Wykład ósmy

Środki synchronizacji w Linuksie

1. *Operacje niepodzielne na liczbach całkowitych i na bitach*

Operacje niepodzielne (*atomowe*) na zmiennych prostych typów są zazwyczaj realizowane za pomocą instrukcji maszynowych właściwych dla architektury procesora. Listy rozkazów niektórych procesorów zawierają rozkazy, które pozwalają wprost wykonać niepodzielnie takie instrukcje jak np.: dodawanie, odejmowanie, odczyt, zapis, inne dostarczają rozkazów ułatwiających implementację takich operacji w systemach wieloprocesorowych (np. rozkaz blokowania magistrali), jeszcze inne nie posiadają żadnych rozkazów tego typu (np. SPARC). Jądro systemu Linux udostępnia szereg interfejsów (funkcji i makrodefinicji) korzystających z tych rozkazów lub pozwalających w inny sposób zrealizować operacje niepodzielne na typach prostych. Twórcy jądra wyróżnili specjalny typ *atomic_t*, który stosowany jest w miejsce zwykłego typu „int". Pozwala to na zdefiniowanie funkcji, które pracują tylko na takim typie, ukrycie szczegółów implementacji oraz zabezpieczenie przed błędną optymalizacją na poziomie kompilacji. Typ *atomic_t* jest 32-bitowy i bazuje na typie „int". We wczesnych wersjach jądra serii 2.6 pozwalał on na przechowywanie wartości jedynie 24-bitowych (3 bajty). Młodsze 8 bitów (1 bajt) zajmowane było przez blokadę, która była konieczna w przypadku takich typów procesorów, jak wspomniany wyżej SPARC. W późniejszych wersjach jądra znaleziono sposób na zlikwidowanie tej niedogodności. Operacje realizujące działania niepodzielne na liczbach całkowitych są implementowane jako makrodefinicje lub funkcje „inline". Oto te z tych podprogramów, które dostępne są na wszystkich platformach sprzętowych:

- ATOMIC_INIT(i) – pozwala na inicjację zmiennej typu *atomic_t* w miejscu deklaracji,
- int atomic_read(const atomic_t *v) – niepodzielny odczyt całkowitej wartości zmiennej wskazywanej przez „v",
- void atomic_set(atomic_t *v, int i) – niepodzielne przypisanie wartości zmiennej „i" do zmiennej wskazywanej przez „v",
- void atomic_add(int i, atomic_t *v) – niepodzielne dodanie wartości zmiennej „i" do wartości zmiennej wskazywanej przez „v",
- void atomic_sub(int i, atomic_t *v) – niepodzielne odejmowanie wartości zmiennej „i" od zmiennej wskazywanej przez „v",
- void atomic_inc(atomic_t *v) – niepodzielna inkrementacja zmiennej wskazywanej przez „v",
- void atomic_dec(atomic_t *v) – niepodzielna dekrementacja zmiennej wskazywanej przez „v",
- int atomic_sub_and_test(int i, atomic_t *v) – niepodzielne odjęcie wartości zmiennej „i" od wartości zmiennej wskazywanej przez „v" i zwrócenie wartości „true", jeśli różnica wynosi „0",
- int atomic_add_negative(int i, atomic_t *v) – niepodzielne dodanie wartości zmiennej „i" do zmiennej wskazywanej przez „v" i zwrócenie wartości „true", jeśli suma jest ujemna,
- int atomic_dec_and_test(atomic_t *v) – niepodzielna dekrementacja zmiennej wskazywanej przez „v" i zwrócenie wartości „true" jeśli po tej operacji zmienna wskazywana przez „v" będzie miała wartość „0",
- int atomic_inc_and_test(atomic_t *v) – niepodzielna inkrementacja zmiennej wskazywanej przez „v" i zwrócenie wartości „true" jeśli po tej operacji zmienna wskazywana przez „v" będzie miała wartość „0".

Ze względu na rosnącą popularność platform 64-bitowych na pewnym etapie rozwoju serii 2.6 jądra dodano typ *atomic64_t*. Jest to 64-bitowy odpowiednik typu *atomic_t*. Funkcje i makrodefinicje, które go obsługują mają takie same nazwy jak te opisane wyżej, ale rozpoczynające się przedrostkiem *atomic64_* lub *ATOMIC64_*.

