Laboratorium 5: „Wątki jądra i mechanizmy synchronizacji” (jedne zajęcia)

dr inż. Arkadiusz Chrobot

12 kwietnia 2024

Spis treści

Wprowadzenie

Ta instrukcja jest poświęcona wątkom jądra i wybranym mechanizmom synchronizacji. Rozdział 1 opisuje mechanizm działania wątków jądra, a rozdział 2 wybrane mechanizmy synchronizacji, czyli operacje niepodzielne, muteksy, zmienne sygnałowe, blokady sekwencyjne i mechanizm RCU. Ostatni rozdział instrukcji zawiera listę zadań do samodzielnego rozwiązania w ramach zajęć laboratoryjnych.

1. Wątki jądra

Wątki jądra tworzone są w przestrzeni jądra, co oznacza, że współdzielą wszystkie zasoby z pozostałymi podsystemami jądra systemu operacyjnego. Takie wątki najczęściej należą do jednej z dwóch kategorii:

1. wątki, które aktywowane są przez określone podsystemy jądra, wykonują pewne czynności i przechodzą w stan oczekiwania,

2. wątki, których działanie aktywowane jest przez mechanizm czasowy co określony odcinek czasu i polega ono na wykonaniu określonych prac i przejściu w stan oczekiwania.

Mechanizm działania obu typów wątków jest zatem podobny. Oba rodzaje wątków przechodzą w stan `TASK_RUNNING` za sprawą zdarzenia pochodzącego z zewnątrz, wykonują pracę np. związaną z monitorowaniem stanu zasobów, a następnie przechodzą w stan oczekiwania, najczęściej `TASK_INTERRUPTIBLE`. To działanie jest cykliczne. Wątki jądra są uruchamiane wraz ze startem systemu operacyjnego lub po załadowaniu modułu, w którym są oprogramowane, a kończą działanie przy usuwaniu modułu lub zamykaniu systemu. Przejście takich wątków w stan oczekiwania jest możliwe dzięki temu, że są one wykonywane w *kontekście procesu*. Tak jak zadania użytkownika, podlegają one szeregowaniu, ale niekoniecznie podlegają wywłaszczaniu. Zależy to od konfiguracji jądra z jaką zostało ono skompilowane. Jeśli opcja wywłaszczania wątków w konfiguracji kompilacji jądra jest wyłączona, to wszystkie wątki jądra będą tak długo aktywne, jak długo same nie zrzekną się procesora. Dlatego ważnym jest, aby wątek jądra po wykonaniu pracy samodzielnie wprowadził się w stan oczekiwania i wywołał planistę procesora, celem przeszeregowania zadań. Informację o uruchomionych wątkach jądra można uzyskać przy pomocy polecenia `ps aux`. Nazwy wątków jądra na liście wyświetlonej przez to polecenie są ujęte w nawiasy kwadratowe.

1.1. Opis API

Wątki jądra są implementowane w postaci funkcji, które mają następujący prototyp:

```
int thread_function(void *data)
```

W definicji tej funkcji należy umieścić kod, który będzie wykonywany przez wątek. Tego rodzaju funkcje

będziemy nazywać *funkcjami wątku*. Do obsługi wątków w jądrze Linuksa zdefiniowano następujące funkcje i makra:

`kthread_create(threadfn, data, namefmt, arg…)` - makro to tworzy wątek. Przyjmuje ono trzy argumenty. Pierwszym z nich jest wskaźnik na funkcję wątku, drugim wskaźnik na dane dla tej funkcji, który jest jej przekazywany przez parametr `data`, trzecim jest nazwa wątku, która może zawierać ciągi formatujące, tak jak w przypadku funkcji `printf()`. Jeśli tak jest, to po trzecim argumencie powinny występować kolejne, których typy są zgodne z zawartymi w nazwie ciągami formatującymi. Makro zwraca wskaźnik na deskryptor utworzonego wątku, czyli strukturę typu `struct task_struct`. Utworzony wątek jest domyślnie nieaktywny (jest w stanie `TASK_INTERRUPTIBLE`. Aby go uaktywnić należy wywołać funkcję `wake_up_process()` ze wskaźnikiem na deskryptor wątku jako argumentem jej wywołania. Funkcja zwróci wartość `1`, jeśli wątek zostanie uaktywniony (obudzony) lub `0`, jeśli już był aktywny.

`kthread_run(threadfn, data, namefmt, …)` - makro to tworzy wątek i uaktywnia go. Przyjmuje takie same argumenty jak `kthread_create` i zwraca wartości tego samego typu. **Nie wolno używać tych dwóch makr z tą samą funkcją wątku.**

`void kthread_bind(struct task_struct *k, unsigned int cpu)` - funkcja ta pozwala określić, na którym procesorze, w komputerze z wieloma procesorami, wątek będzie wykonywany. Przyjmuje ona dwa argumenty wywołania: wskaźnik na deskryptor wątku i identyfikator procesora, który jest liczbą naturalną. Pierwszy procesor ma przyporządkowany identyfikator o wartości `0`. Funkcja nic nie zwraca.

`int kthread_stop(struct task_struct *k)` - funkcja wywoływana w celu zakończenia działania wątku jądra. Jako argument wywołania przyjmuje ona wskaźnik na deskryptor wątku, a zwraca wartość zwróconą przez funkcję wątku lub wartość wyrażenia `-EINTR` jeśli wątek nigdy nie był aktywny.

`bool kthread_should_stop(void)` - funkcja ta wywoływana jest wewnątrz funkcji wątku i zwraca wartość `true`, jeśli dla tego wątku została wywołana funkcja `kthread_stop()`, a `false` w przeciwnym przypadku. Jeśli ta funkcja zwróci prawdę, to funkcja wątku powinna się zakończyć.

Opisane podprogramy nie są jedynymi przeznaczonymi do obsługi wątków jądra, ale stanowią niezbędne minimum. Są one zadeklarowane w pliku nagłówkowym `linux/kthread.h`.

Aby wątek mógł zrzec się procesora i poczekać, aż zostanie uaktywniony przez inny kod jądra, konieczne może być zdefiniowanie kolejki oczekiwania i dodanie go do niej. Taka kolejka jest strukturą typu `wait_queue_head_t`. Funkcje i makra związane z jej obsługą są zadeklarowane w pliku nagłówkowym `linux/wait.h`. Oto lista opisująca niektóre z nich:

`init_waitqueue_head(q)` - makro, które odpowiedzialne jest za inicjację kolejki oczekiwania. Jako argument przyjmuje adres (wskaźnik do) tej kolejki.

`DEFINE_WAIT(name)` - makro, które definiuje element kolejki oczekiwania, który może być do niej dodany.

`void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)` - funkcja ta dodaje element wskazywany przez jej parametr `wait` do kolejki oczekiwania wskazywanej przez jej parametr `q`.

`void prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait, int state)` - funkcja, jeśli jest to konieczne, dodaje element wskazywany przez jej parametr `wait` do kolejki oczekiwania wskazywanej przez parametr `q`, a następnie wprowadza wątek jądra lub zadanie użytkownika, które ją wywołało w stan określony przez wartość przekazaną przez parametr `state`. Argumentem podstawianym za ten parametr jest zazwyczaj stała określająca stan oczekiwania wątku lub zadania.

`void finish_wait(wait_queue_head_t *q, wait_queue_t *wait)` - funkcja ta wprowadza wątek jądra lub zadanie użytkownika, które je wywołało w stan `TASK_RUNNING` i usuwa element wskazywany przez jej parametr `wait` z kolejki wskazywanej przez parametr `q`.

`wake_up(x)` - makro, które uaktywnia (wybudza) pojedynczy wątek czekający w kolejce oczekiwania. Jako argument przyjmuje adres (wskaźnik do) kolejki oczekiwania.

`wake_up_all(x)` - makro, które wybudza wszystkie wątki czekające w kolejce oczekiwania, której wskaźnik jest jej przekazywany jako argument.

1.2. Przykład

Listing 1 zawiera kod źródłowy modułu jądra, który tworzy dwa wątki. Jeden z wątków jest aktywowany co sekundę i jego jedynym zadaniem jest aktywacja drugiego z wątków, który umieszcza komunikat w buforze jądra.

