Systemy Operacyjne 2

Synchronizacja w jądrze Linuksa

Arkadiusz Chrobot

Katedra Systemów Informatycznych

25 kwietnia 2024

Plan

Wstęp

Współbieżne wykonanie kodu w przestrzeni jądra jest powszechne, podobnie jak współdzielenie zasobów. Aby chronić je przed sytuacjami hazardowymi i podobnymi problemami jądro Linuksa używa szeregu mechanizmów synchronizacji, które zbiorczo są nazwane *blokadami* (ang. *locks*). W ramach tego wykładu większość z nich zostanie opisana i zostaną przedstawione ich zastosowania.

Operacje atomowe

Najprostszymi współdzielonymi zasobami są bity i zmienne prostych typów. Większość współczesnych CPU oferuje rozkazy, które przeprowadzają *atomowe* operacje na takich zasobach. *Atomowość* oznacza, że są one wykonywane w jednym kroku, sekwencyjnie i bez przerywania. Jądro Linuksa obsługuje wiele rodzajów procesorów. Niektóre z nich dostarczają takich operacji atomowych jak dodawanie, odejmowanie, odczyt, zapis, itp. Inne posiadają jedynie rozkazy blokujące magistralę danych w platformach wieloprocesorowych, kiedy jeden z procesorów korzysta z RAM. Są również CPU (np. 32-bitowe procesory SPARC), które nie posiadają żadnych atomowych rozkazów. Aby ujednolicić i zapewnić atomowe operacje dla wszystkich obsługiwanych procesorów, programiści jądra Linuksa stworzyli typ abstrakcyjny `atomic_t`. Zmienne tego typu przechowują liczby całkowite. Początkowo były to liczby tylko 24-bitowe, mimo że rozmiar zmiennej typu `atomic_t` zawsze wynosił 32 bity. Najmłodsze 8 bitów (czyli bajt) było używane do implementacji blokady, niezbędnej dla wspomnianych procesorów SPARC.

Operacje atomowe

W nowszych wersjach jądra tę blokadę zrealizowano w inny sposób i obecnie wszystkie 32 bity są używane do przechowywania liczby. Operacje dla typu `atomic_t` są zaimplementowane w postaci makr i funkcji wplatanych (ang. *inline*). Niektóre z nich są określone tylko dla konkretnych CPU, inne są dostępne we wszystkich. Do tych ostatnich zaliczają się:

`ATOMIC_INIT()` inicjuje zmienną typu `atomic_t` w miejscu jej deklaracji; argumentem tego makra jest liczba całkowita,

`int atomic_read(const atomic_t *v)` atomowo odczytuje wartość zmiennej typu `atomic_t`,

`void atomic_set(atomic_t *v, int i)` atomowo przypisuje liczbę przekazaną jako drugi argument do zmiennej typu `atomic_t`,

`void atomic_add(int i, atomic_t *v)` atomowo dodaje liczbę przekazaną jako pierwszy argument do wartości zmiennej typu `atomic_t`,

Operacje atomowe

`void atomic_sub(int i, atomic_t *v)` atomowo odejmuje liczbę przekazaną jako pierwszy argument od wartości zmiennej typu `atomic_t`,

`void atomic_inc(atomic_t *v)` atomowo zwiększa o jeden wartość zmiennej typu `atomic_t`,

`void atomic_dec(atomic_t *v)` atomowo zmniejsza o jeden wartość zmiennej typu `atomic_t`,

`int atomic_sub_and_test(int i, atomic_t *v)` atomowo odejmuje liczbę przekazaną jako pierwszy argument od wartości zmiennej typu `atomic_t` i zwraca prawdę (wartość różną od 0), jeśli wynik wynosi 0, w przeciwnym przypadku fałsz.

