Synchronizacja procesów na poziomie jądra - algorytmy up i down.

Zuzanna Szymańska

grudzień 2000

Spis treści

1 Wstęp

Procesy współbieżne mogą ze sobą konkurować o dostęp do wspólnie użytkowanych zasobów systemowych. Obiekt, z którego w sposób wyłączny może korzystać wiele procesów nazywa się zasobem dzielonym, zaś fragment procesu, w którym wykorzystywany jest zasób dzielony nazywa się sekcją krytyczną procesu. Jądro systemu zarządzające tymi zasobami musi dysponować sprawnym mechanizmem pozwalającym na pełną synchronizację współzawodniczących procesów. Zwykle mamy do czynienia z sytuacją, gdy z danego zasobu może w określonej chwilii korzystać wyłącznie jeden proces, dlatego wykonując swoją sekcję krytyczną uniemożliwia on wykonanie sekcji krytycznych innym procesom.

W systemie operacyjnym Linux mechanizmem zapewniającym wzajemne wykluczanie na poziomie jądra są semafory. Wedle klasycznej definicji, semafor jest nieujemną zmienną całkowitoliczbową z wyróżnionymi atomowymi operacjami podniesienia i opuszczenia. Przed wejściem do sekcji krytycznej proces opuszcza semafor, natomiast wychodząc z niej podnosi go. Proces może przejść pod semaforem jedynie wtedy, gdy wartość semafora jest dodatnia, wtedy zmniejsza ją o jeden i przechodzi do sekcji krytycznej. Natomiast operacja podniesienia semafora polega na zwiększeniu jego wartości również o jeden. Jeżeli proces opuszczający semafor stwierdzi, że jego wartość jest równa zero, zawiesza się. Zawieszony proces czeka, dopóki inny proces nie podniesie semafora i w ten sposób nie zwiększy jego wartości.

Semafory zaimplementowane w jądrze Linuxa różnią się znacząco od klasycznej definicji.

2 Semafory w systemie Linux

Jak już wspomiałam w systemie Linux semafory systemowe (to znaczy dostępne jedynie dla procedur systemowych) są mechanizmem bardziej złożonym. Struktura określająca semafor składa się trzech atrybutów: count, waking i wait.

```
struct semaphore{
  atomic_t count;
  int waking;
  struct wait_queue *wait;
};
```

Atrybut count określa aktualną wartość semafora. Nie jest to zwykła wartość calkowitoliczbowa, ale obiekt typu atomic_t. W odróżnieniu od klasycznej definicji, wartość semafora w tej implementacji może być liczbą ujemną. Zmiennej count nadaje się początkową wartość równą pojemności sekcji krytycznej. Definicja typu atomic_t znajduje się w pliku atomic.h. Zmienna typu atomic_t jest strukturą zawierającą jeden atrybut typu int (przy zdefiniowanym przetwarzaniu równoległym jest to volatile int), na której są w sposób atomowy zdefiniowane niektóre operacje arytmetyczne. Dla naszych rozważań istotna jest funkcja atomic_inc_and_test_greater_zero(volatile atomic_t *v), która w sposób atomowy zwiększa podany parametr o jeden, a następnie sprawdza, czy jego wartość jest większa od zera, oprócz niej również ważne jest definicja atomic_set(v, i), służąca do inicjalizowania wartości semafora.

Wszystkie funkcje z pliku atomic.h są zapisane za pomocą Assemblera, ponieważ język C nie zapewnia wystarczających mechanizmów do właściwego ich zaimplementowania.

Pomocnicza zmienna waking typu int mówi o tym, ile z zawieszonych procesów należy zwolnić. Ten atrybut należy inicjalizować na 0.

Zmienna wait jest wskaźnikiem do kolejki procesów zawieszonych na danym semaforze. Kolejka procesów zawieszonych jest cykliczna. Elementy tych kolejek są zdefiniowane w pliku wait.h w następujący sposób:

```
struct wait_queue{
  struct task_struct *task;
  struct wait_queue *next;
};
```

Atrybut task jest oczywiście wskaźnikiem na strukturę TASK zawieszonego procesu, zaś atrybut next jest wskaźnikiem na następny element w kolejce. W pliku sched.h zdefiniowane są bardzo ważne dla poruszanego przez nas problemu funkcje: add_wait_queue i remove_wait_queue. Pierwsza z nich służy do wstawiania a druga do usuwania procesu z kolejki procesów zawieszonych. Warto zaznaczyć, że wstawianie i usuwanie procesów z omawianych kolejek jest chronione przez wywołania funkcji write_lock_irqsave i write_unlock_irqrestore.

Również jeśli chodzi o możliwe do wykonania operacje, implementacja Linuxowa różni się od definicji klasycznej semafora. Proces może wykonać jedną z czterech operacji:

- down (struct semaphore *sem) - operacja opuszczania semafora, zawieszany proces jest usypiany w taki sposób, że nie może odbierać sygnałów.
- down_interruptible (struct semaphore *sem) - operacja opuszczania semafora, różni się od poprzedniej tym, że zawieszany proces jest usypiany tak, iż może odbierać sygnały.
- down_trylock (struct semaphore *sem) - operacja warunkowego opuszczania semafora, proces rezygnuje z dostępu do zasobu, jeśli musiałby zawiesić się na semaforze.
- up (struct semaphore *sem) - operacja podnoszenia semafor

W kolejnych rozdziałach opisałam bardziej szczegółowo wspomiane operacje.

