechodzić w stan nieograniczonego uśpienia, nie należy próbować brać blokady zaimplementowanej za pomocą kolejek uśpionych wątków (aby nikt kto posiada albo czeka na blokadę zaimplementowaną za pomocą rogatek nie musiał czekać na zewnętrzne zdarzenie),
- zaimplementowane za pomocą kolejek uśpionych wątków – podczas ich posiadania można przechodzić w stan nieograniczonego uśpienia, można brać dowolną blokadę.

4.1. Muteks

W jądrze FreeBSD muteksy zaimplementowane są za pomocą rogatek, zatem mają wbudowane dziedziczenie priorytetów. W jądrze Linux istnieją dwa typy muteksów ([5], s. 363) – zwykły oraz rt-muteks (ang. *real-time mutex*), który różni się od zwykłego tym, że posiada mechanizm dziedziczenia priorytetów.

W obu jądrach muteksy domyślnie są adaptacyjne [28] [30], to znaczy że jeżeli próbujemy wziąć zajęty muteks, a jego właściciel jest aktualnie uruchomiony (na innym procesorze), zamiast zablokować się aktywnie czekamy przez krótką chwilę, w nadziei że muteks zostanie niedługo odblokowany – jeżeli tak się stanie, zyskujemy na czasie, bo proces zablokowania się jest względnie kosztowny. Jeżeli natomiast muteks nie zostanie zwolniony, blokujemy się (na przykład używając rogatek). Jest to optymalizacja dla przypadku w którym w jednej chwili tylko jeden wątek próbuje dostać się do sekcji krytycznej.

Muteksu nie możemy używać w kontekście przerwania, ponieważ w dolnej połówce nie możemy się zablokować. Podczas trzymania muteksu nie możemy przechodzić w stan nieograniczonego uśpienia. To znaczy, że nie możemy czekać na zewnętrzne zdarzenie ani nie możemy wziąć blokady, która została zaimplementowana za pomocą kolejek uśpionych wątków (zmienna warunkowa czy semafor).

Istnieją również rekurencyjne muteksy – właściciel muteksu może go pozyskać wiele razy, ale kończąc użycie powinien oddać go tyle razy, ile wszedł w jego posiadanie. Rekurencyjne muteksy są dostępne w jądrach systemów FreeBSD ([3], s. 139) i Mimiker, natomiast nie są dostępne w jądrze Linux [30].

Poniżej podajemy przykładową implementację muteksu pochodzącą z jądra systemu Mimiker. Jest to implementacja na system jednoprocesorowy, co bardzo upraszcza synchronizację. Gdyby Mimiker obsługiwał architekturę wieloprocesorową, wymagane byłoby użycie blokad wirujących.

```
1  void mtx_lock(mtx_t *m) {
2    if (mtx_owned(m)) {
3      /* muteks rekurencyjny */
4      assert(m->m_type == MTX_RECURSE);
5      m->m_count++;
6      return;
7    }
8
9    WITH_NO_PREEMPTION {
10     /* wywłaszczanie wyłączone */
11     while (m->m_owner != NULL) {
12       turnstile_wait(m, m->m_owner);
13     }
14
15     m->m_owner = thread_self();
16   }
17 }
```

Przy czym blokowanie z użyciem rogatek ma następującą sygnaturę:

```
1 /* Zablokuj obecny wątek W na kanale oczekiwania ‘wchan‘.
2  * Wątek ‘owner‘ jest właścicielem blokady, na którą
3  * czeka W. */
4 void turnstile_wait(void *wchan, thread_t *owner);
```

Więcej o implementacji rogatek dowiemy się w podrozdziale 5.2.

4.2. Blokady współdzielone we FreeBSD

4.2.1. RW

Blokada współdzielona typu RW (ang. *reader/writer lock*) zapewnia albo sekcję krytyczną współdzieloną przez wiele wątków czytających, albo sekcję krytyczną na wyłączność dla wątku piszącego.

Dziedziczenie priorytetów jest wspierane, ale priorytet jest pożyczany tylko dla wątku piszącego (gdy na blokadę czeka wątek czytający o wyższym priorytecie). Propagacja priorytetu nie zachodzi dla wątków czytających – są one anonimowe, czyli nie jest trzymana ich lista.