`int atomic_add_negative(int i, atomic_t *v)` atomowo dodaje liczbę przekazaną jako pierwszy argument do wartości zmiennej typu `atomic_t` i zwraca prawdę jeśli wynik jest ujemny, a w przeciwnym przypadku fałsz,

Operacje atomowe

`int atomic_dec_and_test(atomic_t *v)` atomowo zmniejsza o jeden wartość zmiennej typu `atomic_t` i zwraca prawdę, jeśli wynik wynosi 0, a fałsz w przeciwnym przypadku,

`int atomic_inc_and_test(atomic_t *v)` atomowo zwiększa o jeden wartość zmiennej typu `atomic_t` i zwraca prawdę, jeśli wynik wynosi 0, a fałsz w przeciwnym przypadku.

Proszę zwrócić uwagę, że zmienne typu `atomic_t` są przekazywane do funkcji przez wskaźnik, więc nie jest możliwe przeprowadzenie atomowej operacji na zmiennej typu `int`. Kiedy spopularyzowały się 64-bitowe CPU programiści jądra dodali inny typ o nazwie `atomic64_t` dla operacji atomowych na 64-bitowych zmiennych. Makra i funkcje implementujące te operacje mają nazwy zaczynające się przedrostkami `ATOMIC64_` lub `atomic64_`.

Operacje atomowe

Jądro Linuksa zapewnia także atomowe operacje na bitach. Makra i funkcje, które implementują te operacje nie wymagają specjalnego typu danych. Zazwyczaj, jako argumenty przyjmują numer bitu w słowie binarnym i adres tego słowa, przekazywany przez parametr typu `void *`. Teoretycznie jest możliwe wskazanie dowolnego bitu w pamięci tylko przy pomocy pierwszego argumentu. Numery bitów zaczynają się od 0 dla najmniej znaczącego bitu. Wspomniane funkcje i makra obejmują m.in:

`void set_bit(int nr, volatile void *addr)` atomowo ustawia na 1 nr-ty bit względem adresu przekazanego przez parametr `addr`,

`void clear_bit(int nr, volatile void *addr)` atomowo zeruje nr-ty względem adresu przekazanego przez parametr `addr`,

`int test_and_set_bit(int nr, volatile void *addr)` atomowo ustawia nr-ty bit względem adresu przekazanego w parametrze `addr` i zwraca jego poprzednią wartość,

Operacje atomowe

`int test_and_clear_bit(int nr, volatile void *addr)` atomowo zeruje nr-ty bit względem adresu przekazanego w parametrze `addr` i zwraca jego poprzednią wartość,

`int test_and_change_bit(int nr, volatile void *addr)` atomowo neguje nr-ty bit względem adresu przekazanego w parametrze `addr` i zwraca jego poprzednią wartość,

`test_bit(nr, addr)` atomowo zwraca wartość nr-tego bitu względem adresu przekazanego przez parametr `addr`.

Są również nieatomowe odpowiedniki tych funkcji, których nazwy zaczynają się od przedrostka `__` (podwójny znak podkreślenia). Jądro ma również dwie funkcje, które zwracają lokalizację pierwszego ustawionego lub wygaszonego bitu, począwszy od danego adresu. Są to, odpowiednio, `find_first_bit()` i `find_first_zero_bit()`. Jeśli tylko jedno słowo (32 lub 64-bitowe) ma być przeszukane, to mogą być użyte efektywniejsze w tym przypadku funkcje `__ffs()` i `__ffz()`.

Rygle pętlowe

Rygle pętlowe (ang. *spin locks* lub *spinlocks*) to semafory z aktywnym oczekiwaniem. Jeśli wątek wykonania próbuje zająć rygiel pętlowy, który został wcześniej uzyskany przez inny wątek, to będzie musiał w pętli sprawdzać, czy ten rygiel został już zwolniony. Rygle pętlowe chronią zazwyczaj większe niż pojedyncze zmienne zasoby, takie jak kolejki, listy i inne struktury danych. Czynią operacje na takich zasobach *niepodzielnymi*, co oznacza, że jeśli jedna z takich operacji już się odbywa na współdzielonym zasobie, to inna nie będzie mogła się rozpocząć, aż ta pierwsza się nie zakończy. Innymi słowy, te operacje są wykonywane w *sekcjach krytycznych*. W przypadku rygli pętlowych te sekcje muszą być krótsze iż czas potrzebny na przełączenie kontekstu. Mimo, że rygle pętlowe są najczęściej używanym rodzajem blokad w kodzie źródłowym jądra Linuksa, to w jego wersji skompilowanej występują tylko dla systemów wieloprocesorowych.