Procesy wykonywane w trybie jądra mogą być przerywane przez procedury obsługi przerwań. Może to prowadzić do niekorzystnych sytuacji. Wyobraźmy sobie, że proces wykonuje procedurę obsługi przerwania i w tym czasie dociera do niego kolejne przerwanie. W takiej sytuacji proces powinien natychmiast rozpocząć wykonywanie procedury obsługi nowego przerwania. Co stanie się jednak, jeśli proces znowu odbierze kolejne przerwania? Sposbem na uniknięcie takich sytuacji jest blokowanie flagi przerwań procesora w momencie wchodzenia do sekcji krytycznej i ponowne jej odblokowywanie przy wyjściu. Zablokowanie flagi przerwań uniemożliwia wszelkie przerwania sprzętowe za wyjątkiem przerwań niemaskowalnych.

W wielu przytaczanych przeze mnie funkcjach operacje na fladze przerwań służą właśnie do realizacji wzajemnego wykluczania. Zanim proces znajdzie się w obszarze krytycznym zapisuje rejestr flag na zmienną i blokuje przerwania (asemblerowa operacja cli()). Po wyjściu z obszaru krytycznego rejestr flag jest przywracany, a przerwania odblokowywane. Używane w omawianych algorytmach funkcje ochronne to:

- write_lock_irqsave
- write_unlock_irqrestore
- spin_lock_irqsave
- spin_unlock_irqrestore

3 Operacje semaforowe - schematy algorytmów

W tym rozdziale chciałam przedstawić w nieco uproszczony sposób ogólny zarys algorytmów służących do realizacji operacji semaforowych. Dokładniejszy opis implementacji zamieściłam w rozdziale czwartym.

Algorytm opuszczania semafora down. Do bezwarunkowego opuszczania semafora służą dwie funkcje:
down(struct semaphore *sem) oraz down_interruptible(struct semaphore *sem). Pierwsza z wymienionych funkcji usypia proces w sposób uniemożliwiający mu odbieranie sygnałów, gdy tymczasem druga usypia proces tak, że może on przechwytywać sygnały. Opuszczanie semafora rozpoczyna się od zmniejszenia jego wartości, a następnie porównania jej z zerem. Dla uproszczenia możemy w tym momencie założyć, że te operacje wykonują się niepodzielnie. Jeżeli wartość semafora jest większa, bądź równa zero, to proces może przejść do swojej sekcji krytycznej. Jeżeli natomiast wartość semafora jest ujemna proces musi zostać zawieszony. W takiej sytuacji proces sam siebie wstawia do kolejki procesów zawieszonych (wskaźnikiem do tej kolejki jest właśnie atrybut semafora wait), a następnie w pętli sprawdza wartość atrybutu waking. Jeśli wartość waking jest większa od zera, to znaczy, że zasób dzielony został zwolniony i proces może z niego skorzystać. Wtedy proces zmniejsza ponownie wartość waking (chodzi o ewentualne zablokowanie innych procesów) i wychodzi z pętli. Jeśli natomiast okazało się, że wartość waking nie jest dodatnia, zasób dzielony jest ciągle zajęty, to proces musi oddać sterowanie innemu i dlatego wywołuje funkcję szeregującą schedule(). Tak więc, proces wychodzi z pętli, gdy uda mu się zmniejszyć wartość atrybutu waking. Po wyjściu z pętli proces usuwa siebie z kolejki procesów zawieszonych wait i przechodzi do sekcji krytycznej.

Algorytm opuszczania semafora down_interruptible. Jak napisałam powyżej funkcja down_interruptible (struct semaphore *sem) służy do opuszczania semafora, przy czym w przypadku niepowodzenia usypia proces w taki sposób, że może on odbierać sygnały. Algorytm działania opisywanej operacji jest prawie taki sam jak opisanej powyżej. Różnica polega na tym, że proces oprócz sprawdzania w pętli wartości atrybutu waking sprawdza również, czy nie dostał sygnału. Tak więc, proces może wyjść z pętli, gdy uda mu się zmniejszyć wartość atrybutu waking, lub gdy otrzyma sygnał. Jeśli proces przechwycił sygnał, musi z powrotem zwiększyć wartość atrybutu count i jeśli teraz jest on dodatni, to trzeba również zwiększyć atrybut waking. Zmniejszenie atrybutu waking jest bardzo ważne dla zapewnienia wzajemnego wykluczania, inaczej do sekcji krytycznej mogłoby dostać się zbyt wiele procesów. Funkcja down_interruptible zwraca wartość 0, jeśli procesowi udało się przejść pod semaforem i wartość różną od 0, jeśli proces został obudzony na skutek otrzymania sygnału.

Algorytm warunkowego opuszczania semafora down_trylock. Do warunkowego opuszczania semafora służy funkcja down_trylock(struct semaphore *sem). Proces wykonujący tę operację próbuje przejść pod semforem, jednak w przypadku niepowodzenia (to znaczy gdyby musiał się zawiesić) rezygnuje z dostępu do zasobu. Warunkowe opuszczanie semafora rozpoczyna się tak jak poprzednie operacje od zmniejszenia o jeden wartości atrybutu count. Jeżeli wartość count jest nieujemna, to proces przechodzi dalej, jeżeli natomiast count jest mniejszy od zera, to proces sprawdza wartość atrybutu waking. Jeśli wartość waking jest większa od zera, zasób dzielony został zwolniony, to proces zmniejsza ponownie wartość waking, kończy wykonywanie operacji i przechodzi do sekcji krytycznej. Jeśli natomiast okazało się, że wartość waking nie jest dodatnia i proces nie może przejść pod semaforem, to musi z powrotem zwiększyć atrybut count o jeden. Jeśli dodatkowo atrybut count po zwiększeniu okazał sią większy od zera, to proces musi również zmnijeszyć o jeden atrybut waking. Tak jak w poprzednim algorytmie chodzi tu o wzajemne wykluczanie. W sytuacji, gdy procesowi udało się przejść pod semaforem funkcja zwraca wartość 1, a w przeciwnym przypadku wartość 0.