Listing 1: Przykładowy moduł z dwoma wątkami

```
#include<linux/module.h>
#include<linux/kthread.h>
#include<linux/wait.h>

enum thread_index {WAKING_THREAD, SIMPLE_THREAD};

static struct threads_structure
{
        struct task_struct *thread[2];
} threads;

static wait_queue_head_t wait_queue;
static bool condition;

static int simple_thread(void *data)
{
        DEFINE_WAIT(wait);
        for(;;) {
                add_wait_queue(&wait_queue,&wait);
                while(!condition) {
                        prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                        if(kthread_should_stop())
                                return 0;
                        printk(KERN_INFO "[simple_thread]: awake\n");
                        schedule();
                }
                condition=false;
                finish_wait(&wait_queue,&wait);
        }
}

static int waking_thread(void *data)
{
        for(;;) {
                if(kthread_should_stop())
                        return 0;
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(1*HZ))
                        printk(KERN_INFO "Signal received!\n");
                condition=true;
                wake_up(&wait_queue);
        }

}

static int __init threads_init(void)
{
        init_waitqueue_head(&wait_queue);
        threads.thread[SIMPLE_THREAD] = kthread_run(simple_thread,NULL,"simple_thread");
        threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
        return 0;
}
```

```
53
54 static void __exit threads_exit(void)
55 {
56         kthread_stop(threads.thread[WAKING_THREAD]);
57         kthread_stop(threads.thread[SIMPLE_THREAD]);
58 }
59
60 module_init(threads_init);
61 module_exit(threads_exit);
62
63 MODULE_LICENSE("GPL");
64 MODULE_DESCRIPTION("An example of using the Linux kernel threads.");
65 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
66 MODULE_VERSION("1.0");
```

W wierszach nr 2 i 3 kodu źródłowego tego modułu są włączone pliki nagłówkowe związane z obsługą wątków i kolejek oczekiwania. W wierszu nr 5 zdefiniowany został typ wyliczeniowy, którego elementy zostaną użyte jako stałe określające indeksy elementów tablicy będącej polem struktury `threads`. Ta struktura zadeklarowana jest w wierszu nr 10 opisywanego kodu źródłowego, a wcześniej zdefiniowany jest jej typ (`struct thread_structure`). Struktura `threads` zawiera pojedyncze pole, które jest tablicą wskaźników na deskryptory procesów. Zostaną w niej zapamiętane adresy deskryptorów utworzonych przez moduł wątków jądra. Wątek, adres deskryptora wątku okresowo budzonego będzie zapamiętany w elemencie tej tablicy, którego indeks określa wartość stałej `WAKING_THREAD`, a adres deskryptora wątku uaktywnianego przez wątek okresowo budzony będzie zapamiętywany w elemencie o indeksie, którego wartość jest określona stałą `SIMPLE_THREAD`. W wierszu nr 12 zadeklarowana jest kolejka oczekiwania o nazwie `wait_queue`. W wierszu nr 13 zadeklarowana jest zmienna typu `bool`, o nazwie `condition`, której wartość będzie określała, czy wątek może skończyć oczekiwanie w kolejce.

Wiersze 15-30 zawierają definicję funkcji `simple_thread()`, która jest funkcją wykonywaną w ramach wątku aktywowanego przez drugi z wątków uruchamianych przez moduł. W wierszu 17 zadeklarowany został element kolejki oczekiwania o nazwie `wait` będzie on reprezentował wątek wykonujący tę funkcję w kolejce oczekiwania. Wiersze 18-19 zawierają „nieskończoną” pętlę `for`. Jak opisano to w poprzednim rozdziale, wątki wykonują najczęściej swą pracę cyklicznie, dlatego ta pętla została użyta w funkcji wątku. Pierwszą czynnością wykonywaną w tej pętli jest dodanie wątku, reprezentowanego przez zmienną `wait` do kolejki oczekiwania (wiersz nr 19). W wierszach 20-26 umieszczona jest pętla `while`, która kończy się, kiedy wartość zmiennej `condition` będzie wynosiła `true`. Wewnątrz tej pętli wątek dodaje się do kolejki oczekiwania i zmienia swój stan na stan oczekiwania (`TASK_INTERRUPTIBLE`), o ile jeszcze w nim nie był - proszę zwrócić uwagę, że nie jest to jednoznaczne z przerwaniem działania tego wątku. Następnie wywołuje on funkcję `kthread_should_stop()` celem sprawdzenia, czy nie powinien się zakończyć. Jeśli ta funkcja zwrócić wartość `true`, to funkcja wątku zwraca wartość `0` i kończy działanie. W przeciwnym przypadku wątek dodaje komunikat do bufora jądra, a następnie wywołuje funkcję `schedule()`, czyli planistę procesora, który odbiera wątkowi procesor i przekazuje ją innemu zadaniu (procesowi lub wątkowi użytkownika, bądź innemu wątkowi jądra). Dopiero po wykonaniu tych czynności wątek znajdzie się w stanie oczekiwania. Jeśli nastąpił powrót z funkcji `schedule()`, to znaczy, że wątek został obudzony i planista oddał mu procesor. Wątek wykonuje następną iterację pętli `while` celem upewnienia się, że zmienna `condition` ma wartość `true`, co znaczy, że faktycznie może zakończyć oczekiwanie. Wątki jądra mogą być obudzone tylko z dwóch powodów: albo muszą zakończyć swe działanie, albo wystąpiło zdarzenie, na które oczekiwały. W tym przykładowym module wartość zmiennej `condition` służy opisywanemu wątkowi do rozróżnienia tych dwóch przyczyn. Po zakończeniu pętli `while` wątek ustawia wartość zmiennej `condition` na `false` (wiersz nr 27) oraz zmienia swój stan na `TASK_RUNNING` i usuwa się z kolejki oczekiwania (wiersz nr 28), a następnie wykonuje kolejną iterację pętli `for`.

W wierszach 32-44 umieszczona jest definicja funkcji `waking_thread()`, która będzie wykonywana w ramach wątku aktywowanego cyklicznie po upływie określonego czasu. Ta funkcja jest zbudowana inaczej niż poprzednia. Wykonywana w niej jest tylko jedna pętla. Jest to „nieskończona” pętla `for`. Wewnątrz tej pętli wątek najpierw sprawdza, czy nie powinien się zakończyć (wiersze 35-36) jeśli tak, to funkcja wątku zwraca `0` i kończy działanie. W przeciwnym przypadku wątek zmienia swój stan na `TASK_INTERRUPTIBLE` przy pomocy wywołania funkcji `set_current_state()`, która ustawia stan zdania będącego aktualnie w posiadaniu procesora, na taki, jaki określa przekazana jej jako argument wywo-

łania wartość. Po tym wątek wywołuje funkcję `schedule_timeout()`. Zadaniem tej funkcji jest opóźnienie wykonania zadania na określony czas, którego wartość jest określona jej argumentem wywołania. Aby umożliwić korzystanie innym zadaniom z procesora przypisanego wątkowi, funkcja ta usypia wątek i umieszcza go w kolejce oczekiwania na upływ określonego w jej wywołaniu czasu, tym samym zwalniając procesor. Wszystkie te czynności są zaszyte w jej definicji. Czas oczekiwania w przykładowym module jest określony za pomocą stałej `HZ`. Jej wartość jest zależna od platformy sprzętowej, na której jest uruchomiony Linux, ale zawsze określa ona częstotliwość zegara systemowego, czyli ile sygnałów przerwań wygeneruje on przez sekundę. W opisywanym module wątek będzie czekał na aktywację przez sekundę [1]. Jeśli nastąpił powrót z funkcji `schedule_timeout()` i zwróciła ona wartość `0`, to znaczy, że czas, na jaki działanie wątku zostało opóźnione już upłynął[2]. Jeżeli jednak funkcja zwróciła wartość różną od zera, to znaczy, że dla wątku została wywołana funkcja `kthread_stop()`. W tym przypadku wątek umieszcza odpowiedni komunikat w buforze jądra. Konieczność zatrzymania działania wątku będzie obsłużona na początku kolejnej iteracji pętli `for` przez instrukcję warunkową zawartą w wierszach 35 i 36. Po obudzeniu wątek jest w stanie `TASK_RUNNING`. W wierszu nr 40 zmienia on wartość zmiennej `condition` na `true` i w wierszu nr 41 wywołuje funkcję `wake_up()` dla kolejki, w której (prawdopodobnie) oczekuje na obudzenie pierwszy z przedstawionych wątków. Po wykonaniu tych czynności funkcja `waking_thread()` przechodzi do realizacji kolejnego powtórzenia pętli `for`.

W konstruktorze modułu (wiersze 46-52) inicjowana jest kolejka oczekiwania (wiersz nr 48) oraz tworzone są dwa wątki (wiersze 49-50). Pierwszy z nich (aktywowany zdarzeniem) będzie widoczny na liście zadań pod nazwą *simple_thread*, a drugi (aktywowany upływem czasu) pod nazwą *waking_thread*. Adresy deskryptorów tych wątków zapisywane są w tablicy `thread` struktury `threads`. Proszę zwrócić uwagę, że po utworzeniu oba wątki są aktywowane.

W destruktorze modułu (wiersze nr 54-58) dla obu wątków wywoływana jest funkcja `kthread_stop()`, która sygnalizuje im konieczność zakończenia ich pracy. Wartości zwracane przez wywołania tej funkcji są ignorowane.