Blokada RW jest zaimplementowana za pomocą rogatek, zatem podczas jej posiadania nie można przechodzić w stan nieograniczonego uśpienia.

4.2.2. RM

Blokada współdzielona typu RM (ang. *read-mostly lock*) jest bardzo podobna do blokady RW, jednakże jest zoptymalizowana pod bardzo sporadyczne używanie w trybie pisania.

Wspiera dziedziczenie priorytetów, zarówno dla wątków piszących, jak i dla wątków czytających (zatem utrzymuje w pamięci listę wszystkich wątków obecnie używających blokady).

Blokada RM jest zaimplementowana za pomocą rogatek, zatem podczas jej posiadania nie można przechodzić w stan nieograniczonego uśpienia.

4.2.3. SRM

Blokada współdzielona typu SRM (ang. *sleepable read-mostly lock*) jest odmianą blokady RM. Priorytet propagowany jest jedynie wątkom czytającym. Wyłącznie wątek piszący może przejść w stan nieograniczonego uśpienia podczas posiadania blokady.

4.2.4. SX

Blokada współdzielona typu SX (ang. *shared/exclusive lock*) jest wariantem blokady RM, który nie wspiera dziedziczenia priorytetów. Została zaimplementowana za pomocą kolejek uśpionych wątków. Podczas posiadania blokady SX (niezależnie czy jako wątek piszący, czy czytający) możemy przejść w stan nieograniczonego uśpienia.

4.2.5. Wykorzystanie blokad współdzielonych w jądrze

Sumarycznie blokady współdzielone używane są w bardzo wielu miejscach w kodzie jądra FreeBSD. Z wymienionych powyżej najbardziej popularne są blokady typu SX i RW – pojawiają się w odpowiednio około stu czterdziestu [34] [35] i osiemdziesięciu [36] [37] plikach z kodem źródłowym. Blokadę typu RM znajdziemy w prawie trzydziestu plikach [38] [39], na przykład w `sys/netinet/tcp_fastopen.c`. Co ciekawe, blokada typu SRM występuje wyłącznie w kodzie `sys/kern/kern_sysctl.c` oraz `sys/dev/hyperv/netvsc/if_hn.c` [40].

Możemy zaobserwować, że blokady które wspierają propagowanie priorytetu wątkom czytającym (RM i SRM) są rzadziej używane przez programistów jądra. Przypomnijmy, że takie blokady potrzebują utrzymywać w pamięci listę wątków czytających, których z reguły jest dużo więcej niż piszących.

4.3. Zmienna warunkowa

Zmienna warunkowa w jądrze FreeBSD działa na podobnej zasadzie co zmienna warunkowa z biblioteki wątków POSIX opisana przez nas w podrozdziale 2.1. Warto wspomnieć, że została ona zaimplementowana za pomocą kolejek uśpionych wątków, zatem wątek oczekujący na warunek musi liczyć się z tym, że wchodzi w stan nieograniczonego uśpienia.

4.4. Semafor

Przypomnijmy, że semafor reprezentowany jest przez wartość, którą zmniejszamy o jeden wchodząc do sekcji krytycznej i zwiększamy o jeden wychodząc z sekcji krytycznej. Można myśleć, że jeżeli semafor ma początkowo wartość `n`, to w jednym momencie najwyżej `n` wątków może mieć dostęp do sekcji krytycznej. Wartość semafora może również, zamiast chronić dostępu do sekcji krytycznej, reprezentować liczbę dostępnych zasobów.

Semafor binarny podobny jest do muteksu, jednakże muteks w przeciwieństwie do semafora musi zostać odblokowany przez ten sam wątek, który wszedł w jego posiadanie.

W jądrze FreeBSD semafor został zaimplementowany za pomocą zmiennej warunkowej, zatem będąc w sekcji krytycznej możemy wejść w stan nieograniczonego uśpienia.

W jądrach zarówno Linuksa, jak i FreeBSD, semafor jako środek synchronizacji nie powinien być używany w żadnym nowym kodzie jeżeli można go zastąpić muteksem albo zmienną warunkową [31] ([4], s. 197).