Algorytm podnoszenia semafora up. Na początku wywołana funkcja up(struct semaphore *sem) zwiększa o jeden wartość semafora, czyli atrybut count, a następnie sprwadza jego wartość. Podobnie jak poprzednio dla uproszczenia możemy założyć, że te operacje wykonują się atomowo. Jeżeli wartość count jest mniejsza lub równa zero, to znaczy, że przynajmniej jeden proces czeka zawieszony na tym semaforze. Wówczas proces podnoszący semafor zwiększa o jeden wartość atrybutu waking, i budzi wszystkie procesy. Jeden z obudzonych procesów zmniejsza wartość atrybutu waking i przechodzi pod semaforem, podczas gdy inne ponownie się zawieszają.

4 Operacje semaforowe - zarys implementacji

Opis implementacji operacji down. Jak już wspomiałam, bezwarunkowe opuszczanie semafora zaczyna się od wywołania asemblerowej funkcji down(struct semaphore * sem) zapisanej w pliku semaphore.h. Parametrem wywołania funkcji jest wskaźnik na opuszczany semafor. Tutaj istotne jest użycie właśnie Asemblera, ponieważ umożliwia on skorzystanie z operacji normalnie niedostępnych z poziomu języka C. Dzięki temu operacja zmniejszenia wartości semafora może się odbyć w sposób niepodzielny (chodzi tu między innymi o użycie dla systemów wieloprocesorowych instrukcji lock, która blokuje przepływ danych na szynie). Wykonanie funkcji down(struct semaphore * sem) zaczyna się właśnie od zmniejszenia atrybutu count. Następnie, jeśli atrybut count ma ustawioną flagę znaku wykonywany jest skok do wywołania __down_failed (kod znajduje się w pliku semaphore.S). Warto zwrócić uwagę na to, że ta flaga jest przechowywana w rejestrach procesora, tak więc upraszczając można powiedzieć, że zmniejszenie count i sprawdzenie jego wartości odbywa się w sposób atomowy. Ustawienie flagi znaku oznacza, że wartość semafora jest ujemna i proces musi zostać zawieszony. Jeśli zaś flaga znaku nie była ustawiona, to znaczy, że wartość semafora jest większa, bądź równa zero. Wtedy proces kończy opuszczanie semafora i przechodzi do sekcji krytycznej. Wróćmy jednak do sytuacji, gdy wartość semafora jest ujemna i proces musi się zawiesić. Po zapamiętaniu rejestrów wywoływana jest funkcja __down (struct semaphore *sem) (kod w plikut sched.c). Na początku funkcji znajduje się sie następująca deklaracja zmiennych:

```
struct task_struct * tsk = current;
struct wait_queue wait = {tsk, NULL};
```

W pierwszej deklaracji proces tworzy wskaźnik do struktury TASK procesu i nadaje mu wartość current. Inaczej mówiąc, jest to teraz wskaźnik na jego własną strukturę TASK. Następnie tworzy nowy element kolejki procesów zawieszonych i umieszcza tam uprzednio zadeklarowany wskaźnik do swojej struktury TASK. W dalszej kolejności proces ustawia swój stan na TASK_UNINTERRUPTIBLE i wywołuje funkcję add_wait_queue (&sem->wait, &wait), która służy mu do umieszczania się w kolejce procesów zawieszonych. Definicja tej funkcji znajduje się w pliku sched.h. Warto zwrócić uwagę na dwie rzeczy. Po pierwsze, parametrami wywołania funkcji są adresy w pamięci, a nie wskaźniki, zaś po drugie, sama operacja wstawiania nowego elementu do kolejki jest dodatkowo chroniona (funkcje write_lock_irqsave i write_unlock_irqrestore).

Po umieszczeniu się w kolejce procesów zawieszonych na semaforze proces wykonuje nieskończoną pętlę. Na początku pętli proces wywołuje funkcję waking_non_zero(sem), której uproszczony kod wygląda następująco:

```
int waking_non_zero (struct semaphore *sem){
  spin_lock_irqsave(&semaphore_wake_lock,flags);
  if (sem->waking>0) sem->waking--;
  spin_lock_irq_restore(&semaphore_wake_lock,flags);
};
```

Parametrem wywołania jest wskaźnik do opuszczanego semafora, funkcja sprawdza, jaka jest wartość atrybutu waking i jeśli jest on dodatni, zmniejsza go. Funkcje spin_lock_irqsave i spin_lock_irqrestore służą do ochrony wykonywanej operacji. Jeśli funkcji waking_non_zero(struct semaphore *sem) uda się zmniejszyć atrybut waking, to zwraca 1 i proces wychodzi z pętli. Natomiast w przeciwnym przypadku funkcja zwraca 0, co oznacza, że proces ciągle musi być wstrzymywany i nie może opuścić wykonywanej pętli. Skoro proces ciągle nie może przejść pod semaforem, to powinien oddać sterowanie innemu procesowi. W tym celu wywołuje funkcję szeregującą procesy schedule() (patrz pliki sched.h i sched.c). Gdy sterowanie ponownie powróci do procesu, wykona on następny obrót pętli i tak będzie do momentu aż uda mu się zmniejszyć atrybut waking i wyskoczyć z pętli.

Po wyjściu z pętli proces zmienia swój stan na RUNNING i przy pomocy funkcji remove_wait_queue (&sem->wait, &wait) usuwa się z kolejki procesów zawieszonych. Wywołanie funkcji remove_wait_queue (&sem->wait, &wait) jest analogiczne do wywołania funkcji add_wait_queue (&sem->wait, &wait).