Komunikaty umieszczane w buforze jądra przez wątki najwygodniej jest obserwować wywołując polecenie `dmesg` z opcjami `-w -d`. Pierwsza opcja nakazuje temu poleceniu oczekiwać na pojawienie się kolejnego komunikatu, celem wypisania go na ekranie, a druga opcja powoduje, że `dmesg` umieszcza informację o tym ile czasu upłynęło między umieszczeniem przez wątki w buforze kolejnych komunikatów. Działanie tak wywołanego polecenia `dmesg` można przerwać przez naciśnięcie kombinacji klawiszy `Ctrl+C`.

2. Mechanizmy synchronizacji

Tak, jak w przypadku zdań użytkownika, tak i w przypadku wątków jądra mogą się pojawić sytuacje hazardowe, gdy korzystają one ze wspólnych zasobów. Aby nim zapobiec należy stosować mechanizmy synchronizacji, które zaimplementowali programiści jądra. W tej części instrukcji są opisanie niektóre z nich. Można je podzielić na dwie kategorie: środki ogólnego przeznaczenia (operacje niepodzielne, muteksy i zmienne sygnałowe) oraz środki zorientowane na rozwiązanie problemu czytelników i pisarzy (blokady sekwencyjne, mechanizm RCU).

2.1. Operacje niepodzielne

Jeśli zasobem współdzielonym przez wątki i inne fragmenty kodu jądra są zmienne typu całkowitego lub nawet pojedyncze bity w słowie, to do ich ochrony można zastosować *operacje niepodzielne* zaimplementowane w postaci makr i funkcji. W pliku nagłówkowym `linux/types.h` zostały zdefiniowane dwa abstrakcyjne typy danych: `atomic_t` i `atomic64_t`. Zmienne tych typów przechowują, odpowiednio, 32-bitowe i 64-bitowe liczby całkowite. Operacje, które zdefiniowano dla tych typów są wykonywane w sposób niepodzielny (ang. *atomic*), co w tym przypadku oznacza dwie rzeczy:

1. zanim nie zakończy się bieżąco wykonywana operacja na zmiennej dowolnego ze wspomnianych typów, to nie może się rozpocząć kolejna dotycząca tej samej zmiennej,

[1] Mnożenie stałej `HZ` przez jeden jest w tym wypadku niepotrzebne, ale pokazuje w jaki sposób można zmienić ten okres czasu, zamiast jedynki stosując inną wartość.

[2] Pomiar tego czasu może nie być dokładny, więc wątek może być obudzony o kilka taktów zegara wcześniej lub później.

2. wykonanie tych operacji nie zostanie nigdy przerwane, zawsze wykonają się do końca.

Dla typu `atomic_t` zdefiniowano następujące operacje niepodzielne w postaci makr i funkcji:

`ATOMIC_INIT(i)` - makro, które służy inicjacji zmiennej typu `atomic_t`. Przyjmuje jeden argument, którym jest wartość jaka ma być przypisana do zmiennej wspomnianego typu. Wartość ta może być zapisana wprost lub w postaci stałej, wyrażenia albo zmiennej. Wartość zwracaną przez to makro przypisuje się zmiennej typu `atomic_t`.

`int atomic_read(const atomic_t *v)` - funkcja, która odczytuje wartość przekazanej jej przez adres zmiennej typu `atomic_t` i zwraca ją jako wartość typu `int`.

`void atomic_set(atomic_t *v, int i)` - funkcja, która nadaje zmiennej typu `atomic_t`, przekazanej jej przez adres, wartość typu `int` przekazaną jej przez drugi argument wywołania.

`void atomic_add(int i, atomic_t *v)` - funkcja dodaje do wartości zmiennej typu `atomic_t`, przekazanej jej przez adres, wartość typu `int` przekazaną jej jako pierwszy argument wywołania. Wynik dodawania jest zapisywany w zmiennej typu `atomic_t`.

`void atomic_sub(int i, atomic_t *v)` - funkcja odejmuje od wartości zmiennej `atomic_t` przekazanej jej przez adres, wartość typu `int` przekazaną jej przez pierwszy argument wywołania. Wynik tego odejmowania jest zapisywany w zmiennej typu `atomic_t`.

`void atomic_inc(atomic_t *v)` - funkcja ta zwiększa wartość przekazanej jej przez adres zmiennej typu `atomic_t` o jeden i nic nie zwraca.

`void atomic_dec(atomic_t *v)` - funkcja ta zwiększa wartość przekazanej jej przez adres zmiennej typu `atomic_t` o jeden i nic nie zwraca.

`int atomic_sub_and_test(int i, atomic_t *v)` - funkcja, która odejmuje liczbę przekazaną jej przez jej pierwszy parametr od wartości zmiennej typu `atomic_t` przekazanej jej przez adres za pomocą drugiego parametru. Wynik odejmowania zapisywany jest w zmiennej typu `atomic_t`, a funkcja zwraca wartość różną od zera, jeśli wynik tego odejmowania wynosi `0` i zero w przeciwnym przypadku.

`int atomic_add_negative(int i, atomic_t *v)` - funkcja dodaje liczbę przekazaną jej przez pierwszy parametr do zmiennej typu `atomic_t`, której adres jest jej przekazywany przez drugi parametr. Wynik dodawania zostaje zapisany w zmiennej typu `atomic_t`, a funkcja zwraca wartość różną od zera, jeśli był on ujemny, lub zero w przeciwnym przypadku.

`int atomic_add_return(int i, atomic_t *v)` - funkcja dodaje do liczby umieszczonej w zmiennej typu `atomic_t`, której adres jest przekazywany jej jako drugi argument wywołania, liczbę, która jest jej przekazana przez jej pierwszy parametr. Wynik jest zapisany w zmiennej typu `atomic_t` i zwrócony przez funkcję.

`int atomic_sub_return(int i, atomic_t *v)` - funkcja odejmuje od liczby umieszczonej w zmiennej typu `atomic_t`, której adres jest jej przekazany przez jej drugi parametr, liczbę, która jest jej przekazana przez pierwszy parametr. Wynik jest umieszczany w zmiennej typu `atomic_t` oraz zwracany przez funkcję.

`atomic_inc_return(v)` - makro, które zwiększa wartość przekazanej mu jako argument zmiennej typu `atomic_t` o jeden i zwraca wartość wynikową.

`atomic_dec_return(v)` - makro, które zmniejsza wartość przekazanej mu jako argument zmiennej typu `atomic_t` o jeden i zwraca wartość wynikową.

`int atomic_dec_and_test(atomic_t *v)` - funkcja zmniejsza wartość zmiennej typu `atomic_t`, której adres jest jej przekazany jako argument wywołania i zwraca zero, jeśli wynik jest różny od `0`, lub wartość różną od zera w przeciwnym przypadku.

`int atomic_inc_and_test(atomic_t *v)` - funkcja zwiększa wartość zmiennej typu `atomic_t`, której adres jest jej przekazany jako argument wywołania i zwraca zero, jeśli wynik jest różny od `0` lub wartość różną od zera w przeciwnym przypadku.

`atomic_inc_return(v)` - makro zwiększa o jeden wartość zmiennej typu `atomic_t` przekazanej mu jako argument i zwraca wynik tego działania.

`atomic_dec_return(v)` - makro zmniejsza o jeden wartość zmiennej typu `atomic_t` przekazanej mu jako argument i zwraca wynik tego działania.

Wszystkie wymienione funkcje są funkcjami `inline`. Istnieją także odpowiedniki tych podprogramów dla typu `atomic64_t`. Ich prototypu różnią się od prototypów opisanych podprogramów jedynie tym, że po wyrazie `atomic`, pisanym małymi lub wielkimi literami, występuje liczba `64`.

Operacje niepodzielne na pojedynczych bitach wykonywane są na słowach pamięci wskazywanych przez wskaźnik typu `void *` za pomocą następujących funkcji i makr:

`void set_bit(int nr, volatile void * addr)` - funkcja ustawia na jeden wartość bitu, którego pozycja jest jej przekazana przez jej pierwszy parametr, w słowie bitowym wskazywanym przez jej drugi parametr.

`void clear_bit(int nr, volatile void * addr)` - funkcja ustawia na zero wartość bitu, którego pozycja jest jej przekazana przez jej pierwszy parametr, w słowie bitowym wskazywanym przez jej drugi parametr.

`void change_bit(int nr, volatile void * addr)` - funkcja zmienia na przeciwną wartość bitu, którego pozycja jest jej przekazana przez jej pierwszy parametr, w słowie bitowym wskazywanym przez jej drugi parametr.

`int test_and_set_bit(int nr, volatile void * addr)` - funkcja ustawia na jeden wartość bitu, którego pozycja jest jej przekazana przez pierwszy parametr, w słowie bitowym wskazywanym przez drugi parametr i zwraca poprzednią wartość tego bitu.

`int test_and_clear_bit(int nr, volatile void * addr)` - funkcja ustawia na zero wartość bitu, którego pozycja jest jej przekazana przez pierwszy parametr, w słowie bitowym wskazywanym wskazywanym przez jej drugi parametr i zwraca poprzednią wartość tego bitu.