Rygle pętlowe

Jeśli jądro jest kompilowane dla uniprocesora i dodatkowo z włączonym wywłaszczaniem wątków jądra, to rygle pętlowe zastępowane są *przełącznikami wywłaszczania*, a jeśli wywłaszczanie tych wątków jest wyłączone, to rygle pętlowe są całkowicie pomijane w kompilacji. Rygle pętlowe są użyteczne w środowiskach wieloprocesorowych ze względu na aktywne oczekiwanie. Mogą być one stosowane w kontekście przerwania, ale tylko przy wyłączonych przerwaniach lub linii IRQ, celem uniknięcia zakleszczeń. Rygle pętlowe są także nierekurencyjne, co oznacza, że jeśli wątek, który uzyskał już rygiel próbuje go uzyskać go ponownie, to zakończy się to autozakleszczeniem. Rygle pętlowe są zmiennymi (strukturami) typu `spinlock_t`. Ich API zawiera następujące funkcje i makra:

`DEFINE_SPINLOCK(NAME)` deklaruje i inicjuje rygiel pętlowy o określonej nazwie,

`spin_lock()` uzyskuje (zajmuje) rygiel pętlowy,

`spin_lock_irq()` wyłącza lokalny system przerwań i uzyskuje rygiel pętlowy,

Rygle pętlowe

`spin_lock_irqsave()` zapisuje stan lokalnego systemu przerwań, wyłącza przerwania i uzyskuje rygiel pętlowy,

`spin_lock_bh()` wyłącza dolne połówki (przerwania programowe i tasklety) i uzyskuje rygiel pętlowy,

`spin_unlock()` zwalnia (oddaje) rygiel pętlowy,

`spin_unlock_irq()` zwalnia rygiel pętlowy i włącza lokalny system przerwań,

`spin_unlock_irqrestore()` zwalnia rygiel i przywraca zapisany stan lokalnego systemu przerwań,

`spin_unlock_bh()` włącza dolne połówki (przerwania programowe i tasklety) i zwalnia rygiel pętlowy,

`spin_trylock()` próbuje zająć rygiel pętlowy, zajmuje go jeśli jest dostępny, w przeciwnym przypadku zwraca 0,

`spin_is_locked()` zwraca prawdę, jeśli rygiel pętlowy jest zajęty, lub fałsz w przeciwnym przypadku,

`spin_lock_init()` inicjuje rygiel pętlowy.

Rygle pętlowe

Wiele problemów związanych ze współdzieleniem zasobów w jądrze sprowadza się do problemu czytelników i pisarzy. Linux posiada specjalną wersję rygli pętlowych do rozwiązywania *pierwszego problemu czytelników i pisarzy*, w którym czytelnicy mają priorytet. Te rygle nazywane są *ryglami pętlowymi R-W* (ang. *Reader–Writer Spin Locks* lub *R–W Spin Locks*). Ich API ma osobne funkcje i makra dla wątków-pisarzy i dla wątków-czytelników. Rygiel pętlowy R-W może być zajęty przez więcej niż jednego czytelnika lub nawet kilkukrotnie przez tego samego czytelnika (w tym wypadku rygiel R-W jest rekurencyjny). Jednakże tylko jeden pisarz może zająć taki rygiel w danym momencie i może to zrobić tylko wtedy, gdy tego rygla nie uzyskał wcześniej żaden czytelnik, ani inny pisarz. W skład API rygli pętlowych R-W wchodzą następujące makra i funkcje:

`DEFINE_RWLOCK` deklaruje i inicjuje rygiel pętlowy R-W (zmienną typu `rwlock_t`),

`read_lock()` uzyskuje rygiel pętlowy R-W dla czytelnika,

Rygle pętlowe

`read_lock_irq()` wyłącza lokalny system przerwań i uzyskuje rygiel pętlowy R-W dla czytelnika,

`read_lock_irqsave()` zapisuje stan lokalnego systemu przerwań, wyłącza przerwania i uzyskuje rygiel R-W dla czytelnika,

`read_lock_bh()` wyłącza dolne połówki (przerwania programowe i tasklety) i uzyskuje rygiel pętlowy R-W dla czytelnika,

`read_unlock()` zwalnia rygiel pętlowy R-W dla czytelnika,

`read_unlock_irq()` zwalnia rygiel R-W dla czytelnika i włącza lokalny system przerwań,

`read_unlock_irqrestore()` zwalnia rygiel R-W dla czytelnika i przywraca zapisany stan lokalnego systemu przerwań,

`read_unlock_bh()` zwalnia rygiel R-W dla czytelnika i włącza dolne połówki (przerwania programowe i tasklety).

`write_lock()` uzyskuje rygiel R-W dla pisarza,

Rygle pętlowe

`write_lock_irq()` wyłącza lokalny system przerwań i uzyskuje rygiel R-W dla pisarza,

`write_lock_irqsave()` zapisuje stan lokalnego systemu przerwań, wyłącza przerwania i uzyskuje rygiel R-W dla pisarza,

`write_lock_bh()` wyłącza dolne połówki (przerwania programowe i tasklety) oraz uzyskuje rygiel R-W dla pisarza,

`write_unlock()` zwalnia rygiel R-W dla pisarza,

`write_unlock_irq()` zwalnia rygiel R-W dla pisarza i włącza przerwania,

`write_unlock_irqrestore()` zwalnia rygiel R-W dla pisarza i przywraca zapamiętany stan lokalnego systemu przerwań,

`write_unlock_bh()` zwalnia rygiel R-W dla pisarza i włącza dolne połówki (przerwania programowe i tasklety),

`write_trylock()` próbuje uzyskać rygiel R-W dla pisarza, zwraca wartość różną od 0, jeśli się nie uda,

Rygle pętlowe

`rw_lock_init()` inicjuje rygiel pętlowy R-W,

`rw_is_locked()` jeśli rygiel R-W jest zajęty zwraca prawdę (wartość różną od 0).

Proszę zwrócić uwagę, że nie istnieje funkcja `read_trylock()`. Nie miała by ona uzasadnienia, bo rygiel R-W może być zajmowany wielokrotnie w tym samym czasie przez czytelników.

Semafory

W przeciwieństwie do rygla pętlowego, semafor usypia wątek wykonania, który próbuje go zająć, jeśli jest on już zajęty. Ten wątek jest także dodawany do kolejki oczekiwania związanej z tym semaforem. Aby uniknąć zakleszczenia wątek, który uzyskał rygiel pętlowy, nie może próbować uzyskać semafora. Semafory w przeciwieństwie do rygli pętlowych nie wyłączają wywłaszczania wątków jądra. Używane są one do ochrony zasobów, których użycie wymaga dłuższych sekcji krytycznych niż czas wymagany na przełączenie kontekstu. Liczba wątków wykonania, które mogą w tym samym czasie pozyskać semafor jest określana przez *licznik semafora*. Semafor jest zmienną typu `struct sempaphore`. Jest to abstrakcyjny typ danych. Operacje dla niego są zaimplementowane w postaci następujących makr i funkcji:

`sema_init(struct semaphore *, int)` inicjuje semafor; drugim argumentem jest wartość początkowa licznika semafora,

Semafory

`down_interruptible(struct semaphore *)` próbuje zająć semafor; jeśli się nie uda, to zmienia stan wątku wykonania na `TASK_INTERRUPTIBLE`,