W całości uproszczony pseudokod operacji opuszczania semafora wygląda następująco:

```
void down (struct semaphore *sem){
  if (--sem->count < 0){
    struct task_struct * tsk = current;
    struct wait_queue wait = {tsk, NULL};
```

```
    tsk->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(&sem->wait, &wait);
    for (;;){
      if (waking_non_zero(sem)) break;
      schedule();
      tsk->state=TASK_UNINTERRUPTIBLE;
    };
    tsk->state=RUNNING;
    remove_wait_queue(&sem->wait, &wait);
  };
};
```

Opis implementacji operacji down_interruptible. Opuszczanie semafora przy pomocy funkcji down_interruptible (struct semaphore * sem) jest bardzo podobne do opuszczania semafora przy pomocy funkcji down (struct semaphore * sem). Operacja zaczyna się podobnie jak poprzednio wywołaniem asemblerowej funkcji (również z pliku semaphore.h) down_interruptible (struct semaphore * sem) (nazwy używanych funkcji i wywołań różnią się dodaniem sufiksu interruptible). Jedna z niewielu różnic polega na tym, że down_interruptible zwraca 0, jeśli procesowi udało się przejść pod semaforem i wartość różną od 0, jeśli proces nie mógł przejść pod semaforem, ponieważ otrzymał sygnał, podczas gdy funkcja down (struct semaphore * sem) nie zwraca żadnej wartości. Drugą istoną różnicą jest to, że funkcja __down_interrupible (wywoływana zamiast funkcji __down) ustawia stan zawieszanego procesu na TASK_INTERRUPTIBLE, tak aby mógł on odbierać sygnały. Podobnie jak w porzednio omawianej operacji proces, po wstawieniu się do kolejki procesów zawieszonych zaczyna wykonywać nieskończoną pętlę. Trzecia różnica, to właśnie nieco inny wygląd tej pętli. Przede wszystkim zamiast wywoływać funkcję waking_non_zero(`struct semaphore *sem`) proces wywołuje funkcję waking_non_zero_interruptible(struct semaphore *sem, `struct task_struct *tsk`), której schemat wygląda następująco:

```
int waking_non_zero_interruptible(struct semaphore *sem,
                                  struct task_struct *tsk ){
  int ret = 0;
  spin_lock_irqsave(&semaphore_wake_lock,flags);
  if (sem->waking>0) {
    sem->waking--;
    ret = 1;
  }else
      if (signal_pending(tsk))
        if (atomic_test_and_set_greater_zero(&sem->count)){
          --sem->waking
          ret = EINTR;
        }
  spin_lock_irqrestore(&semaphore_wake_lock,flags);
  return ret;
};
```

Funkcja waking_non_zero_interruptible (struct semaphore *sem, struct task_struct *tsk) zwraca wartość 1, jeśli procesowi udało się zmnijeszyć atrybut waking, 0 jeśli procesowi nie udało się zmniejszyć waking i nie otrzymał sygnału oraz EINTR (interrupted system call) jeśli proces odebrał sygnał. Podobnie jak w funkcji waking_non_zero (struct semaphore *sem) proces zaczyna wykonanie funkcji od wywołania spin_lock_irqsave, a następnie próbuje zmniejszyć atrybut waking. Jeśli ta próba się powiedzie, proces właściwie kończy wykonywanie funkcji. Jeśli zaś procesowi nie udało się zmniejszyć waking, to w dalszej kolejności sprawdza on, czy został wysłany do niego sygnał. Jeśli proces rzeczywiście otrzymał sygnał, i musi przerwać opuszczanie semafora, to zanim to uczyni musi zrobić po sobie porządek. Rozpoczynając opuszczanie semafora proces zmniejszył wartość atrybutu count, dlatego teraz musi wykonać już opisywaną przeze mnie operację atomic_test_and_set_greater_zero(&sem->count). Jeśli okaże się, że wartość semafora jest większa od zera, to należy jeszcze zmniejszyć wartość atrybutu waking. Na końcu wykonywanej funcji proces wywołuje spin_lock_irqrestore.

Po wyjściu z funkcji waking_non_zero_interruptible (struct semaphore *sem, struct task_struct *tsk) proces musi postąpić odpowiednio do zwróconej wartości. Jeśli jest ona równa 0, czyli atrybut waiting pozostał bez zmian, proces dalej zachowuje się tak jak przy wywołaniu __down (struct semaphore *sem). W przeciwnym przypadku proces nie będzie dłużej wstrzymywany na semaforze i dlatego wyskakuje z pętli. Zanim to uczyni musi jeszcze tylko właściwie ustalić zwracaną wartość, to jest w przypadku, gdy funkcja waking_non_zero_interruptible (struct semaphore *sem, struct task_struct *tsk) zwróciła wartość 1 (sekcja krytyczna stoi otworem) wartość zwracana przez funkcję __down_interruptible (struct semaphore *sem) musi być równa zero. Zakończenie wykonywanej operacji jest takie samo zakończenie poprzedniej.