`int test_and_change_bit(int nr, volatile void * addr)` - funkcja zmienia na przeciwną wartość bitu, którego pozycja jest jej przekazana przez pierwszy parametr, w słowie bitowym wskazywanym przez drugi parametr i zwraca poprzednią wartość tego bitu.

`test_bit(nr,addr)` - makro, które zwraca wartość bitu w słowie bitowym wskazywanym przez jego drugi argument. Pozycja bitu jest określona jego pierwszym argumentem.

Funkcje, które przedstawiono w tym zestawieniu są funkcjami `inline`. Implementacja podprogramów wykonujących operacje niepodzielne na bitach jest zależna jest od platformy sprzętowej.

Listing 2 zawiera kod przykładowego modułu, w którym dwa wątki wykonują operację na zmiennej typu `atomic64_t`.

Listing 2: Przykładowy moduł prezentujący operacje niepodzielne

```
1  #include<linux/module.h>
2  #include<linux/kthread.h>
3  #include<linux/wait.h>
4  #include<linux/types.h>
5
6  enum thread_index {WAKING_THREAD, FIRST_THREAD, SECOND_THREAD};
7
8  static struct thread_structure
9  {
10         struct task_struct *thread[3];
11 } threads;
12
13 static wait_queue_head_t wait_queue;
14 static bool condition;
```

```
15 static atomic64_t number = ATOMIC64_INIT(0);
16
17 static int first_thread(void *data)
18 {
19         int counter = 0;
20         DEFINE_WAIT(wait);
21         for(;;) {
22                 pr_info("[first_thread] Number value: %ld\n",atomic64_read(&number));
23                 atomic64_inc(&number);
24                 if(counter%3) {
25                         add_wait_queue(&wait_queue,&wait);
26                         while(!condition) {
27                                 prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
28                                 if(kthread_should_stop())
29                                         return 0;
30                                 printk(KERN_INFO "[first_thread]: awake\n");
31                                 schedule();
32                         }
33                         condition=false;
34                         finish_wait(&wait_queue,&wait);
35                 }
36                 counter++;
37         }
38 }
39
40 static int second_thread(void *data)
41 {
42         int counter = 0;
43         DEFINE_WAIT(wait);
44         for(;;) {
45                 pr_info("[second_thread] Number value: %ld\n",atomic64_read(&number));
46                 atomic64_dec(&number);
47                 if(counter%7) {
48                         add_wait_queue(&wait_queue,&wait);
49                                 while(!condition) {
50                                         prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
51                                         if(kthread_should_stop())
52                                                 return 0;
53                                         printk(KERN_INFO "[second_thread]: awake\n");
54                                         schedule();
55                                 }
56                                 condition=false;
57                                 finish_wait(&wait_queue,&wait);
58                 }
59                 counter++;
60         }
61 }
62
63 static int waking_thread(void *data)
64 {
65         for(;;) {
66                 if(kthread_should_stop())
67                         return 0;
68                 set_current_state(TASK_INTERRUPTIBLE);
69                 if(schedule_timeout(1*HZ))
70                         printk(KERN_INFO "Signal received!\n");
71                 condition=true;
72                 wake_up_all(&wait_queue);
73         }
74
```

```
75  }
76
77  static int __init threads_init(void)
78  {
79          init_waitqueue_head(&wait_queue);
80          threads.thread[FIRST_THREAD] = kthread_run(first_thread,NULL,"first_thread");
81          threads.thread[SECOND_THREAD] = kthread_run(second_thread,NULL,"second_thread");
82          threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
83          return 0;
84  }
85
86  static void __exit threads_exit(void)
87  {
88          kthread_stop(threads.thread[WAKING_THREAD]);
89          kthread_stop(threads.thread[SECOND_THREAD]);
90          kthread_stop(threads.thread[FIRST_THREAD]);
91  }
92
93  module_init(threads_init);
94  module_exit(threads_exit);
95
96  MODULE_LICENSE("GPL");
97  MODULE_DESCRIPTION("An example of using the Linux kernel threads and the atomic64_t data type.");
98  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

W module tworzone są trzy wątki, jeden aktywowany upływem czasu i dwa reagujące na zdarzenie, jakim jest pobudzenie ich do działania przez wątek aktywowany czasowo. W porównaniu z poprzednim modułem zmieniła się liczba tych ostatnich wątków. To w ramach funkcji wykonywanych w tych wątkach przeprowadzane są operacje na zmiennej typu `atomic64_t`. Sama zmienna tego typu jest zadeklarowana w wierszu nr 15 modułu. W funkcji wątku, o nazwie `first_thread()`, jej wartość jest odczytywana i umieszczana w buforze jądra w wierszu nr 22, a następnie zwiększana o jeden w wierszu nr 23. W drugiej funkcji wątku, o nazwie `second_thread()`, wartość tej zmiennej również jest odczytywana i umieszczana w buforze jądra (wiersz nr 45), ale w kolejnym wierszu jest zmniejszana o jeden (wiersz nr 46). Aby zwiększyć intensywność operacji wykonywanych na tej zmiennej, wątek realizujący funkcję `first_thread()` jest ustawiany w stan oczekiwania co trzy iteracje pętli `for`, a wątek realizujący funkcję `second_thread()`, co siódmą iterację. Efekt ten osiągnięto deklarując w każdej funkcji zmienną lokalną `counter`, zwiększając jej wartość po każdej iteracji pętli `for` i usypiając dany wątek, kiedy jej wartość jest podzielna, odpowiednio, przez `3` lub przez `7`. (wiersze nr 24 i 47). Pozostała część kodu jest podobna do kodu pierwszego modułu zaprezentowanego w tej instrukcji.

2.2. Muteksy

Muteksy w jądrze Linuksa są semaforami binarnymi, których implementacja jest niezależna od platformy sprzętowej, na której są używane. Zazwyczaj są one używane do ochrony przez sytuacjami hazardowymi zasobów o złożonych typach, takich jak struktury, ale mogą być również wykorzystane do ochrony zmiennych prostych typów. Aby móc w module używać tych środków synchronizacji należy włączyć do kodu modułu plik nagłówkowy `linux/mutex.h`. Jest w nim zdefiniowany typ strukturalny `struct mutex` służący do deklarowania muteksów, oraz następujące podprogramy, które operują na nich:

`DEFINE_MUTEX(mutexname)` - makro, które definiuje i inicjuje jako niezajęty muteks, o nazwie, która jest przekazana jako jego argument.

`mutex_init(mutex)` - makro, które inicjuje muteks o adresie przekazanym za pomocą jego argumentu, jako niezajęty.

`mutex_lock_interruptible(lock)` - makro zajmuje muteks o adresie przekazanym mu przez argument. Jeśli próba zajęcia się nie powiedzie, to wątek, który używa tego makra jest wprowadzany w stan oczekiwania `TASK_INTERRUPTIBLE`.

`mutex_lock(lock)` - makro zajmuje muteks o adresie przekazanym mu przez argument. Jeśli próba zajęcia się nie powiedzie, to wątek, który używa tego makra jest wprowadzany w stan oczekiwania `TASK_UNINTERRUPTIBLE`. Makro to zwraca wartość różną od zera, jeśli wątek został aktywowany z innego powodu, niże ten na który czekał.

`mutex_lock_killable(lock)` - makro zajmuje muteks o adresie przekazanym mu przez argument. Jeśli próba zajęcia się nie powiedzie, to wątek, który używa tego makra jest wprowadzany w stan oczekiwania `TASK_KILLABLE`. Makro to zwraca wartość różną od zera, jeśli wątek został aktywowany z innego powodu, niż ten na który czekał.

`int mutex_trylock(struct mutex *lock)` - funkcja, która próbuje zająć muteks o adresie przekazanym jej przez argument wywołania. Jeśli ta próba się nie powiedzie funkcja zwraca zero, a w przeciwnym przypadku jeden. Funkcja nie wprowadza w stan oczekiwania wątku, który ją wywołał.

`void mutex_unlock(struct mutex *lock)` - funkcja zwalnia muteks o adresie przekazanym jej przez argument wywołania.

`int mutex_is_locked(struct mutex *lock)` - funkcja zwraca jeden, jeśli muteks, którego adres jest jej przekazany jako argument wywołania jest zajęty przez wątek, lub zero w przeciwnym przypadku.

Listing 3 zawiera kod źródłowy modułu będącego modyfikacją modułu z listingu 2, w którym zamiast operacji niepodzielnych na zmiennej `atomic64_t` użyto muteksów do synchronizacji dostępu do zmiennej typu `int`.

Listing 3: Przykładowy moduł prezentujący działanie muteksów