Oprócz operacji niepodzielnych na liczbach całkowitych jądro dostarcza funkcji i makrodefinicji realizujących niepodzielne operacje na pojedynczych bitach. Nie działają one na żadnym konkretnym typie, tylko pobierają przez parametry adres miejsca w pamięci i numer bitu:

- void set_bit(int nr, volatile void *addr) – niepodzielne ustawienie bitu na pozycji „nr", zmiennej wskazywanej przez „addr";
- void clear_bit(int nr, volatile void *addr) – niepodzielne wyzerowanie bitu na pozycji „nr", zmiennej wskazywanej przez „addr";
- int test_and_set_bit(int nr, volatile void *addr) – niepodzielne ustawienie bitu na pozycji „nr" zmiennej wskazywanej przez „addr", wraz ze zwróceniem jego poprzedniej wartości;
- int test_and_clear_bit(int nr, volatile void *addr) – niepodzielne wyzerowanie bitu na pozycji „nr", zmiennej wskazywanej przez „addr", wraz ze zwróceniem jego poprzedniej wartości;
- int test_and_change_bit(int nr, volatile void *addr) – niepodzielna negacja bitu na pozycji „nr", zmiennej wskazywanej przez „addr", wraz ze zwróceniem jego poprzedniej wartości;
- test_bit(nr, addr) – niepodzielne testowanie bitu na pozycji „nr", zmiennej wskazywanej przez „addr".

Istnieją także odpowiedniki tych funkcji realizujące takie same operacje, ale nie w sposób niepodzielny. Nazwy tych funkcji są takie same, z tym, że rozpoczynają się znakami „__" (podwójnego podkreślenia). Jądro dostarcza również funkcji pozwalających na znalezienie pierwszego ustawionego bitu w określonym obszarze pamięci i pierwszego bitu o wartości zero w określonym obszarze pamięci: *find_first_bit()* i *find_first_zero_bit()*. Jeśli chcemy przeszukać tylko jedno słowo możemy skorzystać z szybszych wywołań *__ffs()* i *__ffz()*.

2. *Rygle pętlowe*

Najczęściej stosowanym rodzajem blokad są rygle pętlowe (*ang. spin lock*), czyli zmienne typu strukturalnego *spinlock_t*, które chronią „większe" zasoby (takie jak np.

listy zadań) niż zmienne typów całkowitych lub poszczególne bity słów w pamięci operacyjnej. Dany rygiel może przetrzymywać tylko jeden wątek wykonania (zasada wzajemnego wykluczania), natomiast inne wątki chcące skorzystać z chronionego zasobu wykonują *aktywne oczekiwanie*, czyli w pętli oczekują, aż rygiel zostanie zwolniony i *jeden* z nich będzie mógł uzyskać dostęp do zasobu. Ponieważ aktywne oczekiwanie jest marnotrawieniem czasu procesora rygle pętlowe powinny być stosowane wszędzie tam, gdzie nie można zawiesić wątku i gdzie czas przełączania kontekstu byłby wielokrotnie dłuższy od czasu aktywnego oczekiwania. Rygle pętlowe mogą być używane w procedurach obsługi przerwań, ale tylko wraz z wyłączeniem lokalnego systemu przerwań, aby uniknąć zakleszczeń. Należy również pamiętać, że rygle nie są rekurencyjne i nie są stosowane w systemach jednoprocesorowych (kompilator wstawia w ich miejsce puste instrukcje lub, jeśli podczas kompilacji włączona jest opcja wywłaszczania wątków jądra, zastępuje je funkcjami włączającymi i wyłączającymi ich wywłaszczanie, opisanymi dalej). Jądro udostępnia szereg funkcji i makrodefinicji związanych z obsługą rygli pętlowych:

- spin_lock() - zakłada zadany rygiel,
- spin_lock_irq() - wyłącza lokalne przerwania i zakłada rygiel,
- spin_lock_irqsave() - zachowuje stan lokalnych przerwań, wyłącza je i zakłada rygiel,
- spin_lock_bh() - zakłada rygiel i wyłącza dolne połówki (oprócz kolejek prac),
- spin_unlock() - zwalnia rygiel,
- spin_unlock_irq() - włącza przerwania i zwalnia rygiel,
- spin_unlock_irqrestore() - przywraca stan przerwań i zwalnia rygiel,
- spin_unlock_bh() - zwalnia rygiel i włącza dolne połówki (oprócz kolejek prac),
- spin_trylock() - próbuje założyć rygiel, jeśli operacja ta się nie powiedzie zwraca zero i nie powoduje wejścia w pętlę aktywnego oczekiwania,
- spin_is_locked() - zwraca wartość różną od zera, jeśli rygiel jest już przetrzymywany,
- spin_lock_init() - inicjuje rygiel.