4.5. Blokada sekwencyjna

Blokada sekwencyjna (ang. *sequential lock*) jest typem blokady współdzielonej występującej w jądrze Linux ([4], s. 200-201). Każda blokada sekwencyjna ma przypisaną zmienną typu całkowitego, początkowo o wartości zero. Wątek piszący inkrementuje tą zmienną podczas wchodzenia do sekcji krytycznej i podczas wychodzenia z sekcji krytycznej. Wątek czytający sprawdza wartość zmiennej przed rozpoczęciem czytania i po zakończeniu czytania – jeżeli w obu przypadkach wartość była ta sama (i była parzysta), odczyt uznajemy za poprawny. W przeciwnym razie wątek czytający ponawia tę procedurę.

Ten typ blokady jest użyteczny gdy wątków piszących jest niewiele, choć i tak są one faworyzowane (zauważmy że wątek piszący nigdy nie czeka na wątek czytający). Przykładem użycia jest synchronizacja dostępu do 64-bitowej zmiennej `jiffies`,

która trzyma liczbę cykli zegara, które upłynęły od uruchomienia systemu. Synchronizacja jest potrzebna w architekturach, które nie wspierają atomowych odczytów i zapisów do zmiennych 64-bitowych.

Aktualizacja `jiffies` w kodzie przerwania zegarowego wygląda następująco:

```
1 write_seqlock(&lock);
2 jiffies_64++;
3 write_sequnlock(&lock);
```

Natomiast tak wygląda pobranie wartości:

```
1 u64 get_jiffies_64(void) {
2   u64 ret;
3   do {
4     unsigned long seq = read_seqbegin(&lock);
5     ret = jiffies_64;
6   } while (read_seqretry(&lock, seq));
7
8   return ret;
9 }
```

4.6. Blokada RCU

Blokada RCU (ang. *Read-Copy-Update*) pochodzi z systemu Linux i jest kolejną odmianą blokady współdzielonej. Po wprowadzeniu do jądra została pozytywnie przyjęta przez społeczność, a obecnie jest używana w wielu miejscach systemu ([5], s. 357). Jest wydajna, choć generuje narzut pamięciowy, zwykle zaniedbywalny. Ograniczenia, jakie są z nią związane, to:

- modyfikacja danych powinna zachodzić relatywnie rzadko,
- wątek wykonujący kod w sekcji chronionej przez blokadę RCU nie może się zablokować,
- dostęp do struktury danych chronionej przez blokadę RCU musi zachodzić przez dereferencję wskaźnika.

W sytuacji gdy wątek piszący chce zmienić dane na inne, tworzy nową strukturę, a po wypełnieniu odpowiednimi wartościami *publikuje* nowy wskaźnik. W pamięci istnieją dwie kopie danych, nowa i stara. Wątki które zaczęły czytać po publikacji zobaczą nowe dane, natomiast wątki, które uzyskały dostęp do danych przed modyfikacją, widzą stare dane. Gdy wszystkie wątki czytające stare dane wyjdą z sekcji krytycznej, będzie można (w razie potrzeby) zwolnić tę pamięć.

Kod wątku czytającego wygląda następująco:

```
1 rcu_read_lock();
2
3 local_ptr = rcu_dereference(ptr);
4 if (local_ptr != NULL) {
5   do_something(local_ptr);
6 }
7
8 rcu_read_unlock();
```

Natomiast kod wątku piszącego może wyglądać tak:

```
1 new_ptr = kmalloc(sizeof(*new_ptr), GFP_KERNEL);
2 new_ptr->field1 = 42;
3 new_ptr->field2 = 43;
4
5 rcu_assign_pointer(ptr, new_ptr);
```

Funkcja `synchronize_rcu()` pozwala nam odczekać aż wszystkie wątki obecnie przebywające w sekcji krytycznej opuszczą ją (wywołają `rcu_read_unlock()`). Zatem zwolnienie pamięci ze starymi danymi może wyglądać następująco:

```
1 synchronize_rcu();
2 kfree(some_old_ptr);
```

W jaki sposób ten mechanizm mógłby być zaimplementowany? Prześledźmy przykład:

- `rcu_assign_pointer`, `rcu_dereference()` są zwykłym przypisaniem i dereferencją z dodatkowymi instrukcjami barier pamięciowych,
- `rcu_read_lock()` i `rcu_read_unlock()` są pustymi funkcjami, bez kodu,
- `synchronize_rcu()` posiada następujący kod:

```
1 for_each_possible_cpu(cpu) {
2   run_on(cpu);
3 }
```

  To znaczy że uruchamiamy obecnie wykonujący się wątek po kolei na każdym procesorze. Skoro w sekcji chronionej przez RCU nie można się blokować, jeżeli każdy procesor uruchomił nasz wątek, to na żadnym procesorze nie został inny wątek znajdujący się w sekcji RCU. Na procesorach mogą być uruchomione inne wątki znajdujące się w sekcji krytycznej, o ile weszły do niej po naszym wywołaniu `synchronize_rcu()`.