```
#include<linux/module.h>
#include<linux/kthread.h>
#include<linux/wait.h>
#include<linux/mutex.h>

enum thread_index {WAKING_THREAD, FIRST_THREAD, SECOND_THREAD};

static struct thread_structure
{
        struct task_struct *thread[3];
} threads;

static wait_queue_head_t wait_queue;
static bool condition;
static DEFINE_MUTEX(number_lock);
static int number;

static int first_thread(void *data)
{
        int counter = 0;
        DEFINE_WAIT(wait);
        for(;;) {
                mutex_lock(&number_lock);
                pr_info("[first_thread] Number value: %d\n",number);
                number++;
                mutex_unlock(&number_lock);
                if(counter%3) {
                        add_wait_queue(&wait_queue,&wait);
                        while(!condition) {
                                prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                                if(kthread_should_stop())
                                        return 0;
                                printk(KERN_INFO "[first_thread]: awake\n");
                                schedule();
```

```
                        }
                        condition=false;
                        finish_wait(&wait_queue,&wait);
                }
                counter++;
        }
}

static int second_thread(void *data)
{
        int counter = 0;
        DEFINE_WAIT(wait);
        for(;;) {
                mutex_lock(&number_lock);
                pr_info("[second_thread] Number value: %d\n",number);
                number--;
                mutex_unlock(&number_lock);
                if(counter%7) {
                        add_wait_queue(&wait_queue,&wait);
                                while(!condition) {
                                        prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                                        if(kthread_should_stop())
                                                return 0;
                                        printk(KERN_INFO "[second_thread]: awake\n");
                                        schedule();
                                }
                                condition=false;
                                finish_wait(&wait_queue,&wait);
                }
                counter++;
        }
}

static int waking_thread(void *data)
{
        for(;;) {
                if(kthread_should_stop())
                        return 0;
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(1*HZ))
                        printk(KERN_INFO "Signal received!\n");
                condition=true;
                wake_up_all(&wait_queue);
        }

}

static int __init threads_init(void)
{
        init_waitqueue_head(&wait_queue);
        threads.thread[FIRST_THREAD] = kthread_run(first_thread,NULL,"first_thread");
        threads.thread[SECOND_THREAD] = kthread_run(second_thread,NULL,"second_thread");
        threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
        return 0;
}

static void __exit threads_exit(void)
{
        kthread_stop(threads.thread[WAKING_THREAD]);
        kthread_stop(threads.thread[SECOND_THREAD]);
```

```
95          kthread_stop(threads.thread[FIRST_THREAD]);
96  }
97
98  module_init(threads_init);
99  module_exit(threads_exit);
100
101 MODULE_LICENSE("GPL");
102 MODULE_DESCRIPTION("An example of using the Linux kernel threads and a mutex.");
103 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

Różnic między zaprezentowanym w listingu 3, a tym opisanym wcześniej jest kilka. Przede wszystkim w wierszu nr 15 zdefiniowano muteks o nazwie `number_lock`, a w wierszu nr 16 zadeklarowano zmienną `number` o typie `int`, na której oba wątki aktywowane zdarzeniami będą wykonywały operacje. W wątku wykonującym funkcję `first_thread()`, w pętli `for`, w wierszu nr 23 zajmowany jest wspomniany muteks. Jeśli ta operacja się powiedzie, to wątek umieszcza w buforze jądra wartość zmiennej `number` (wiersz nr 24), a natępnie zwiększa wartość tej zmiennej o jeden (wiersz nr 25), po czym zwalnia muteks (wiersz nr 26). Jeśli zajęcie muteksa się nie powiedzie, to wątek będzie musiał poczekać z wykonaniem operacji z wierszy 24-26 do czasu zwolnienia muteksa przez drugi wątek. Drugi z wątków, realizujący funkcję `second_thread()` również zajmuje muteks i umieszcza wartość zmiennej `number` w buforze jądra, ale potem jej wartość zmniejsza o jeden (wiersz nr 50), a dopiero następnie zwalnia muteks. Podobnie jak w przypadku pierwszego wątku, jeśli nie uda mu się zająć muteksa, to przechodzi w stan oczekiwania na jego zwolnienie. Reszta kodu modułu jest podobna do tego zaprezentowanego w listingu 2.

2.3. Zmienne sygnałowe

Zmienne sygnałowe są uproszczoną wersją semaforów, używaną, kiedy synchronizacja realizowana jest według scenariusza, w którym wątek musi poczekać na wynik pracy innego wątku. Typem zmiennej sygnałowej jest typ strukturalny `struct completion` zdefiniowany w pliku nagłówkowym `linux/completion.h` W nim również zadeklarowane lub zdefiniowane są następujące podprogramy realizujące niepodzielne operacje na takich zmiennych:

`DECLARE_COMPLETION(work)` - makro, które deklaruje i inicjuje zmienną sygnałową, o nazwie podanej jako jego argument.

`void init_completion(struct completion *x)` - funkcja `inline`, która inicjuje zmienną sygnałową, której adres jest jej przekazany jako argument jej wywołania.

`void wait_for_completion(struct completion *)` - funkcja wywoływana przez wątek oczekujący na zakończenie pracy przez inny wątek. Jako argument wywołania przyjmuje ona adres zmiennej sygnałowej. Funkcja wprowadza wątek ją wywołujący w stan `TASK_UNINTERRUPTIBLE`.

`int wait_for_completion_interruptible(struct completion *x)` - funkcja wywoływana przez wątek oczekujący na zakończenie pracy przez inny wątek. Jako argument wywołania przyjmuje ona adres zmiennej sygnałowej. Funkcja wprowadza wątek ją wywołujący w stan `TASK_INTERRUPTIBLE`. Jeśli wątek zostanie aktywowany przez inne zdarzenie, niż to na które oczekuje, o funkcja zwróci wartość różną od zera, a zero w przeciwnym przypadku.

`int wait_for_completion_killable(struct completion *x)` - funkcja wywoływana przez wątek oczekujący na zakończenie pracy przez inny wątek. Jako argument wywołania przyjmuje ona adres zmiennej sygnałowej. Funkcja wprowadza wątek ją wywołujący w stan `TASK_KILLABLE`. Jeśli wątek zostanie aktywowany przez inne zdarzenie, niż to na które oczekuje, o funkcja zwróci wartość różną od zera, a zero w przeciwnym przypadku.

`unsigned long wait_for_completion_timeout(struct completion *x, unsigned long timeout)` - funkcja ta działa podobnie jak `wait_for_completion()`, ale przyjmuje dodatkowy argument wywołania, który określa po jakim czasie oczekiwanie zostanie zakończone, jeśli inny wątek nie zakończy swojej pracy. Zwraca ona `0` jeśli czas oczekiwania upłynie lub czas pozostały do zakończenia oczekiwania,

jeśli inny wątek zasygnalizuje, że zakończył swoją pracę. Zarówno czas oczekiwania, który przekazany jest przez argument wywołania tej funkcji, jak i czas zwrócony przez nią jest mierzony w taktach zegara systemowego.

`long wait_for_completion_interruptible_timeout(struct completion *x, unsigned long timeout)` - funkcja ta działa podobnie jak `wait_for_completion_interruptible()`, ale przyjmuje dodatkowy argument wywołania, który określa po jakim czasie oczekiwanie zostanie zakończone, jeśli inny wątek nie zakończy swojej pracy. Zwraca ona `0` jeśli czas oczekiwania upłynie, liczbę ujemną, jeśli wątek zostanie aktywowany przez inne zdarzenie, niż to na które oczekiwał lub czas pozostały do zakończenia oczekiwania, jeśli inny wątek zasygnalizuje, że zakończył swoją pracę. Zarówno czas oczekiwania, który przekazany jest przez argument wywołania tej funkcji, jak i czas zwrócony przez nią jest mierzony w taktach zegara systemowego.

`long wait_for_completion_killable_timeout(struct completion *x, unsigned long timeout)` - funkcja ta działa podobnie jak `wait_for_completion_killable()`, ale przyjmuje dodatkowy argument wywołania, który określa po jakim czasie oczekiwanie zostanie zakończone, jeśli inny wątek nie zakończy swojej pracy. Zwraca ona `0` jeśli czas oczekiwania upłynie, liczbę ujemną, jeśli wątek zostanie aktywowany przez inne zdarzenie, niż to na które oczekiwał lub czas pozostały do zakończenia oczekiwania, jeśli inny wątek zasygnalizuje, że zakończył swoją pracę. Zarówno czas oczekiwania, który przekazany jest przez argument wywołania tej funkcji, jak i czas zwrócony przez nią jest mierzony w taktach zegara systemowego.

`void complete(struct completion *)` - funkcja wywoływana przez wątek po zakończeniu pracy, na której koniec czekał inny wątek. Przyjmuje ona jako argument wywołania adres zmiennej sygnałowej.

`void complete_all(struct completion *)` - funkcja wywoływana przez wątek po zakończeniu pracy, aby powiadomić inne wątki, które czekały na to zakończenie. Przyjmuje ona jako argument wywołania adres zmiennej sygnałowej.

Listing 4 zawiera kod źródłowy przykładowego modułu w którym wątki używają zmiennej sygnałowej do zsynchronizowania swojej pracy.