Całość uproszczonego pseudokodu operacji down_interruptible (struct semaphore * sem) wygląda następująco:

```
int down_interruptible (struct semaphore *sem){
  if (--sem->count < 0){
    struct task_struct * tsk = current;
    struct wait_queue wait = {tsk, NULL};
    tsk->state = TASK_INTERRUPTIBLE;
    add_wait_queue(&sem->wait, &wait);
    for (;;){
      ret = (waking_non_zero_interruptible(sem,tsk));
      if(ret){
        if (ret == 1) ret = 0;
        break;
      }
      schedule();
      tsk->state=TASK_INTERRUPTIBLE;
    };
    tsk->state=RUNNING;
    remove_wait_queue(&sem->wait, &wait);
  }else ret = 1;
  return ret;
};
```

Opis implementacji operacji down_trylock. Warunkowe opuszczanie semafora przeprowadza się wywołując funkcję down_trylock (struct semaphore * sem). Również i ta operacja zaczyna się podobnie jak poprzednie od wywołania funkcji down_trylock (struct semaphore * sem) (również plik semaphore.h). Kolejno wykonuje się analogiczne wywołania i funkcje (wołanie __down_trylock_failed i funkcja __down_trylock (struct semaphore * sem)). Różnica pojawia się dopiero wewnątrz funkcji __down_trylock (struct semaphore * sem). Znajduje się w niej bowiem jedynie wywołanie funkcji waking_non_zero_trylock(struct semaphore * sem), która wygląda następująco:

```
int waking_non_zero_trylock(struct semaphore *sem){
  int ret = 0;
  spin_lock_irqsave(&semaphore_wake_lock,flags);ci
  if (sem->waking ≤0)
    if (atomic_test_and_set_greater_zero(&sem->count))
      --sem->waking
  else {
    --sem->waking--;
    ret=0;
  }
  spin_lock_irqrestore(&semaphore_wake_lock,flags);
  return ret;
};
```

Podobnie jak analogiczne funkcje dla innych operacji waking_non_zero_trylock (struct semaphore *sem) jest chroniona przez wywołanie spin_lock_irqsave. Dalej, funkcja sprawdza jaka jest wartość atrybutu waking. Jeżeli waking jest większa od zera proces zmniejsza ją, wywołuje spin_lock_irqrestore, kończy wykonywanie wszystkich funkcji i przechodzi do sekcji krytycznej. Jeżeli waking jest mniejsza, bądź równa zero proces nie może przejść pod semaforem. Musi jednak, podobnie jak przy funkcji waking_non_zero_interruptible (struct semaphore *sem, struct task_struct *tsk) poprawić wartości atrybutów. Próbując opuścić semafor proces zmniejszył wartosć atrybutu count, dlatego teraz musi wywołać atomic_test_and_set_greater_zero(&sem->count). Jeśli po tym wartość semafora jest większa od zera, to jeszcze zmniejsza wartość atrybutu waking. Na końcu wywołuje funcję spin_lock_irqrestore.

```
int down_trylock(struct semaphore *sem){
  if (--sem->count < 0){
    ret = waking_non_zero_trylock(sem);
  }else ret = 1;
  return ret;
};
```

Opis implementacji operacji up. Podnoszenie semafora zaczyna się od wywołania asemblerowej funkcji up (struct semaphore * sem) zapisanej w pliku semaphore.h. Jak zwykle parametrem wywołania funkcji jest wskaźnik na podnoszony semafor. Wykonanie funkcji zaczyna się od atomowego zwiększenia wartości atrybutu count.

Następnie, sprawdzana jest flaga znaku i zera atrybutu count. Jeśli flaga znaku i zera nie jest ustawiona, co oznacza, że nie ma procesów zwieszonych na semaforze, proces kończy wykonywanie operacji. Jeśli zaś flaga jest ustawiona,

(to znaczy, że po zwiększeniu count jest mniejszy lub równy zero) wykonywany jest skok do wywołania __up_wakeup (kod znajduje się w pliku semaphore.S). Dalej proces zapamiętuje potrzebne rejestry i wywołuje funkcję __up (struct semaphore *sem) (kod w plikut sched.c). W tym miejscu znajdują się tylko dwa wywołania innych funkcji. Pierwsza z nich wake_one_more() służy do zwiększania wartości atrybutu waking (jeden z już zawieszonych procesów dowie się w ten spsób, że zasób dzielony został zwolniony).

```
void wake_one_more (struct semaphore *sem){
  spin_lock_irqsave(&semaphore_wake_lock,flags);
  ++sem->waking;
  spin_lock_irq_restore(&semaphore_wake_lock,flags);
};
```

Wykonywana po niej funkcja wake_up (&sem->wait) służy do budzenia wszystkich zawieszonych procesów. Jednemu z obudzonych procesów uda się zmniejszyć wartość waking i przejść pod semaforem. Inne natomiast ponownie się zawieszą.

```
void up (struct semaphore *sem){
  if (++sem->count>0)
     wake_one_more (struct semaphore *sem);
    wake_up(&sem->wait);
  };
};
```

5 Zadania

Zadanie 1 Podaj przykład błędnego scenariusza, gdyby w funkcji waking_non_zero_interruptible (sem, tsk) zamiast kodu:

```
if (signal_pending(tsk))
  if (atomic_test_and_set_greater_zero(&sem->count)){
    --sem->waking
    ret = EINTR;
  }
```

znajdował się następujący kod:

```
if (signal_pending(tsk))
  if (atomic_test_and_set_greater_zero(&sem->count)){
    ret = EINTR;
  }
```

Zadanie 2 Podaj scenariusz błędu, jaki mógłby pojawić się, gdyby proces wykonujący up, po zwiększeniu atrybutu count porównywał go z zerem nie za pomocą flagi znaku, ale poprzez sięganie do pamięci.

Rozwiązanie zadania 1 Załóżmy, że w systemie cztery procesy: P1, P2, P3, P4 współzawodniczą o dostęp do zasobu. W sekcji krytycznej może jednocześnie przebywać jeden proces. Atrytbut count jest zainicjalizowany na 1, zaś waking na 0. Wymagany scenariusz może być następujący: Proces P1 opuszcza semafor i przechodzi do sekcji krytycznej (po tej operacji count=0, waking=0). Dalej proces P2 wykonuje down_interruptible. W międzyczasie P1 podnosi semafor, P2 otrzymuje sygnał i rezygnuje z dostępu do zasobu, tak więc po tych operacjach count=0, waking=1. W takiej sytuacji dwa kolejne procesy P3 i P4 mogą bez przeszkód dostać się do sekcji krytycznej i spowodować błąd.