`down(struct semaphore *)` próbuje zająć semafor; jeśli się nie uda, to zmienia stan wątku na `TASK_UNINTERRUPTIBLE`,

`down_killable(struct semaphore *)` próbuje zająć semafor, jeśli się to nie uda, to zmienia stan wątku na `TASK_KILLABLE`; ta funkcja jest dostępna od wersji 2.6.26 jądra,

`down_timeout(struct semaphore *, long)` dostępna od wersji 2.6.16 jądra; pozwala wątkowi wykonawczemu określić maksymalny czas oczekiwania na uzyskanie semafora,

`down_trylock()` próbuje uzyskać semafor; jeśli jest to w danym czasie niemożliwe, to zwraca prawdę,

`up(struct semaphore *)` zwalnia semafor i budzi wątki wykonania oczekujące na to wydarzenie.

Semafory

Semafor może zostać zadeklarowany i zainicjowany z użyciem makra `DEFINE_SEMAPHORE`. Tak, jak w przypadku rygli pętlowych, istnieją specjalne semafory dla pierwszego problemu czytelników i pisarzy. Te semafory są zmiennymi typu `struct rw_semaphore`. Te semafory mogą być tworzone i inicjowane przy użyciu makra `DECLARE_RWSEM`. Do samej ich inicjacji służy funkcja `init_rwsem()`. Funkcje obsługujące semafory R-W dla czytelników mają podobne nazwy do funkcji obsługujących zwykłe semafory, ale kończą się one wyrazem `read`. Podobnie, funkcje obsługujące semafory R-W mają nazwy kończące się wyrazem `write`. Są również funkcje `down_read_trylock()` i `down_write_trylock()`, ale w przeciwieństwie do `down_trylock()` zwracają 0 jeśli semafor jest niedostępny w momencie, w którym próbują go zająć. Jest także funkcja `downgrade_writer()`, która konwertuje blokadę dla pisarza na blokadę dla czytelnika.

Muteksy

Programiści Linuksa zauważyli, że w kodzie jądra często używane są semafory binarne (przyjmujące tylko dwie wartości). W związku z tym zdecydowali o dodaniu nowego typu blokady, która miałaby je zastąpić. Tą blokadą jest *mutex*[1]. Muteksy są zmiennymi o następującym typie[2]:

```
struct mutex {
        atomic_t count;
        spinlock_t wait_lock;
        struct list_head wait_list;
};
```

Listing 1: Uproszczona definicja typu `struct mutex`.

[1]Nazwa pochodzi od wyrażenia *wzajemne wykluczanie* (ang. *mutual exclusion*).

[2]Rzeczywista definicja tego typu posiada kilka dodatkowych pól używanych do debugowania.

Muteksy

W przeciwieństwie do budowy typu `struct semaphore`, budowa typu `struct mutex` jest niezależna od CPU, podobnie jak większość operacji dla muteksa (niektóre są zoptymalizowane dla określonych CPU obsługiwanych przez jądro). Pole `count` przechowuje stan muteksa. Jeśli jego wartość wynosi 1, to mutex jest niezajęty, jeśli 0 to mutex jest zajęty, a jeśli jest mniejsza od 0, to mutex jest zajęty i co najmniej jeden wątek wykonania oczekuje na jego zwolnienie. Ujemna wartość oznacza także, że oczekujący wątek wykonania będzie potrzebował wybudzenia. Podobnie jak semafory, muteksy mogą być używane wyłącznie w kontekście procesu. Nie są one również rekurencyjne. Ich API zawiera następujące makra i funkcje:

`DEFINE_MUTEX(NAME)` deklaruje i inicjuje mutex o danej nazwie,

`mutex_init(struct mutex *lock)` inicjuje mutex,

`mutex_lock_interruptible(struct mutex *lock)` próbuje zająć mutex; jeśli się to nie uda to zmienia stan wątku wykonania na `TASK_INTERRUPTIBLE`,