Listing 4: Przykładowy moduł prezentujący działanie zmiennych sygnałowych

```
1  #include<linux/module.h>
2  #include<linux/kthread.h>
3  #include<linux/wait.h>
4  #include<linux/completion.h>
5
6  enum thread_index {WAKING_THREAD, WRITER_THREAD, READER_THREAD};
7
8  static struct thread_structure
9  {
10         struct task_struct *thread[3];
11 } threads;
12
13 static wait_queue_head_t wait_queue;
14 static bool condition;
15 static DECLARE_COMPLETION(number_completion);
16 static int number;
17
18 static int writer_thread(void *data)
19 {
20         DEFINE_WAIT(wait);
21         for(;;) {
22                 number++;
23                 complete(&number_completion);
24                 add_wait_queue(&wait_queue,&wait);
25                 while(!condition) {
```

```
                        prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                        if(kthread_should_stop())
                                return 0;
                        printk(KERN_INFO "[writer_thread]: awake\n");
                        schedule();
                }
                condition=false;
                finish_wait(&wait_queue,&wait);
        }
}

static int reader_thread(void *data)
{
        DEFINE_WAIT(wait);
        for(;;) {
                wait_for_completion(&number_completion);
                pr_info("[reader_thread] Number value: %d\n",number);
                if(kthread_should_stop())
                        return 0;
                schedule();
        }
}

static int waking_thread(void *data)
{
        for(;;) {
                if(kthread_should_stop())
                        return 0;
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(1*HZ))
                        printk(KERN_INFO "Signal received!\n");
                condition=true;
                wake_up(&wait_queue);
        }

}

static int __init threads_init(void)
{
        init_waitqueue_head(&wait_queue);
        threads.thread[READER_THREAD] = kthread_run(reader_thread,NULL,"reader_thread");
        threads.thread[WRITER_THREAD] = kthread_run(writer_thread,NULL,"writer_thread");
        threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
        return 0;
}

static void __exit threads_exit(void)
{
        kthread_stop(threads.thread[READER_THREAD]);
        kthread_stop(threads.thread[WAKING_THREAD]);
        kthread_stop(threads.thread[WRITER_THREAD]);
}

module_init(threads_init);
module_exit(threads_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("An example of using the Linux kernel threads and a completion variable.");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

Kod źródłowy tego modułu, choć bazuje na innych zaprezentowanych w tej instrukcji, to jednak znacząco się od nich różni. Role wątków aktywowanych zdarzeniami zmieniły się. Do tej pory te wątki były równoprawne. Teraz jeden z nich został pisarzem (ang. *writer*), a drugi czytelnikiem (ang. *reader*). Zostało to odzwierciedlone w nazwach elementów typu wyliczeniowego `thread_index` w wierszu nr 6. Jednak najważniejsze zmiany zaszły w funkcjach wątków, które teraz nazywają się `wirter_thread()` i `reader_thread()`. Obie korzystają ze zmiennej sygnałowej, która została zadeklarowana i zainicjowana w wierszu nr 15. Użycie tej zmiennej wymaga także włączenia pliku nagłówkowego `linux/completion.h` (wiersz nr 4). Wątek pisarza zwiększa wartość zmiennej `number` (wiersz nr 22), która jest współdzielona przez oba wątki i sygnalizuje zakończenie swojej pracy (wiersz nr 23). Następnie przechodzi w stan oczekiwania, podobnie jak inne opisywane do tej pory wątki. Funkcja realizowana przez wątek czytelnika jest znacznie prostsza niż inne prezentowane dotąd funkcje wątków. W pętli `for` oczekuje ona na zakończenie pracy przez wątek pisarza (wiersz nr 41), a następnie wypisuje wartość zmiennej `number`, sprawdza, czy wątek w ramach którego jest realizowana, nie powinien się zakończyć i wywołuje funkcję `schedule()`. Reszta kodu modułu jest podobna do innych, wcześniej zaprezentowanych modułów.

2.4. Blokady sekwencyjne

Blokady sekwencyjne są przykładem mechanizmów synchronizacji zoptymalizowanych dla problemu czytelników i pisarzy, w którym priorytet mają pisarze. Są to zmienne, które pełnią rolę liczników. Wątek pisarza zwiększa wartość tego licznika przez rozpoczęciem modyfikacji zasobu i po jej zakończeniu. Wątek czytelnika odczytuje wartość tej zmiennej przed odczytem stanu zasobu i po jego zakończeniu. Jeśli czytelnik otrzyma dwie takie same wartości, to oznacza to, że operacja odczytu nie została przepleciona z operacją zapisu. Tylko w takim przypadku odczytany stan zasobu jest prawidłowy. Jeśli te dwie wartości różnią się, to należy powtórzyć odczyt. Blokady te używane są w przypadkach, kiedy liczba modyfikacji zasobu przewyższa liczbę odczytów.

Blokady sekwencyjne są zmiennymi typu `seqlock_t`, który wraz z podprogramami obsługującymi te zmienne zdefiniowany jest w pliku nagłówkowym `linux/seqlock.h`. Wśród tych podprogramów znajdują się:

`DEFINE_SEQLOCK(x)` - makro, które i inicjuje blokadę sekwencyjną, o nazwie przekazanej mu jako argument.

`unsigned read_seqbegin(const seqlock_t *sl)` - funkcja `inline`, którą wywołuje czytelnik przez odczytem zasobu współdzielonego. Przyjmuje ona jako argument wywołania adres blokady sekwencyjnej i zwraca jej wartość.

`unsigned read_seqretry(const seqlock_t *sl, unsigned start)` - funkcja `inline`, którą czytelnik wywołuje po odczycie współdzielonego zasobu. Przyjmuje ona jako pierwszy argument wywołania adres blokady sekwencyjnej, a jako drugi wartość zwróconą przez `read_seqbegin()`. Funkcja zwraca zero, jeśli wartość blokady odczytana przez nią jest równa wartości otrzymanej przez drugi argument wywołania. W przeciwnym przypadku zwraca ona wartość różną od zera.

`void write_seqlock(seqlock_t *sl)` - funkcja `inline` wywoływana przez pisarza przed wykonaniem modyfikacji zasobu współdzielonego. Zwiększa ona o jeden wartość blokady sekwencyjnej, której adres jest jej przekazany jako argument wywołania.

`void write_sequnlock(seqlock_t *sl)` - funkcja `inline` wywoływana przez pisarza po wykonaniu modyfikacji zasobu współdzielonego. Zwiększa ona o jeden wartość blokady sekwencyjnej, której adres jest jej przekazany jako argument wywołania.

Listing 5 przedstawia kod źródłowy przykładowego modułu, w którym wątki korzystają z mechanizmu blokady sekwencyjnej.

Listing 5: Przykładowy moduł prezentujący działanie blokad sekwencyjnych

```
1 #include<linux/module.h>
2 #include<linux/kthread.h>
3 #include<linux/wait.h>
```

```
4  #include<linux/seqlock.h>
5
6  enum thread_index {WAKING_THREAD, WRITER_THREAD, FIRST_READER_THREAD, SECOND_READER_THREAD};
7
8  static struct thread_structure
9  {
10         struct task_struct *thread[4];
11 } threads;
12
13 static wait_queue_head_t wait_queue;
14 static bool condition;
15 static int number;
16 static DEFINE_SEQLOCK(number_lock);
17 static const int first_reader_number = 1, second_thread_number = 2;
18
19 static int reader_thread(void *data)
20 {
21         DEFINE_WAIT(wait);
22         unsigned long int seqlock_value = 0;
23         int local_number = 0;
24         for(;;) {
25                 do {
26                                 seqlock_value = read_seqbegin(&number_lock);
27                                 local_number = number;
28                 } while(read_seqretry(&number_lock, seqlock_value));
29                 pr_info("[reader_number: %d] Value of \"number\" variable: %d\n",
30                                                         *(int *)data,local_number);
31                 add_wait_queue(&wait_queue,&wait);
32                 while(!condition) {
33                         prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
34                         if(kthread_should_stop())
35                                 return 0;
36                         pr_info("[reader_thread %d]: awake\n", *(int *)data);
37                         schedule();
38                 }
39                 condition=false;
40                 finish_wait(&wait_queue,&wait);
41         }
42 }
43
44 static int writer_thread(void *data)
45 {
46         for(;;) {
47                 write_seqlock(&number_lock);
48                 number++;
49                 write_sequnlock(&number_lock);
50                 if(kthread_should_stop())
51                         return 0;
52                 set_current_state(TASK_INTERRUPTIBLE);
53                 if(schedule_timeout(HZ>>2))
54                         pr_info("Signal received!\n");
55                 pr_info("[writer_thread] awake!\n");
56         }
57 }
58
59 static int waking_thread(void *data)
60 {
61         for(;;) {
62                 if(kthread_should_stop())
63                         return 0;
```