Rozwiązanie zadania 2 Załóżmy, że w systemie trzy procesy: P1, P2, P3 współzawodniczą o dostęp do zasobu. W sekcji krytycznej mogą jednocześnie przebywać dwa procesy. Atrytbut count jest zainicjalizowany na 2, zaś waking na 0. Wymagany scenariusz może być następujący: Procesy P1 i P2 znajdują się w sekcji krytycznej, zaś proces P3 wisi na opuszczonym semaforze. Dalej proces P1 zaczyna wykonywać up, zwiększa wartość count, ale w tym momencie sterowanie przejmuje proces P2, który również zaczyna wykonywać operację up. Proces P2 zwiększa atrybut count, porównuje go z zerem, stwierdza, że jest większy od zera i dlatego nie budzi zawieszonegpo procesu. Również P1, gdy ponownie przejmie sterowanie, nie będzie budził procesów i w ten sposób proces P3 zostanie zawieszony.

Bibliografia

[1] Weiss Z., Gruźlewski T., Programowanie współbieżne i rozproszone WNT, 1993 Warszawa ,

[2] Silberschatz A.,Petwrson J.,Galvin P., Podstawy systemów operacyjnych WNT, 1993 Warszawa,

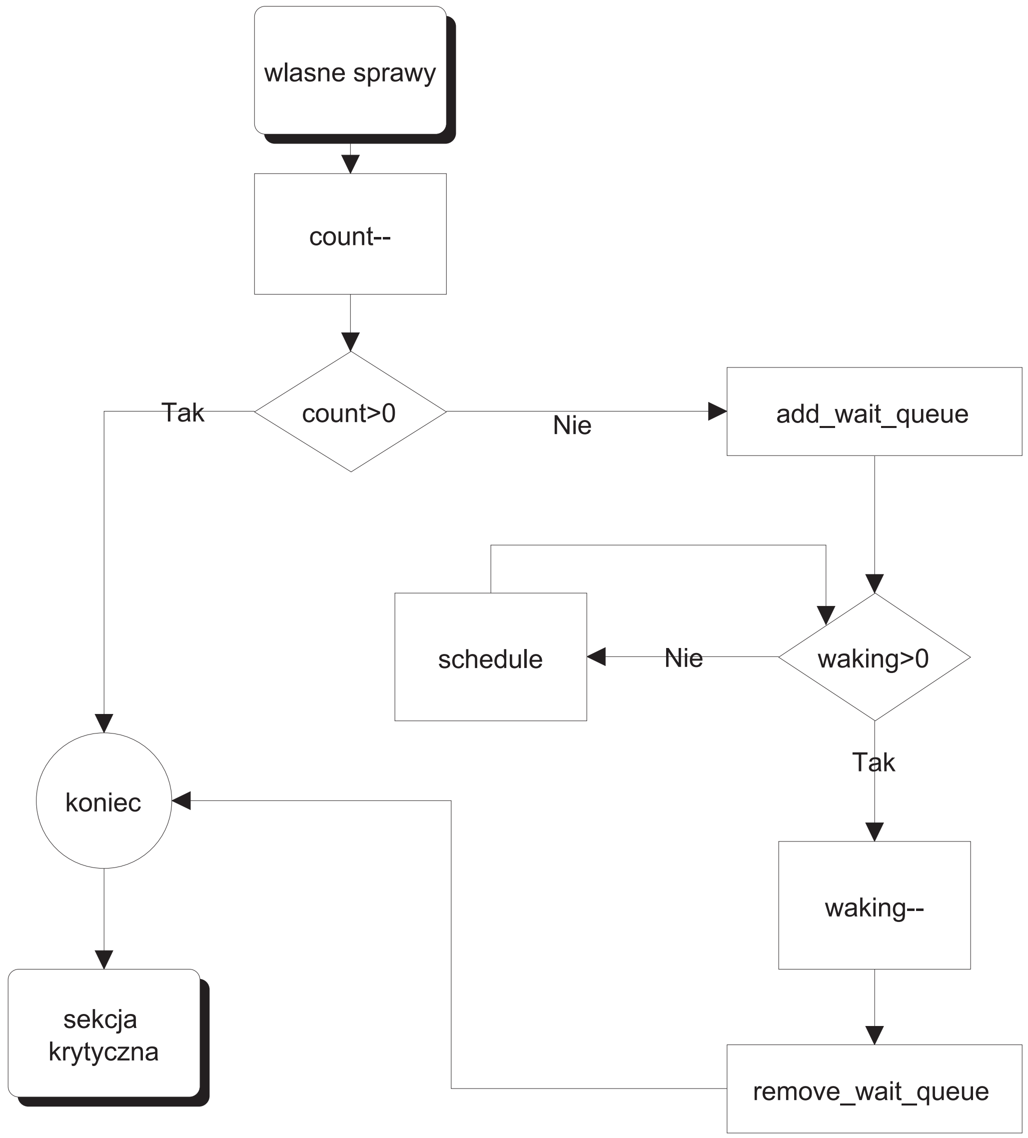
[3] Bach M., Budowa systemu operacyjnego UNIX WNT, 1995 Warszawa

[4] Beck M., Bohme H., Dziadzka M., Kunitz U., Magnus R., Verwarner D. MIKOM, 1999 Warszawa