4.7. Ułatwienia dla programisty jądra

4.7.1. Pytania pomocnicze w doborze blokad

Każdorazowo pisząc kod w jądrze używający pewnych danych można wspomóc się tym zestawem pytań ([4], s. 169):

- Czy dane są globalne? To znaczy, czy wątek inny niż mój może mieć do nich dostęp?
- Czy dane są współdzielone pomiędzy kontekstem wątku a kontekstem przerwania? Czy są współdzielone pomiędzy dwoma różnymi procedurami obsługi przerwań?
- Czy jeżeli wątek podczas używania danych zostanie wywłaszczony, czy nowo umieszczony na procesorze wątek może uzyskać dostęp do tych samych danych?
- Czy obecny wątek może się z jakiegoś powodu zablokować? Jeżeli tak, w jakim stanie pozostawi współdzielone dane?
- Co jeżeli ta funkcja zostanie wywołana ponownie, ale na innym procesorze?

Musimy być w stanie przewidzieć każdy scenariusz (przeplot wykonania) i na tej podstawie wybrać najbardziej odpowiedni środek synchronizacji by uniknąć sytuacji wyścigu pomiędzy wątkami czy procedurami obsługi przerwań. Zawsze podczas pisania kodu należy wziąć pod uwagę czy będzie on wykonywany w środowisku jednoprocesorowym czy wieloprocesorowym.

4.7.2. Analiza poprawności podczas działania systemu

System FreeBSD zapewnia nam specjalny moduł, zwany modułem świadka (ang. *witness module*), który podczas działania systemu analizuje zakładane w jądrze blokady [41]. Jest to wartościowa pomoc w procesie odpluskwiania.

Moduł świadka jest w stanie wykryć między innymi:

- czy nie próbujemy ponownie założyć blokady, która nie jest rekurencyjna,
- czy graf oczekiwania wątków na blokady należące do innych wątków nie zawiera cyklu,
- czy zakładamy blokady w dobrej kolejności, na przykład czy nie próbujemy wejść w posiadanie blokady współdzielonej typu SX, gdy jesteśmy w posiadaniu muteksu (co mogłoby skutkować przejściem w stan nieograniczonego uśpienia, a założenie jest takie, że wątki posiadające lub oczekujące na muteks nie czekają na zewnętrzne zdarzenie).

Rozdział 5.

System operacyjny Mimiker

Mimiker jest systemem operacyjnym rozwijanym od końca 2015 roku w Instytucie Informatyki Uniwersytetu Wrocławskiego. Obecnie system jest pisany na jednoprocesorową architekturę MIPS, a konkretniej na płytkę MIPS Malta. Kod źródłowy i więcej informacji można znaleźć w [32] i [33].

5.1. Obecny stan środków synchronizacji

W systemie Mimiker dostępne są głównie niskopoziomowe środki synchronizacji:

- operacje atomowe udostępniane przez procesor,
- wyłączanie przerwań (plik `sys/interrupt.c`),
- wyłączanie wywłaszczania (plik `sys/sched.c`),
- kolejki uśpionych wątków (plik `sys/sleepq.c`),
- rogatki (plik `sys/turnstile.c`).

Bariery pamięciowe nie są potrzebne, jako że w procesorach MIPS nie występuje przetwarzanie poza kolejnością. Blokady wirujące w jądrze również nie są potrzebne, jako że służą one do synchronizacji danych pomiędzy procesorami, a Mimiker jest tworzony pod architekturę jednoprocesorową.

Spośród wysokopoziomowych środków synchronizacji dostępne są:

- blokada współdzielona (plik `sys/rwlock.c`),
- muteks (plik `sys/mutex.c`),
- zmienna warunkowa (plik `sys/condvar.c`).

Muteks jako jedyny zaimplementowany jest za pomocą rogatek, zatem wspiera dziedziczenie priorytetów.