```
64                  set_current_state(TASK_INTERRUPTIBLE);
65                  if(schedule_timeout(1*HZ))
66                          pr_info("Signal received!\n");
67                  condition=true;
68                  wake_up_all(&wait_queue);
69          }
70
71  }
72
73  static int __init threads_init(void)
74  {
75          init_waitqueue_head(&wait_queue);
76          threads.thread[WRITER_THREAD] = kthread_run(writer_thread,NULL,"writer_thread");
77          threads.thread[FIRST_READER_THREAD] =
78                  kthread_run(reader_thread,(void *)&first_reader_number,"first_reader_thread");
79          threads.thread[SECOND_READER_THREAD] =
80                  kthread_run(reader_thread,(void *)&second_thread_number,"second_reader_thread");
81          threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
82          return 0;
83  }
84
85  static void __exit threads_exit(void)
86  {
87          kthread_stop(threads.thread[WAKING_THREAD]);
88          kthread_stop(threads.thread[WRITER_THREAD]);
89          kthread_stop(threads.thread[FIRST_READER_THREAD]);
90          kthread_stop(threads.thread[SECOND_READER_THREAD]);
91  }
92
93  module_init(threads_init);
94  module_exit(threads_exit);
95
96  MODULE_LICENSE("GPL");
97  MODULE_DESCRIPTION("An example of using the Linux kernel threads and a seqential lock.");
98  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

Budowa kodu źródłowego tego modułu przypomina kod źródłowy modułu z listingu 4, jednak zawiera on kilka istotnych zmian. Pierwszą jest włączenie pliku nagłówkowego do obsługi blokad sekwencyjnych (wiersz nr 4). Ten moduł będzie tworzył nie jeden lecz dwa wątki czytelników, co wymaga dodania elementu do tablicy deskryptorów w strukturze `thread_structure` (wiersz nr 10), oraz zmiany typu wyliczeniowego `thread_index` (wiersz nr 6). Zmienną, do której dostęp będzie podlegał ochronie przez blokadę sekwencyjną jest tak jak poprzednio zmienna `number` (wiersz nr 15). Blokada sekwencyjna jest zadeklarowana i zainicjowana w wierszu nr 16. W wierszu nr 17 zdefiniowano dwie stałe o wartościach 1 i 2. Ponieważ wątki czytelników będą realizowały tę samą funkcję (`read_thread()`), to aby je odróżnić będą im przez argument wywołania przekazane adresy tych stałych (wiersze 78 i 80). Wspomniana funkcja ma dwie nowe zmienne lokalne `seqlock_value` typu `unsigned long int` oraz `local_number` typu `int`. Pierwsza będzie służyła do przechowywania wartości blokady sekwencyjnej odczytanej przez funkcję `read_seqbegin()`, a druga do przechowania wartości odczytanej ze zmiennej `number`. W wierszach 25-28 wykonywana jest pętla `do…while()`, w której odczytywana i zapamiętywana jest wartość początkowa blokady sekwencyjnej (wiersz nr 26) oraz wartość zmiennej `number`. Następnie wartość blokady sekwencyjnej jest ponownie odczytywana i porównywana z jej wartością początkową w wierszu nr 28, przy pomocy funkcji `read_seqretry()`. Wynik tego porównania jest równocześnie warunkiem zakończenia wspomnianej pętli `do…while`. Jeśli będzie on różny od zera, to odczyt zostanie powtórzony - pętla wykonana kolejną iterację, jeśli nie, to jej działanie zakończy się. Po skończeniu tej pętli funkcja umieszcza w buforze jądra komunikat zawierający odczytaną wartość zmiennej `number` oraz identyfikator wątku przekazany funkcji przez parametr `data`. Wątek pisarza realizuje funkcję `writer_thread()`. Przypomina ona tę z listingu 4, jednak tym razem usypia ona wątek na określony czas, a dokładniej na $\frac{1}{4}$ sekundy. Zatem wątek pisarza jest czterokrotnie częściej aktywowany niż oba wątki czytelników.

Pisarz wywołuje w pętli `for` obie opisane wcześniej funkcje przeznaczone dla niego (wiersze nr 47 i 49) oraz zwiększa wartość zmiennej `number` o jeden (wiersz nr 48).

2.5. Mechanizm RCU

Nazwa mechanizmu RCU pochodzi od pierwszych liter angielskich wyrazów *Read-Copy-Update*. Pozwala on rozwiązać problem synchronizacji dostępu do zasobu współdzielonego, jeśli sprowadza się on do problemu czytelników i pisarzy, w którym priorytet mają czytelnicy. Aby mógł on być zastosowany musi być spełnionych kilka warunków:

1. chroniony zasób musi być dostępny dla wątków tylko za pomocą wskaźnika,
2. wątki czytelników w sekcji krytycznej nie mogą przejść w stan oczekiwania, czyli ich wykonanie nie może zostać zawieszone,
3. zapisy chronionego zasobu powinny być sporadyczne, a odczyty częste.

Mechanizm RCU pozwala na efektywną synchronizację dostępu do zasobu współdzielonego, kosztem pewnego narzutu pamięci operacyjnej. Należy pamiętać, że jeśli jest więcej niż jeden pisarz, to ich dostęp do zasobu nie jest synchronizowany przez ten mechanizm i należy użyć jeszcze innego dodatkowego mechanizmu, aby taką synchronizację stworzyć.

Zasada działania mechanizmu RCU polega na tym, że wątek, który jest czytelnikiem pozyskuje wskaźnik na zasób, który chce odczytać, odczytuje go i sygnalizuje zakończenie odczytu. Wątek pisarz, jeśli chce zmodyfikować zasób, to tworzy jego kopię i ją zapisuje, a następnie upublicznia wskaźnik do tej kopii. Oryginalny zasób jest likwidowany dopiero wtedy, gdy wszyscy czytelnicy zakończą jego odczyt. Kolejne odczyty będą dotyczyły już kopii.

Podprogramy odpowiedzialne za obsługę mechanizmu RCU są zdefiniowane lub zadeklarowane lub zdefiniowane w pliku nagłówkowym `linux/rcupdate.h`. Należą do nich między innymi:

`void rcu_read_lock(void)` - funkcja `inline`, która wywoływana jest przez czytelnika przed rozpoczęciem sekcji krytycznej (odczytem zasobu).

`void rcu_read_unlock(void)` - funkcja `inline`, która wywoływana jest przez czytelnika po zakończeniu sekcji krytycznej (odczytu zasobu).

`rcu_dereference(p)` - makro, które pozwala czytelnikowi uzyskać dostęp do współdzielonego zasobu poprzez dereferencję głównego wskaźnika do zasobu, jaki jest mu przekazywany przez argument.

`rcu_assign_pointer(p, v)` - makro, które pozwala pisarzowi upublicznić wskaźnik na nową wersję (kopię) zmodyfikowanego zasobu. Jego pierwszym argumentem jest główny wskaźnik do zasobu, a drugim wskaźnik na kopię tego zasobu.

`void synchronize_rcu(void)` - funkcja, która wstrzymuje działanie pisarza tak długo, aż wszyscy czytelnicy skończą korzystać z poprzedniej wersji zasobu współdzielonego.

`void call_rcu(struct rcu_head *head, rcu_callback_t func)` - funkcja wykorzystywana przez pisarza do zarejestrowania funkcji, która będzie wywołana po tym, kiedy czytelnicy skończą korzystanie z bieżącej wersji zasobu współdzielonego. Funkcja ta może usunąć tę wersję zasobu. Najczęściej to rozwiązanie wykorzystywane jest wtedy, gdy chronionym zasobem jest struktura. Ta struktura musi zawierać pole typ `struct rcu_head`. Funkcja wywoływana po zaprzestaniu przez czytelników korzystania z zasobu powinna posiadać jeden parametr będący wskaźnikiem na strukturę typu `struct rcu_head` i nie zwracać żadnej wartości. Wskaźnik na strukturę, którą ma zwolnić może ona uzyskać ze wskaźnika typu `struct rcu_head` przy pomocy makra `container_of`[3]. Funkcja `call_rcu` przyjmuje dwa argumenty wywołania: wskaźnik na strukturę typu `struct rcu_head` i wskaźnik na funkcję wywoływaną po zakończeniu przez czytelników korzystania ze współdzielonego zasobu.

Listing 6 zawiera kod źródłowy modułu, w którym wątki korzystają z mechanizmu RCU.

[3]Ma ono podobne argumenty jak `list_entry`, opisane w instrukcji o strukturach danych jądra.