Jeśli problem, który chcemy rozwiązać sprowadza się do problemu pisarzy i czytelników, a ściślej do wersji tego problemu, gdzie faworyzowani są czytelnicy, to możemy zastosować *rygle pętlowe R-W*. Rygiel dla czytelników może być przetrzymywany przez większą liczbę wątków wykonania tego typu (a nawet może być rekurencyjnie zakładany przez jeden wątek wykonania tego typu), rygiel dla pisarzy – tylko przez jeden wątek wykonania tego typu. Ponadto pisarz może zakładać rygiel tylko wtedy, gdy z zasobu nie korzysta żaden z czytelników. Również w przypadku rygli R-W istnieje w jądrze szereg funkcji i makrodefinicji pozwalających na manipulację nimi:

- read_lock() - zakłada rygiel R (dla czytelnika),
- read_lock_irq() - zakłada rygiel R (dla czytelnika) i wyłącza lokalne przerwania,
- read_lock_irqsave() - zakłada rygiel R, zapamiętuje stan lokalnych przerwań i wyłącza je,
- read_lock_bh() - zakłada rygiel R (dla czytelnika) i wyłącza dolne połówki (oprócz kolejek prac),
- read_unlock() - zdejmuje rygiel R,
- read_unlock_irq() - zdejmuje rygiel R i włącza lokalne przerwania,
- read_unlock_irqrestore() - zdejmuje rygiel R i przywraca stan lokalnych przerwań,
- read_unlock_bh() - zdejmuje rygiel R (dla czytelnika) i włącza dolne połówki (oprócz kolejek prac),
- write_lock() - zakłada rygiel W (dla pisarza),
- write_lock_irq() - zakłada rygiel W (dla pisarza) i wyłącza lokalne przerwania,
- write_lock_irqsave() - zakłada rygiel W, zapamiętuje stan lokalnych przerwań i wyłącza je,
- write_lock_bh() - zakłada rygiel W (dla pisarza) i wyłącza dolne połówki (oprócz kolejek prac),
- write_unlock() - zdejmuje rygiel W,
- write_unlock_irq() - zdejmuje rygiel W i włącza lokalne przerwania,
- write_unlock_irqrestore() - zdejmuje rygiel W i przywraca stan lokalnych przerwań,
- write_unlock_bh() - zdejmuje rygiel W i włącza dolne połówki (oprócz kolejek prac),
- write_trylock() - próbuje założyć rygiel W, w przypadku niepowodzenia, zwraca wartość równą zero, ale nie powoduje rozpoczęcia pętli aktywnego

oczekiwania,

- rw_lock_init() - inicjuje rygiel pętlowy RW (strukturę typu *rwlock_t*),
- rw_is_locked() - jeśli rygiel jest przetrzymywany, to zwraca wartość różną od zera.

3. *Semafory*

Semafory w przeciwieństwie do rygli pętlowych powodują wprowadzenie wątków wykonania oczekujących na ich podniesienie w stan uśpienia. Takie wątki wstawiane są do kolejki zadań oczekujących (*ang. waitqueue*) związanej z danym semaforem. Semafory nie mogą być przetrzymywane przez wątki, które już przetrzymują rygle pętlowe. Powyższe cech powodują, że semafory są używane tylko w kontekście procesu, kiedy czas oczekiwania na ich podniesienie jest dużo dłuższy niż czas przełączania kontekstu. Semafory nie powodują, tak jak rygle pętlowe wyłączenia wywłaszczania wątków jądra, ponadto umożliwiają przetrzymywanie blokady przez dowolną, z góry określoną przez *licznik semafora* liczbę wątków. Oto funkcje i makrodefinicje związane z obsługą semaforów:

- sema_init(sturct semaphore *, int) – inicjuje strukturę semafora podaną wartością,
- down_interruptible(struct semaphore *) - próbuje opuścić semafor, jeśli to się nie udaje to wprowadza wątek wykonania w stan TASK_INTERRUPTIBLE,
- down(struct semaphore *) - podobnie jak wyżej, ale wątek wykonania jest wprowadzany w stan TASK_UNINTERRUPTIBLE,
- down_killable(struct semaphore *) - dostępna od wersji 2.6.26 jądra, działa jak *down_interruptible*(), ale wprowadza wątek w stan TASK_KILLABLE,
- down_timeout(struct semaphore *, long) - dostępna od wersji 2.6.26 pozwala ograniczyć czas oczekiwania na podniesienie semafora,
- down_trylock(struct semaphore *) - próbuje opuścić semafor, jeśli nie jest to możliwe zwraca wartość niezerową i nie zawiesza wątku,
- up(struct semaphore *) - podnosi semafor i budzi jeden z wątków wykonania czekających na jego podniesienie.

Semafory reprezentowane są przez struktury o typie *struct semaphore*. Można je tworzyć i inicjować za pomocą DEFINE_SEMAPHORE. Podobnie jak w przypadku rygli istnieją semafory R-W. Tworzy się je i inicjuje przy pomocy DECLARE_RWSEM, a samej ich inicjacji można dokonać przy pomocy *init_rwsem()*. Funkcje obsługi działają podobnie jak w przypadku zwykłych semaforów, ale są podzielone na przeznaczone dla czytelników (nazwy zakończone słowem *read*) i pisarzy (*write*). Należy pamiętać, że funkcje *down_read_trylock*() i *down_write_trylock*() zwracają wartości o znaczeniu odwrotnym niż funkcja *down_trylock*(). Ponadto dla tych semaforów istnieje unikatowa funkcja *downgrade_writer()* pozwalająca przekształcić pisarza do roli czytelnika.

4. *Muteksy*

Osoby z grona programistów jądra systemu Linux, które są odpowiedzialne za mechanizmy synchronizacji zauważyły, że najczęściej używanym rodzajem semaforów są semafory binarne, które pełnią rolę muteksów, dlatego w wersji jądra 2.6.16 została wprowadzona poprawka, która definiuje osobny typ danych o nazwie *mutex*. Jest on określony strukturą, którą można zapisać następująco (bez uwzględnienia pól związanych z debugowaniem):

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

W przeciwieństwie do struktury *semaphore*, zawartość tej struktury jest niezależna od architektury procesora. Tę cechę posiadają również w dużej mierze implementacje operacji wykonywanych na niej – mogą być co najwyżej optymalizowane pod względem czasu wykonania na poszczególnych platformach. Pole *count* przechowuje stan muteksu. Jeśli jego wartość jest równa jeden to muteks jest wolny, zero – zajęty, mniejsza od zera – zajęty i na jego podniesienie czeka co najmniej jeden wątek. Rozróżnienie dwóch stanów zajętości pozwala określić, czy konieczne jest budzenie wątków. API muteksów jest dostępne po włączeniu pliku nagłówkowego *linux/mutex.h* i obejmuje następujące funkcje i makrodefinicje:

- DEFINE_MUTEX(name) – definiowanie i inicjacja muteksu na etapie kompilacji,
- mutex_init(struct mutex *lock) – inicjalizacja muteksu na etapie wykonania,
- mutex_lock_interruptible(struct mutex *lock) – zajęcie muteksu, jeśli operacja się nie powiedzie to wątek jest ustawiany w stan TASK_INTERRUPTIBLE,
- mutex_lock(struct mutex *lock) – jak wyżej, ale wątek jest ustawiany w stan TASK_UNINTERRUPTIBLE,
- mutex_lock_killable(struct mutex *lock) - jak wyżej, ale wątek jest ustawiany w stan TASK_KILLABLE,
- mutex_trylock(struct mutex *lock) – zwraca zero jeśli nie uda się zająć muteksu, jeden w przeciwnym przypadku,
- mutex_unlock(struct mutex *lock) – zwalnia mutex,
- mutex_is_locked(struct mutex *lock) – zwraca wartość większą od zera jeśli mutex jest zajęty, zero jeśli jest dostępny.

Muteksy nie są blokadami rekurencyjnymi i nie mogą być używane w kontekście przerwania. Do jądra wprowadzono także muteksy czasu rzeczywistego. Ich działanie jest takie samo jak zwykłych muteksów, ale wątek, który zajmie taki mutex zyskuje priorytet czasu rzeczywistego. Ma to na celu zapobieżenie wystąpieniu zjawiska, które nazywa się „inwersją priorytetów".