5.2. Omówienie implementacji rogatek

Jako część tej pracy w systemie Mimiker zostały zaimplementowane (przeze mnie i Wojciecha Jasińskiego) rogatki, a następnie na ich podstawie muteks [42] [43] [44] [45]. Kod muteksu widzieliśmy w podrozdziale 4.1. Teraz natomiast zapoznamy się z uproszczoną wersją rogatek z Mimikera.

5.2.1. Zasady działania i struktury danych

Mechanizm i implementacja rogatek wzorowane są na systemie FreeBSD.

Głównym zadaniem pojedynczej rogatki jest utrzymywanie listy wątków w stanie ograniczonego uśpienia, czekających na tą samą rzecz (czekających na tym samym kanale oczekiwania).

Aby szybko znaleźć rogatkę odpowiadającą danemu kanałowi oczekiwania (ang. *waiting channel* albo *wchan*), w pamięci znajduje się tablica haszująca. Rogatki wkładane są do kubełka w tablicy haszującej na podstawie hasza kanału oczekiwania (który jest zwykle wskaźnikiem, czyli liczbą). W przypadku kolizji, rogatki o takim samym haszu nawijamy na łańcuch rogatek (listę dwukierunkową). Makro `TC_HASH` służy do obliczania hasza, natomiast makro `TC_LOOKUP` znajduje łańcuch rogatek na podstawie kanału oczekiwania.

```
1 #define TC_HASH(wc) ((((wc) >> 8) ^ (wc)) & 0xFF)
2 #define TC_LOOKUP(wc) &turnstile_chains[TC_HASH(wc)]
```

Zanim przejdziemy dalej, wprowadźmy jeszcze definicje typów odpowiadające listom wątków oraz listom rogatek:

```
1 typedef TAILQ_HEAD(td_queue, thread) td_queue_t;
2 typedef LIST_HEAD(ts_list, turnstile) ts_list_t;
```

Definicja tablicy haszującej jest następująca:

```
1 typedef struct turnstile_chain {
2   ts_list_t tc_turnstiles;
3 } turnstile_chain_t;
4
5 static turnstile_chain_t turnstile_chains[256];
```

Opiszemy teraz bardziej szczegółowo sposób działania rogatek, na przykładzie muteksu.

Każdy wątek posiada swoją własną rogatkę (wewnątrz struktury wątku znajduje się wskaźnik na strukturę `turnstile_t`, której definicję podamy niedługo).

Pierwszy wątek, W1, który spróbuje wziąć wolny muteks, wchodzi w jego posiadanie, a mechanizm rogatek nie jest wcale używany. W szczególności w strukturze

W1 dalej znajduje się wskaźnik na jego rogatkę R1. Drugi wątek, W2, który spróbuje wziąć (już zajęty) muteks, musi się zablokować (w tym wypadku: przejść w stan ograniczonego uśpienia). W tym celu bierze swoją rogatkę R2, jako kanał oczekiwania używa adresu muteksu, makrem `TC_LOOKUP` szuka miejsca w tablicy haszującej i umieszcza tam swoją rogatkę (lub nawija na łańcuch rogatek o tym samym haszu, jeżeli wystąpiła kolizja). Następnie ustawia w strukturze rogatki R2 wątek W1 jako właściciela (blokady), nawija się na listę zablokowanych wątków i przechodzi w stan uśpienia, to znaczy zmienia swój stan z `TDS_RUNNING` na `TDS_BLOCKED` i wywłaszcza się, oddając sterowanie do planisty w celu wybrania kolejnego wątku do uruchomienia.

Trzeci wątek, który spróbuje wziąć (dalej zajęty) muteks, zobaczy że istnieje już rogatka R2 przypisana do kanału oczekiwania związanego z muteksem. Zatem znajdzie ją, nawinie się na listę zablokowanych wątków i przejdzie w stan uśpienia. Tak naprawdę pominęliśmy jeden ważny punkt. Jako że trzeci wątek nie będzie używał swojej rogatki R3 będąc w stanie uśpienia, odda ją do puli wolnych rogatek, która znajduje się w strukturze rogatki R2.