Listing 6: Przykładowy moduł prezentujący działanie mechanizmu RCU

```
#include<linux/module.h>
#include<linux/kthread.h>
#include<linux/wait.h>
#include<linux/slab.h>
#include<linux/rcupdate.h>

enum thread_index {WAKING_THREAD, WRITER_THREAD, FIRST_READER_THREAD, SECOND_READER_THREAD};

static struct thread_structure
{
        struct task_struct *thread[4];
} threads;

static wait_queue_head_t wait_queue;
static bool condition;
static int *number_pointer;
static const int first_reader_number = 1, second_thread_number = 2;

static int reader_thread(void *data)
{
        int *local_number_pointer = NULL;
        for(;;) {
                rcu_read_lock();
                local_number_pointer = rcu_dereference(number_pointer);
                if(local_number_pointer)
                        pr_info("[reader_number: %d] Value of \"number\" variable: %d\n",
                                                        *(int *)data,*local_number_pointer);
                rcu_read_unlock();
                if(kthread_should_stop())
                        return 0;
                set_current_state(TASK_INTERRUPTIBLE);
                if(schedule_timeout(HZ>>2))
                        pr_info("Signal received!\n");
        }
}

static int writer_thread(void *data)
{
        int *local_number_pointer = NULL;
        int number = 0;
        DEFINE_WAIT(wait);
        for(;;) {
                void *old_pointer = NULL;
                local_number_pointer = kmalloc(sizeof(int),GFP_KERNEL);
                if(IS_ERR(local_number_pointer)) {
                        pr_alert("Error allocating memory: %ld\n",PTR_ERR(local_number_pointer));
                        return 0;
                }
                *local_number_pointer = number++;
                old_pointer = number_pointer;
                rcu_assign_pointer(number_pointer,local_number_pointer);
                synchronize_rcu();
                if(old_pointer)
                        kfree(old_pointer);
                add_wait_queue(&wait_queue,&wait);
                while(!condition) {
                        prepare_to_wait(&wait_queue,&wait,TASK_INTERRUPTIBLE);
                        if(kthread_should_stop())
```

```
59                                  return 0;
60                          pr_info("[writer_thread]: awake\n");
61                          schedule();
62                  }
63                  condition=false;
64                  finish_wait(&wait_queue,&wait);
65          }
66  }
67
68  static int waking_thread(void *data)
69  {
70          for(;;) {
71                  if(kthread_should_stop())
72                          return 0;
73                  set_current_state(TASK_INTERRUPTIBLE);
74                  if(schedule_timeout(HZ))
75                          pr_info("Signal received!\n");
76                  condition=true;
77                  wake_up(&wait_queue);
78          }
79
80  }
81
82  static int __init threads_init(void)
83  {
84          init_waitqueue_head(&wait_queue);
85          threads.thread[WRITER_THREAD] = kthread_run(writer_thread,NULL,"writer_thread");
86          threads.thread[WAKING_THREAD] = kthread_run(waking_thread,NULL,"waking_thread");
87          threads.thread[FIRST_READER_THREAD] =
88                  kthread_run(reader_thread,(void *)&first_reader_number,"first_reader_thread");
89          threads.thread[SECOND_READER_THREAD] =
90                  kthread_run(reader_thread,(void *)&second_thread_number,"second_reader_thread");
91          return 0;
92  }
93
94  static void __exit threads_exit(void)
95  {
96          kthread_stop(threads.thread[WAKING_THREAD]);
97          kthread_stop(threads.thread[WRITER_THREAD]);
98          kthread_stop(threads.thread[FIRST_READER_THREAD]);
99          kthread_stop(threads.thread[SECOND_READER_THREAD]);
100 }
101
102 module_init(threads_init);
103 module_exit(threads_exit);
104
105 MODULE_LICENSE("GPL");
106 MODULE_DESCRIPTION("An example of using the Linux kernel threads and an RCU mechanism.");
107 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
```

Kod tego modułu jest modyfikacja kodu z listingu 5. Tym razem to wątek pisarza jest aktywowany co sekundę, a wątki czytelników co $\frac{1}{4}$ sekundy. Do modułu, oprócz pliku nagłówkowego `linux/rcupdate.h` (wiersz nr 5) jest także dołączany plik nagłówkowy `linux/slab.h` (wiersz nr 6), ponieważ zasób współdzielony przez wątki będzie zmienną typu `int`, na którą pamięć będzie przydzielana dynamicznie przy pomocy funkcji `kmalloc()` i zwalniana za pomocą `kfree()`. W wierszu nr 16 zdefiniowano główny wskaźnik do współdzielonego zasobu. Funkcja wątku wykonywana przez wątki-czytelników zdefiniowana jest w wierszach 19-35 kodu modułu. W wierszu nr 21 zdefiniowany jest lokalny dla tej funkcji wskaźnik, do którego zapisywany będzie uzyskany adres współdzielonego zasobu. Zanim to jednak nastąpi funkcja wywołuje `rcu_read_lock()` celem zasygnalizowania pisarzowi, że rozpoczyna odczyt zasobu. Następnie pozyskuje ona wskaźnik na ten zasób (wiersz nr 24) i sprawdza, czy nie jest to wskaźnik pusty. Jeśli

ten warunek jest prawdziwy, to umieszcza w buforze jądra odczytaną wartość zasobu, wraz z identyfikatorem wątku, który został jej przekazany przez parametr `data` (wiersze nr 26 i 27), a następnie wywołuje `rcu_read_unlock()` celem poinformowania pisarza, że zakończyła odczyt zasobu. Funkcja `writer_thread()` wykonywana w ramach wątku pisarza jest bardziej skomplikowana. Wierszu nr 40 zadeklarowana jest zmienna `number`, której wartości będą nadawane kolejnym kopiom zasobu współdzielonego. W wierszu nr 39 zdefiniowany jest lokalny wskaźnik na utworzoną kopię zasobu współdzielonego. Wewnątrz pętli `for` w wierszu nr 43 zadeklarowany jest wskaźnik, w którym zostanie zapisany adres poprzedniej kopii zasobu współdzielonego. W wierszu nr 44 wątek pisarza tworzy nową kopię zasobu współdzielonego przydzielając na nią pamięć przy pomocy wywołania funkcji `kmalloc()`. Jeśli ten przydział się powiedzie, to temu zasobowi nadawana jest wartość zmiennej `number`, która następnie ulega zwiększeniu o jeden w tej zmiennej (wiersz nr 49). W wierszu nr 50 zapamiętywany jest we wskaźniku `old_pointer` adres kopii zasobu, która bieżąco jest dostępna dla czytelników. Jest on kopiowany ze wskaźnika głównego. Proszę zauważyć, że w przeciwieństwie do czytelników pisarz nie musi uzyskiwać tego adresu w szczególny sposób, po prostu kopiuje go przy pomocy instrukcji przypisania. W wierszu nr 51 pisarz publikuje adres nowej kopii zasobu przepisując go ze wskaźnika `local_number_pointer` do wskaźnika `number_pointer` (głównego wskaźnika zasobu współdzielonego) przy pomocy wywołania funkcji `rcu_assign_pointer()`. Następnie pisarz wywołuje funkcję `synchronize_rcu()` celem zaczekania, aż czytelnicy przestaną korzystać ze poprzedniej kopii współdzielonego zasobu. Jeśli tak się stanie, to zwolni on pamięć na ten zasób przy pomocy wskaźnika `old_pointer` (wiersz nr 54), wcześniej upewniając się, że nie był on pusty (wiersz nr 53). Pozostały kod funkcji i reszty modułu jest podobny do tych prezentowanych wcześniej w tej instrukcji.

Proszę czytając kody źródłowe modułów zwrócić szczególną uwagę na kolejność uruchamiania i kończenia działania poszczególnych wątków! Jest ona bardzo ważna. Jeśli praca wątków będzie kończona w niewłaściwej kolejności, to może dojść do zawieszenia działania jądra systemu.

Zdania

1. `[2 punktów]` Zademonstruj działanie niepodzielnych operacji na bitach.
2. `[4 punkty]` Rozwiąż problem producenta i konsumenta przy pomocy listy i zmiennej sygnałowej. Zauważ, że jeśli jest używana lista, to praca producenta nie musi być wstrzymywana.
3. `[6 punkty]` Zmodyfikuj moduł z listingu 6 tak, aby korzystał on z funkcji `call_rcu()` do niszczenia poprzedniej wersji zasobu współdzielonego.
4. `[2 punktów]` Zmień kod źródłowy modułu z listingu 1 tak, aby korzystał on z funkcji `kthread_create()` zamiast `kthread_run()`.
5. `[4 punktów]` Rozwiąż problem producenta i konsumenta przy pomocy kolejki FIFO, operacji niepodzielnych na zmiennej typu `atomic_t` i muteksa.
6. `[6 punktów]` Rozwiąż problem producenta i konsumenta przy pomocy dziesięcioelementowej tablicy, muteksa i zmiennych sygnałowych.
7. `[2 punktów]` Zmień moduł z listingu 4 tak, aby wątek czytelnika korzystał, zamiast z funkcji `wait_for_completion()`, z funkcji `wait_for_completion_timeout()` i umieszczał w buforze jądra informację ile to oczekiwanie zajmowało taktów zegara.
8. `[4 punktów]` Zmień moduł z listingu 4 tak, aby wątek czytelnika sygnalizował wątkowi pisarza, że zakończył już odczyt zmiennej.
9. `[6 punktów]` Zmodyfikuj kod modułu z listingu 6 tak, aby uruchamiane w nim były dwa wątki pisarzy.