5. *Zmienne sygnałowe* (*ang. completion*)

Zmienne sygnałowe służą do synchronizacji pracy wątków i są uproszczoną wersją semaforów. Ich typ jest określony strukturą *struct completion*. Najczęściej są one tworzone jako dynamiczne składowe większych struktur danych. Mogą być także definiowane i inicjowane podczas kompilacji przy pomocy makrodefinicji *DECLARE_COMPLETION()*. Z obsługą tych zmiennych związane są między innymi następujące funkcje:

- init_completion(struct completion *) - inicjuje zmienną sygnałową,
- wait_for_completion(struct completion *) - oczekuje na sygnał ze zmiennej sygnałowej,
- complete(struct completion *) - budzi wątki oczekujące na sygnał.

5. *Blokada BKL (ang. Big Kernel Lock)*

Blokada ta była pomocna w przejściu z wersji jądra 2.0 do 2.2. W jądrze 2.0 implementacja wieloprocesorowości SMP pozwalała na przebywanie w trybie jądra tylko jednemu procesorowi, w 2.2 możliwe było współbieżne wykonywanie kodu jądra na wielu procesorach jednocześnie. BKL miała być rozwiązaniem przejściowym, które miało być zastąpione blokadami o mniejszej ziarnistości. Niestety, przez bardzo długi czas, aż do wydania wersji 2.6.39, wiele mechanizmów jądra korzystało z tego rozwiązania. Blokada ta, wprowadza wątek, który stara się ją zająć w aktywne oczekiwanie, jeśli jest już zajęta przez inny wątek. Wątek przetrzymujący ją może przejść w stan oczekiwania. Wówczas ta blokada jest odblokowywana i automatycznie przywracana, kiedy ten wątek wróci do stanu wykonania. Ponadto BKL jest rekurencyjna, wyłącza wywłaszczanie jądra i można wykorzystywać ją jedynie w kontekście procesu. Do jej obsługi stosowane były następujące funkcje:

- lock_kernel() - zakłada blokadę BKL,
- unlock_kernel() - zdejmuje blokadę BKL,
- kernel_locked() - sprawdza, czy blokada BKL jest już założona.

6. *Blokady sekwencyjne (ang. seq locks)*

Blokady sekwencyjne (zmienne typu *seqlock_t*) korzystają z licznika sekwencyjnego, który jest zwiększany o jeden kiedy blokada jest zakładana i zdejmowana, a dzieje się to wówczas, gdy chroniony zasób jest zapisywany. Blokady sekwencyjne w prosty sposób pozwalają określić, czy operacja odczytu nie została przepleciona z operacją zapisu. Przed odczytem jest zapamiętywana wartość licznika. Po wykonaniu tej operacji odczytywany jest ponownie licznik i jego wartość porównywana jest z wartością poprzednią. Jeśli te wartości są takie same, to można być pewnym, że odczyt nie został przepleciony przez zapis. Do zakładania blokady sekwencyjnej służy funkcja *wirte_seqlock()*, a do zdejmowania *write_sequnlock()*. Do odczytu wartości początkowej *read_seqbegin()*, a do odczytu wartości końcowej *read_seqretry()*. Obie te funkcje są wywoływane w pętli „do" ... „while" . Blokady te są używane do rozwiązywania problemów typu pisarze i czytelnicy, gdzie faworyzowani są pisarze. Jeśli o dostęp do zasobu ubiega się co najmniej dwóch pisarzy, to blokada sekwencyjna działa dla nich jak rygiel pętlowy. Wartość początkowa tej zmiennej (po jej utworzeniu) wynosi zero.

7. *Blokowanie wywłaszczania*

W przypadkach, gdy trzeba chronić dane, które są lokalne dla danego procesora (tj. nie są widziane przez inne procesory) przed dostępem współbieżnym można zrezygnować z rygli pętlowych i zastosować zwykłe zablokowanie wywłaszczania. Do tego służą funkcje *preempt_disable()* i *preempt_enable()*, których wywołania można wielokrotnie zagnieżdżać. Wartość licznika wywłaszczeń, który jest zwiększany o jeden za każdym wywołaniem pierwszej z tych funkcji i zmniejszany o jeden z każdym wywołaniem drugiej, jest zwracana przez *preempt_count()*. Licznik ten w przypadku większości platform sprzętowych (poza między innymi x86) jest polem o nazwie *preempt_count* umieszczonym w strukturze *thread_info*, a więc właściwym każdemu z wątków wykonania z osobna. Jest także dostępna funkcja *preempt_enable_no_resched()*, która włącza wywłaszczanie wątków jądra, ale nie powoduje przeszeregowania zadań. Zablokowania wywłaszczania wątków jądra w systemach wieloprocesorowych można też dokonać przy pomocy *get_cpu()* i *put_cpu()*. Pierwsza z tych funkcji zwraca numer procesora, na którym jest wywoływana.