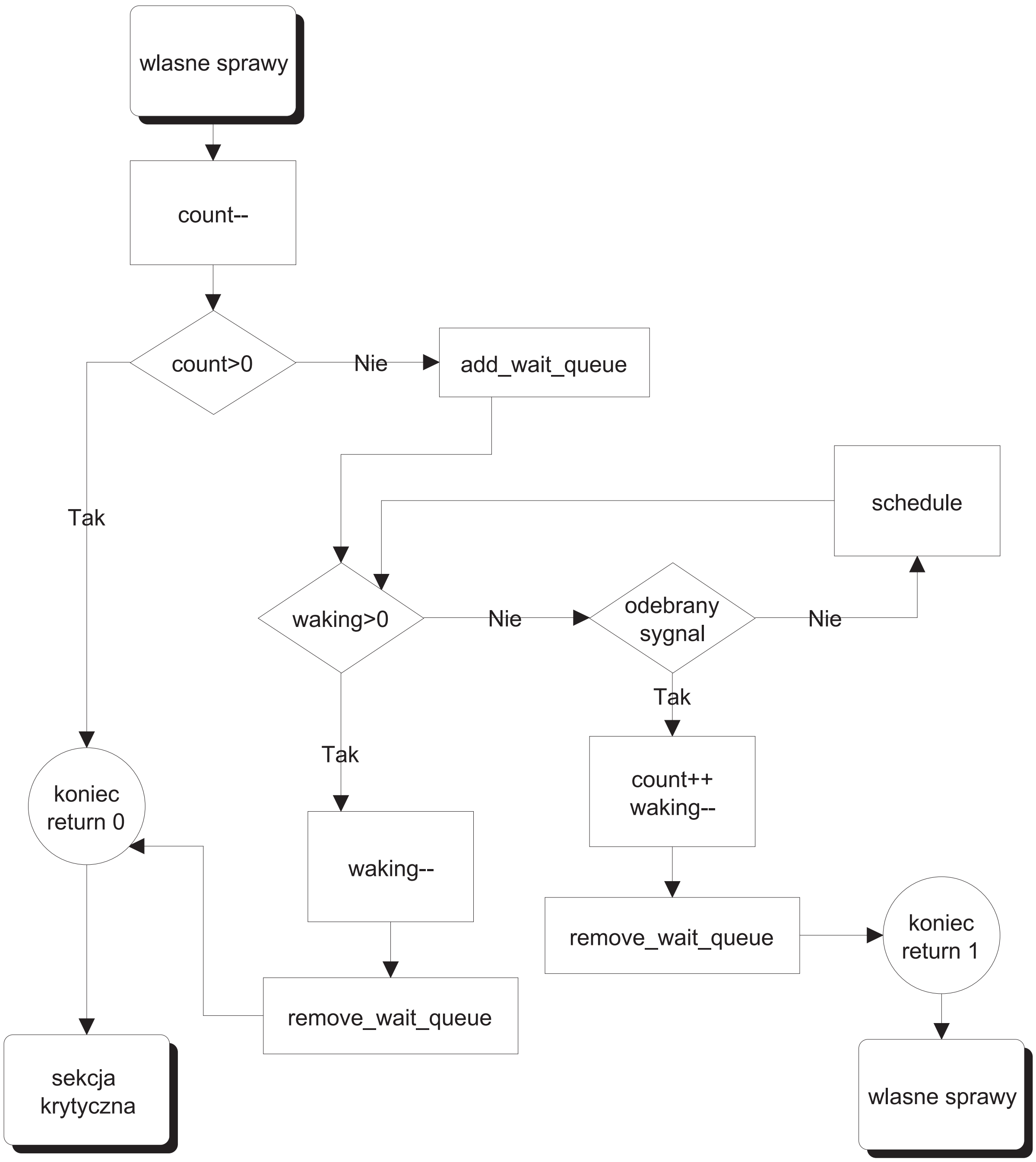
[5] Wróbel E. Asembler 8086/88 WNT, 1990 Warszawa