Muteksy

`mutex_lock(struct mutex *lock)` próbuje zająć mutex; jeśli jest on w danym czasie niedostępny, to zmienia stan wątku wykonania na `TASK_UNINTERRUPTIBLE`,

`mutex_lock_killable(struct mutex *lock)` próbuje zająć mutex; jeśli się to nie uda, to zmienia stan wątku wykonania na `TASK_KILLABLE`,

`mutex_trylock(struct mutex *lock)` próbuje zająć mutex i zwraca 1, jeśli się to uda lub 0 jeśli nie,

`mutex_unlock(struct mutex *lock)` zwalnia mutex,

`mutex_is_locked(struct mutex *lock)` zwraca 0, jeśli mutex jest odblokowany lub wartość różną od 0, jeśli nie jest.

W jądrze Linuksa dostępne są również muteksy czasu rzeczywistego, które zostały krótko opisane w trzecim wykładzie. Każdy wątek wykonania, który zajmie taki mutex zyskuje priorytet czasu rzeczywistego, aby zapobiec wystąpieniu problemu *odwrócenia priorytetów*. Priorytet wątku wraca do poprzedniej wartości po zwolnieniu tego muteksa.

Zmienne sygnałowe

Zmienne sygnałowe to uproszczone semafory. Stosowane są w sytuacjach, gdzie występuje kilka wątków i jeden z nich musi poinformować pozostałe o zakończeniu wykonywanego przez niego zadania. Zmienne sygnałowe są typu `struct complection`. To abstrakcyjny typ danych z operacjami zrealizowanymi w postaci następujących makr i funkcji:

`DECLARE_COMPLETION()` deklaruje i inicjuje zmienną sygnałową,

`init_completion(struct completion *)` inicjuje zmienną sygnałową,

`wait_for_completion(struct completion *)` oczekuje na sygnał, że inny wątek skończył zadanie,

`complete(struct completion *)` sygnalizuje zakończenie zadania przez wątek.

Więcej informacji o API zmiennych sygnałowych znajduje się w piątej instrukcji laboratoryjnej.

Blokada BKL

Blokada BKL (ang. *Big Kernel Lock* - BKL) została dodana do jadra w serii 2.2, by umożliwić w środowisku SMP synchronizację pracy, gdy więcej niż jeden CPU próbuje wykonać kod jądra. Miała być ona tymczasowym rozwiązaniem, niestety pozostała w jądrze dłużej niż początkowo planowano. BKL jest globalnym ryglem pętlowym z dodatkowymi własnościami. Wątek wykonania rozpoczyna aktywne oczekiwanie próbując uzyskać BKL, jeśli jest ona już zajęta. Jednakże wątek, który posiada BKL może przejść w stan uśpienia i wtedy ta blokada jest zwalniana. BKL jest ponownie zakładana, kiedy ten wątek się budzi. Jest ona również rekurencyjna, wyłącza wywłaszczanie jądra i może być używana tylko w kontekście procesu. Jej API składa się z trzech funkcji:

`lock_kernel()` zakłada BKL,

`unlock_kernel()` zwalnia BKL,

`kernel_lock()` sprawdza, czy BKL jest założona.

W wersji 2.6.39 jądra programistom udało się ostatecznie pozbyć BKL i w kolejnych wersjach jądra nie jest już obecna.