Każdy następny wątek blokujący się na R2 odda swoją rogatkę do puli wolnych rogatek wewnątrz R2. Gdy jakiś wątek zostanie wybudzony, weźmie rogatkę z puli wolnych. Dzięki takiemu mechanizmowi wystarczy że w systemie będziemy mieć dokładnie tyle rogatek, ile jest wątków – wątek potrzebuje rogatki wyłącznie wtedy, gdy przechodzi w stan uśpienia, a każdy wątek może być zablokowany w oczekiwaniu na co najwyżej jedną rzecz. Zauważmy, że jest to o wiele bardziej wydajny pamięciowo model w porównaniu do trzymania rogatki (w szczególności listy zablokowanych wątków) dla każdej blokady. Muteksów (i innych blokad) w systemie, to znaczy w kodzie jądra, może być bardzo dużo.

Rozważmy co się stanie, jeżeli tym razem wątek W4 o wysokim priorytecie zablokuje się na muteksie, będącym cały czas w posiadaniu wątku W1. Jako że mechanizm rogatek implementuje dziedziczenie priorytetów, wątek W1 do czasu zwolnienia muteksu otrzyma priorytet wątku W4.

Poniżej znajduje się struktura pojedynczej rogatki, w której widzimy omówione przez nas przed chwilą elementy.

```
1  typedef struct turnstile {
2    /* węzeł w łańcuchu rogatek */
3    LIST_ENTRY(turnstile) ts_chain_link;
4
5    /* węzeł na liście wolnych rogatek */
6    LIST_ENTRY(turnstile) ts_free_link;
7
8    /* wolne rogatki poprzednio należące do wątków, które obecnie
9     * są zablokowane na tej rogatce (są na liście ts_blocked) */
10   ts_list_t ts_free;
11
```

```
12   /* zablokowane wątki, czekające na kanale ts_wchan */
13   td_queue_t ts_blocked;
14
15   /* kanał oczekiwania */
16   void *ts_wchan;
17
18   /* właściciel blokady */
19   thread_t *ts_owner;
20 } turnstile_t;
```

5.2.2. Wystawiany interfejs

Dwie najważniejsze funkcje w interfejsie rogatek to:

```
1 void turnstile_wait(void *wchan, thread_t *owner);
2 void turnstile_broadcast(void *wchan);
```

Pierwsza służy do zablokowania obecnego wątku na kanale oczekiwania `wchan`, na blokadzie należącej do wątku `owner`. Druga wybudza wszystkie wątki czekające na `wchan`.

Dlaczego nie ma `turnstile_signal`, wybudzającego tylko jeden wątek? Na razie rogatki zostały zaimplementowane na potrzeby muteksu, a okazuje się, że w przypadku muteksu przy odblokowaniu blokady najszybsze jest wybudzenie wszystkich czekających wątków ([3], s. 138). Najczęstszy przypadek będzie taki, że każdy z wybudzonych wątków w swoim kwancie czasu podejmie i opuści blokadę, co jest efektywne, bo żaden z wątków nie będzie musiał przechodzić w stan uśpienia używając rogatek.

5.2.3. Synchronizacja w kodzie implementującym rogatki

Cały kod w pliku `mimiker/sys/turnstile.c` wykonuje się z wyłączonym wywłaszczaniem. Dlaczego nie musimy wyłączać przerwań? Przypomnijmy, że nie jest możliwe współdzielenie danych z kodem obsługi przerwania, ponieważ w dolnej połówce nie możemy używać rogatek (zatem muteksów, ani niczego, co może spowodować przejście w stan uśpienia).

5.3. Dalszy rozwój środków synchronizacji

Widzimy, że w systemie definitywnie brakuje wielu wysokopoziomowych środków synchronizacji obecnych w innych systemach operacyjnych. Również blokadę współdzieloną należy przepisać tak, aby używała rogatek zamiast kolejek uśpionych

wątków. Same rogatki natomiast były jednym z najbardziej brakujących elementów w Mimikerze – system cierpiał na odwrócenie priorytetów.

Rozdział 6.