[6] Bielecki J. Turbo assembler i turbo debugger WNT, 1990 Warszawa

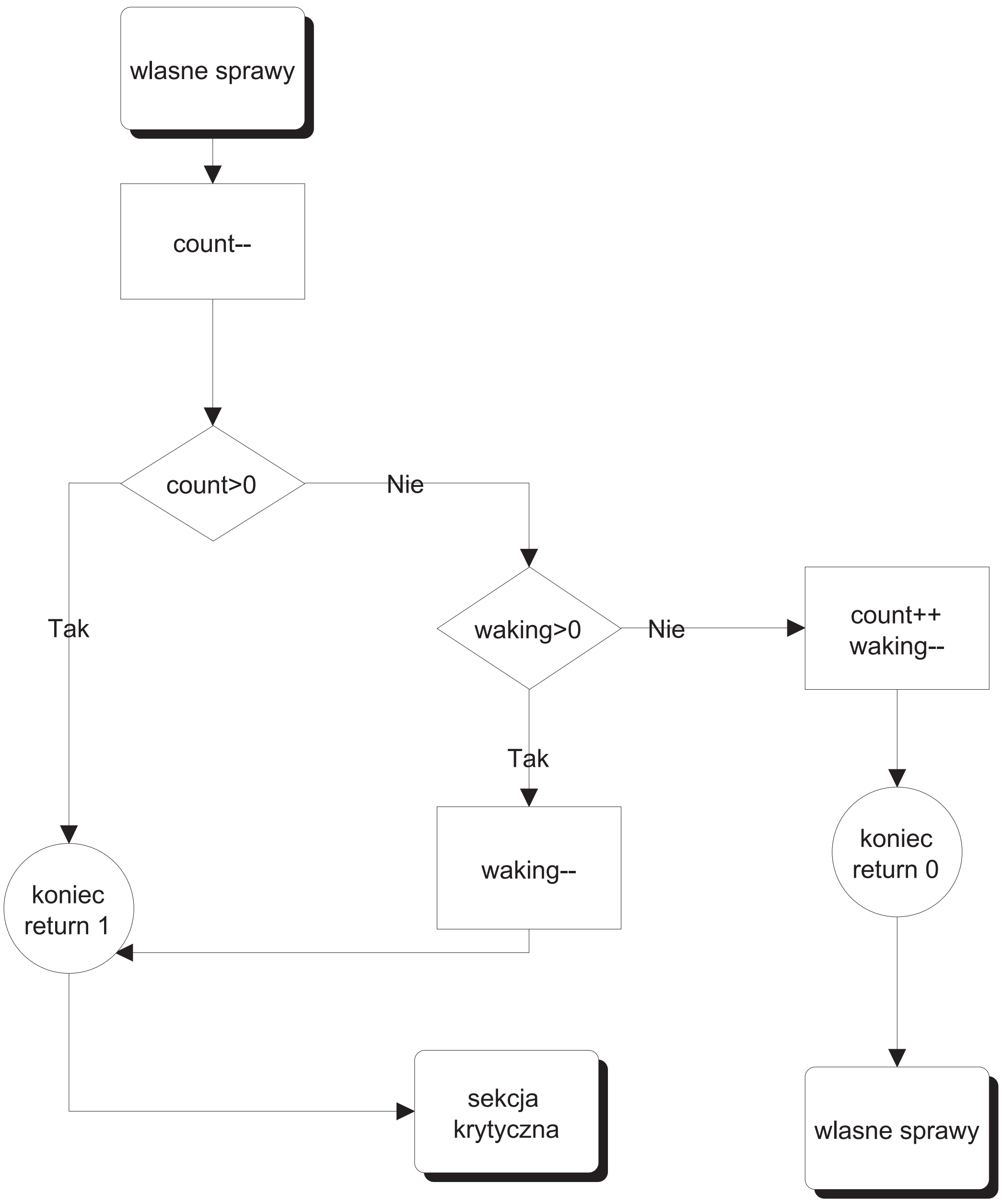
[7] Pliki źródłowe wersji Linuxa 2.2.17: atomic.h, wait.h, sched.h, sched.c, semafor-helper.h, spinlock.h, semafore.S, semafore.h, system.h, errno.h



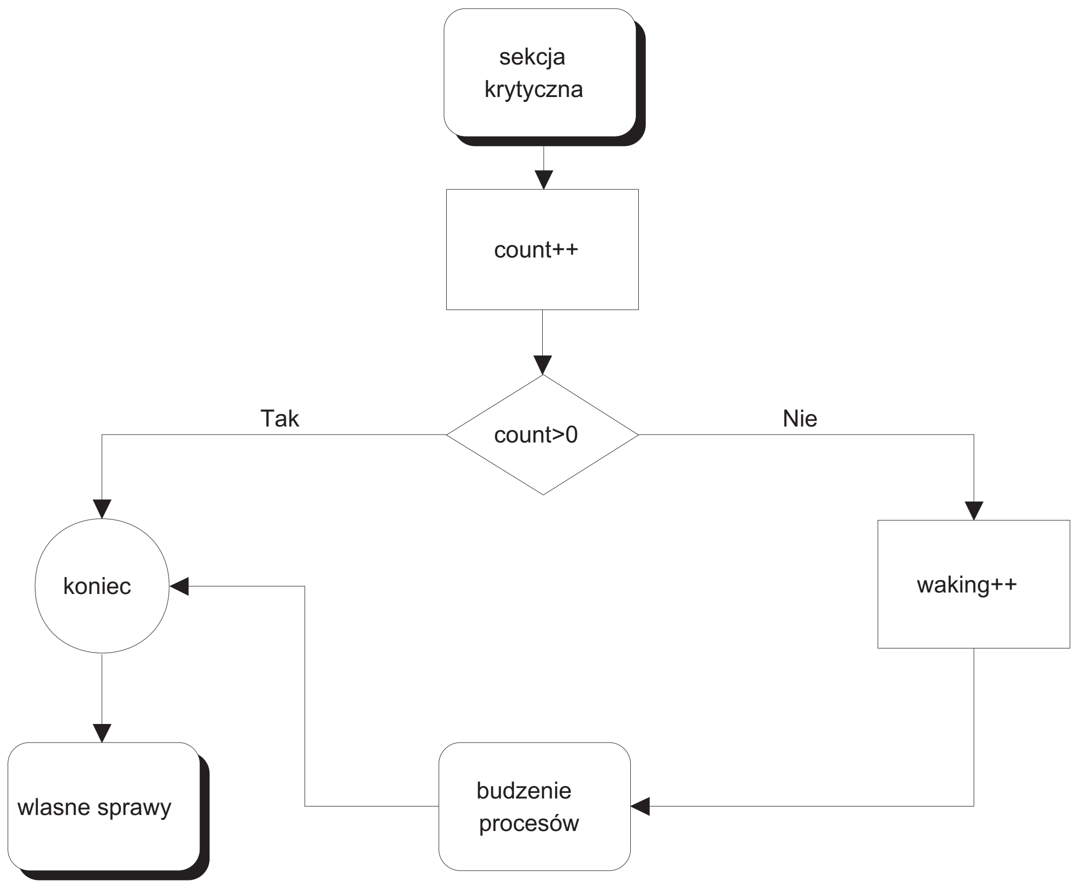
Rysunek 1. Opuszczanie semafora - funkcja down.



Rysunek 2. Opuszczanie semafora - funkcja down_interruptible.



Rysunek 3. Warunkowe opuszczanie semafora.



Rysunek 4. Podnoszenie semafora.