Blokady sekwencyjne

Blokady sekwencyjne (ang. *sequence locks* lub *sequential locks* lub *seqlocks*) zostały dodane do jądra w serii 2.6. To prosty mechanizm synchronizacji przewidziany dla *drugiego problemu czytelników i pisarzy*, gdzie priorytet mają pisarze. Są one w zasadzie licznikami sekwencyjnymi. Ich wartość początkowa wynosi 0. W sytuacji, gdzie jest tylko jeden pisarz i wielu czytelników, pisarz zwiększa o 1 wartość blokady przed i po modyfikacji wspólnego zasobu. Czytelnik odczytuje wartość blokady zarówno przed, jaki i po skorzystaniu z tego zasobu i porównuje wyniki odczytów. Jeśli ich wartości są takie same, ale nieparzyste, to oznacza, że czytelnik odczytywał zasób, kiedy był on modyfikowany i mógł on mieć nieprawidłową wartość. Jeśli odczytane wartości blokady sekwencyjnej są różne, to operacja odczytu została przepleciona z operacją zapisu. W obu wypadkach odczyt zasobu musi być powtórzony. Jeśli z zasobu korzysta więcej niż jeden pisarz, to blokada sekwencyjna działa dla wątków tego typu jak rygiel pętlowy.

Blokady sekwencyjne

Blokady sekwencyjne są zmiennymi typu `seqlock_t`. Pisarze używają dwóch funkcji do inkrementacji licznika takiej blokady. Pierwszą jest `wirte_seqlock()`, która jest wywoływana przez zmianą wspólnego zasobu, a drugą `write_sequnlock()`, która jest używana po zakończeniu tej operacji. Czytelnik, przed skorzystaniem ze wspólnego zasobu, odczytuje wartość blokady sekwencyjnej przy pomocy funkcji `read_seqbegin()` i zapamiętuje ją w swojej zmiennej lokalnej. Po użyciu wspólnego zasobu wątek czytelnika wywołuje `read_seqretry()`, która odczytuje bieżącą wartość blokady i porównuje ją z poprzednią zapamiętaną w zmiennej lokalnej. Czytelnik powtarza te operacje, łącznie z korzystaniem z zasobu, tak długo, jak długo te wartości są różne lub takie same, ale nieparzyste. Obie funkcje czytelnika są zazwyczaj wywoływane w pętli `do…while`. Inne szczegóły użycia blokady sekwencyjnej zostały opisane w piątej instrukcji laboratoryjnej.

Wyłączanie wywłaszczania

Jeśli wspólne zasoby wymagające ochrony są lokalne dla jednego z CPU w środowisku wieloprocesorowym, czyli niedostępne dla pozostałych procesorów, to wyłączanie i włączanie wywłaszczania wątków jądra dla tego CPU jest wystarczającym środkiem synchronizacji. Wywłaszczanie jądra jest wyłączone, jeśli wartość *licznika wywłaszczeń* (jedno z pól struktury `struct thread_info`, w przypadku większości platform sprzętowych wspieranych przez Linuksa) jest większa od 0. Funkcja `preempt_count()` zwraca bieżącą wartość tego pola. Z kolei `preempt_disable()` zwiększa jego wartość o 1, a `preempt_enable()` zmniejsza o 1. Wynika z tego, że wywołania dwóch ostatnich funkcji mogą być zagnieżdżone, ale druga musi być wywołana tyle razy co pierwsza, aby przywrócić wywłaszczanie wątków jądra. Funkcja `preempt_enable_no_resched()` włącza wywłaszczanie jądra, ale nie aktywuje planisty.

Wyłączanie wywłaszczania

W środowisku wieloprocesorowym funkcja `get_cpu()` może być użyta do wyłączenia wywłaszczania jądra na określonym CPU. Zwraca ona numer identyfikacyjny tego procesora. Funkcja `put_cpu()` włącza wywłaszczanie jadra na określonym procesorze.

Wyłączanie dolnych połówek

Dolne połówki, a dokładniej przerwania programowe i tasklety na danym procesorze mogą być wyłączone funkcją `local_bh_disable()`. Ich działanie jest przywracane przez funkcję `local_bh_enable()`.