Zakończenie

Synchronizacja jest ważnym elementem zarówno kodu jądra systemu operacyjnego, jak i interfejsu programisty systemowego. W przypadku przetwarzania współbieżnego, bez odpowiednio dobranych blokad nawet zwykła inkrementacja może być podatna na wystąpienie sytuacji wyścigu. Zobaczyliśmy, że poprawna (pod względem synchronizowania dostępu do danych) implementacja jądra jest bardziej skomplikowana niż implementacja programu w przestrzeni użytkownika. Ma na to wpływ wiele rzeczy, między innymi:

- różne konteksty wykonania kodu i podział jądra na dwie połówki,
- dodatkowe środki synchronizacji, niedostępne w przestrzeni użytkownika, na przykład wyłączenie wywłaszczania,
- podział wysokopoziomowych blokad na dwa rodzaje, w związku ze stanem uśpienia, jaki może wywołać próba wzięcia zajętej blokady,
- różne reguły mówiące, które blokady możemy zajmować jeżeli jesteśmy w posiadaniu innych blokad.

Jednym z elementów tej pracy była implementacja rogatek w systemie operacyjnym Mimiker. Zdecydowanie sprawiło to, że zestaw dostępnych środków synchronizacji w jądrze jest teraz bardziej uniwersalny, a używanie muteksu przestało sprawiać zagrożenie odwrócenia priorytetów.

Dwa cytaty, które zostały umieszczone na początku rozdziału 4. sugerują, że w dziedzinie środków synchronizacji istnieje jeszcze wiele możliwości na przyszłe usprawnienia.

Bibliografia

[1] William Stallings, *Operating Systems: Internals and Design Principles*, wyd. 8, Pearson Education, 2015.

[2] Andrew S. Tanenbaum, *Modern Operating Systems*, wyd. 4, Pearson Education, 2015.

[3] Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson, *The Design and Implementation of the FreeBSD Operating System*, wyd. 2, Pearson Education, 2015.

[4] Robert Love, *Linux Kernel Development*, wyd. 3, Pearson Education, 2010.

[5] Wolfgang Mauerer, *Professional Linux Kernel Architecture*, wyd. 1, Wiley Publishing, 2008.

[6] Michael Kerrisk, *Linux Programming Interface*, wyd. 1, No Starch Press, 2010.

[7] *Moore's law*, Encyclopedia Britannica, `https://www.britannica.com/technology/Moores-law` (dostęp 4.07.2018)

[8] *FreeBSD Architecture Handbook*, rozdział 8.3.1., *Interrupt Handling*, `https://www.freebsd.org/doc/en/books/arch-handbook/smp-design.html` (dostęp 13.05.2018)

[9] *Library Functions Manual*, `PTHREAD_MUTEX(3)`, `https://manpages.debian.org/stretch/glibc-doc/pthread_mutex_init.3.en.html` (dostęp 3.06.2018)

[10] *Library Functions Manual*, `SEM_OVERVIEW(7)`, `https://manpages.debian.org/stretch/manpages/sem_overview.7.en.html` (dostęp 3.06.2018)

[11] *Library Functions Manual*, `PTHREAD_RWLOCK_INIT(3)`, `https://man.openbsd.org/man3/pthread_rwlock_init.3` (dostęp 3.06.2018)

[12] *Library Functions Manual*, `PTHREAD_COND(3)`, `https://manpages.debian.org/stretch/glibc-doc/pthread_cond_init.3.en.html` (dostęp 3.06.2018)

[13] *Library Functions Manual*, PTHREAD_BARRIER_INIT(3), https://man.openbsd.org/pthread_barrier_init.3 (dostęp 3.06.2018)

[14] *Library Functions Manual*, PTHREAD_SPIN_LOCK(3), https://man.openbsd.org/pthread_spin_lock.3 (dostęp 3.06.2018)

[15] *Library Functions Manual*, POSIXOPTIONS(7), https://manpages.debian.org/stretch/manpages/posixoptions.7.en.html (dostęp 3.06.2018)

[16] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2*, https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf (dostęp 10.06.2018)

[17] *Library Functions Manual*, FUTEX(2), https://manpages.debian.org/stretch/manpages-dev/futex.2.en.html (dostęp 10.06.2018)

[18] *FreeBSD Handbook*, rozdział 10., *Linux® Binary Compatibility*, https://www.freebsd.org/doc/handbook/linuxemu.html (dostęp 10.06.2018)

[19] *MIPS32® Instruction Set Quick Reference* https://www2.cs.duke.edu/courses/fall13/compsci250/MIPS32_QRC.pdf (dostęp 26.08.2018)

[20] *Zircon Kernel Concepts*, https://fuchsia.googlesource.com/zircon/+/master/docs/concepts.md (dostęp 10.06.2018)