8. *Blokowanie dolnych połówek*

Aby zablokować i odblokować dolne połówki zrealizowane w postaci przerwań programowych i taskletów należy użyć odpowiednio: funkcji *local_bh_disable()* i *local_bh_enable()*.

9. *Bariery*

Bariery są konstrukcjami, które powstrzymują kompilator i procesor przed zmianą kolejności operacji odczytu i zapisu w kodzie. Wykaz barier pamięciowych i kompilatora jest następujący:

- rmb() - zapobiega zmianie kolejności odczytów wokół niej,
- read_barrier_depends() - zapobiega zmianie kolejności odczytów zależnych wokół niej,
- wmb() - zapobiega zmianie kolejności zapisów wokół niej,
- mb() - zapobiega zmianie porządku odczytów i zapisów wokół niej,
- smp_rmb() - w systemach SMP działa jak rmb(), w jednoprocesorowych jak barrier(),
- smp_read_barrier_depends() - w systemach SMP działa jak read_barrier_depends(), w jednoprocesorowych działa jak barrier(),
- smp_wmb() - w systemach SMP działa jak wmb(), w jednoprocesorowych jak barrier(),
- smp_mb() - w systemach SMP działa jak mb(), w jednoprocesorowych jak barrier(),
- barrier() - zapobiega optymalizacji kodu wokół niej na etapie kompilacji.

10. *Mechanizm RCU*

Mechanizm RCU (skrót od Read-Copy-Update) jest efektywnym, zapewniającym skalowalność środkiem synchronizacji, który pozwala rozwiązać problemy

synchronizacji typu pisarze i czytelnicy. Wymaga on pewnego, najczęściej niewielkiego narzutu pamięci i wymusza spełnienie trzech warunków, aby poprawnie funkcjonował:

- kod korzystający z zasobu chronionego tym mechanizmem nie może ulec uśpieniu,
- zapisy chronionego zasobu powinny być sporadyczne, a odczyty częste,
- chroniony zasób musi być dostępny dla wątków za pomocą wskaźnika.

Zasada działania tego mechanizmu jest stosunkowo prosta. Jeśli wątek-czytelnik chce odczytać zasób, to musi wcześniej pozyskać do niego wskaźnik i przeprowadza operację czytania za pomocą tego wskaźnika. Wątek-pisarz jeśli chce zapisać zasób, to najpierw tworzy jego kopię, modyfikuje ją, a następnie upublicznia wskaźnik na tę kopię. Po tej ostatniej operacji, jeśli któryś z czytelników spróbuje pozyskać wskaźnik do zasobu, to otrzyma w nim adres nowej kopii. Oryginał jest niszczony dopiero po zakończeniu operacji przez wszystkich czytelników, którzy otrzymali do niego wskaźnik przed upublicznieniem przez pisarza nowej wersji tego zasobu. Pisarz posługuje się funkcją *rcu_assign_ptr*(), aby opublikować wskaźnik do nowej, zmodyfikowanej kopii zasobu. Czytelnik korzysta z trzech podprogramów: *rcu_read_lock*(), *rcu_dereference*() - aby uzyskać adres zasobu, *rcu_read_unlock*() - aby zwolnić wskaźnik. Wskaźnikiem uzyskanym przez *rcu_dereference*() może posługiwać się on w kodzie umieszczonym między wywołaniami *rcu_read_lock*() i *rcu_read_unlock*(). Właściciel zasobu (wątek-pisarz), który został zmodyfikowany może go usunąć po powrocie z funkcji *synchronize_rcu*(). Może on również wykorzystać funkcję *call_rcu*(), aby zarejestrować funkcję, która zostanie wykonana, jeśli wszyscy czytelnicy korzystający z zasobu zakończą na nim operację. Należy zaznaczyć, że ten mechanizm nie zapewnia ochrony przed współbieżnym zapisem.