Bariery

Większość współczesnych potokowych procesorów stosuje technikę *wykonania poza kolejnością* (ang. *out–of–order execution*) by wykonywać programy bardziej efektywnie. Oznacza to, że niektóre operacje odczytu i zapisu mogą być wykonane w innej kolejności niż są umieszczone w kodzie źródłowym. Jeśli są od siebie niezależne, to nie ma to znaczenia, ale jeśli nie są, to mogą generować błędne wyniki. Procesor może rozpoznać niektóre proste zależności między odczytami i zapisami, ale nie te bardziej skomplikowane. Podobne problemy może spowodować kompilacja, bo współczesne kompilatory dokonują optymalizacji kodu także poprzez zmianę kolejności instrukcji. Aby im zapobiec programiści jądra wprowadzili blokady nazywane *barierami* (ang. *barriers*):

`rmb()` zapobiega zmianie kolejności dowolnych odczytów wokół niej,

`read_barrier_depends()` jak wyżej, ale tylko dla odczytów wzajemnie zależnych,

Bariery

`wmb()` zapobiega zmianie kolejności dowolnych zapisów wokół niej,

`mb()` zapobiega zmianie kolejności dowolnych zapisów i odczytów wokół niej,

`smp_rmb()` w systemach SMP zachowuje się jak `rmb()`, a w uniprocesorach jak `barrier()`,

`smp_read_barrier_depends()` w systemach SMP zachowuje się jak `read_barrier_depends()`, a w systemach jednoprocesorowych jak `barrier()`,

`smp_wmb()` w systemach SMP zachowuje się jak `wmb()`, a w uniprocessorach jak `barrier()`,

`smp_mb()` w systemach SMP zachowuje się jak `mb()` a w uniprocesorach jak `barrier()`,

`barrier()` zapobiega optymalizacji przez kompilator odczytów i zapisów wokół niej.

Mechanizm RCU

Mechanizm RCU[3] jest efektywnym, wysoce skalowalnym mechanizmem synchronizacji rozwiązującym pierwszy problem czytelników i pisarzy. Wymaga on małego nakładu pamięci i spełnienia następujących warunków, aby działać poprawnie:

- kod czytelnika związany z odczytem dzielonego zasobu nie może przejść w stan oczekiwania,
- zapisy zasobu powinny być rzadkie, a odczyty częste,
- wątki czytelników mogą korzystać ze wspólnego zasobu tylko za pośrednictwem wskaźników.

Jeśli czytelnik chce odczytać wspólny zasób, to musi uzyskać do niego wcześniej wskaźnik i wykonać tę operację za jego pośrednictwem. Jeśli pisarz chce zmodyfikować zasób, to robi jego kopię, zmienia ją i publikuje do niej wskaźnik. Jeśli dowolny czytelnik spróbuje po tym uzyskać dostęp do wspólnego zasobu, to otrzyma wskaźnik do tej jego kopii.

[3]Nazwa pochodzi od nazw trzech operacji: odczytu, kopiowania i aktualizacji (ang. *Read–Copy–Update*).

Mechanizm RCU

Oryginał zasobu jest niszczony, kiedy przestanie z niego korzystać ostatni czytelnik, który miał do niego dostęp. Pisarz używa makra `rcu_assign_ptr`, aby opublikować wskaźnik do zmodyfikowanej kopii wspólnego zasobu. Może on również zarejestrować funkcję wywołania zwrotnego (ang. *callback function*), która usunie oryginał zasobu, kiedy ostatni czytelnik przestanie z niego korzystać. W tym celu może użyć funkcji `call_rcu()`. Pisarz może także usunąć oryginał po zakończeniu funkcji `sychronize_rcu()`. Czytelnik używa funkcji `rcu_read_lock()` celem uzyskania wskaźnika do wspólnego zasobu oraz makra `rcu_dereference`, by dostać się do tego zasobu. By zwolnić wskaźnik na zasób (ale nie wskazywaną przez niego pamięć) pisarz wywołuje funkcję `rcu_read_unlock()`. Opisane makro może być użyte tylko między wywołaniami tych dwóch funkcji. Mechanizm RCU nie zapewnia ochrony przed współbieżnymi zapisami, więc stosowany jest głównie wtedy, gdy jest tylko jeden pisarz. W innych przypadkach należy zastosować inne środki synchronizacji dla pisarzy.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!