[21] *GCC online documentation*, *Built-in functions for atomic memory access*, https://gcc.gnu.org/onlinedocs/gcc-4.2.3/gcc/Atomic-Builtins.html (dostęp 10.06.2018)

[22] *Linux Kernel Memory Barriers*, https://www.kernel.org/doc/Documentation/memory-barriers.txt (dostęp 17.06.2018)

[23] *Lesson 1: Spin locks*, https://www.kernel.org/doc/Documentation/locking/spinlocks.txt (dostęp 27.06.2018)

[24] https://saiparancs.wordpress.com/2012/06/29/spinlock-implementation-in-arm-linux-kernel-28540/ (dostęp 30.08.2018)

[25] https://0xax.gitbooks.io/linux-insides/content/SyncPrim/linux-sync-1.html (dostęp 30.08.2018)

[26] http://bxr.su/FreeBSD/sys/sys/systm.h#245 (dostęp 30.08.2018)

[27] *Proper Locking Under a Preemptible Kernel: Keeping Kernel Code Preempt-Safe* https://www.kernel.org/doc/Documentation/preempt-locking.txt (dostęp 27.06.2018)

[28] *FreeBSD Manual Pages*, LOCKING(9), https://www.freebsd.org/cgi/man.cgi?locking(9) (dostęp 8.07.2018)

[29] *FreeBSD Manual Pages*, `SLEEPQUEUE(9)`, `https://www.freebsd.org/cgi/man.cgi?query=init_sleepqueues&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports` (dostęp 11.07.2018)

[30] *Generic Mutex Subsystem*, `https://www.kernel.org/doc/Documentation/locking/mutex-design.txt` (dostęp 15.07.2018)

[31] *FreeBSD Kernel Developer's Manual*, `SEMA(9)`, `https://www.freebsd.org/cgi/man.cgi?query=sema&sektion=9&apropos=0&manpath=FreeBSD+11.2-RELEASE+and+Ports` (dostęp 15.07.2018)

[32] `https://github.com/cahirwpz/mimiker` (dostęp 29.07.2018)

[33] `http://mimiker.ii.uni.wroc.pl/source/xref/mimiker` (dostęp 29.07.2018)

[34] `http://bxr.su/search?q=&defs=&refs=sx_init&path=&project=FreeBSD` (dostęp 30.08.2018)

[35] `http://bxr.su/search?q=&defs=&refs=sx_init_flags&path=&project=FreeBSD` (dostęp 30.08.2018)

[36] `http://bxr.su/search?q=&defs=&refs=rw_init&path=&project=FreeBSD` (dostęp 30.08.2018)

[37] `http://bxr.su/search?q=&defs=&refs=rw_init_flags&path=&project=FreeBSD` (dostęp 30.08.2018)

[38] `http://bxr.su/search?q=&defs=&refs=rm_init&path=&project=FreeBSD` (dostęp 30.08.2018)

[39] `http://bxr.su/search?q=&defs=&refs=rm_init_flags&path=&project=FreeBSD` (dostęp 30.08.2018)

[40] `http://bxr.su/search?q=&defs=&refs=RM_SLEEPABLE&path=&project=FreeBSD` (dostęp 30.08.2018)

[41] *FreeBSD Kernel Interfaces Manual*, `WITNESS(4)`, `https://www.freebsd.org/cgi/man.cgi?query=witness&sektion=4&apropos=0&manpath=FreeBSD+11.2-RELEASE+and+Ports` (dostęp 30.07.2018)

[42] *Added priority propagation functions to scheduler*, GitHub `cahirwpz/mimiker` repository pull request no. 416, `https://github.com/cahirwpz/mimiker/pull/416` (dostęp 1.09.2018)

[43] *Little cleanup before turnstile PR*, GitHub `cahirwpz/mimiker` repository pull request no. 426, `https://github.com/cahirwpz/mimiker/pull/426` (dostęp 1.09.2018)

[44] *Added turnstile tests*, GitHub `cahirwpz/mimiker` repository pull request no. 428, `https://github.com/cahirwpz/mimiker/pull/428` (dostęp 1.09.2018)

[45] *Turnstile and mutex implementation*, GitHub `cahirwpz/mimiker` repository pull request no. 418, `https://github.com/cahirwpz/mimiker/pull/418` (dostęp 1.09.2018)