Rozdział 1

Zmiany stanów procesu

Andrzej Wisłowski
Temat nr 9.
Zmiany stanów procesu - algorytmy sleep_on, wake_up, wake_up_process

1.1 Wprowadzenie

W Linuxie procesy są wykonywane współbieżnie. Aby zapewnić bezkonfliktowe dzielenie procesora i obsługę różnych zdarzeń wpływających na wykonywanie programów procesy przebywają w określonym stanie. Jest on zapisany w strukturze procesu `task_struct`, w której znajduje się pole `volatile long state`.

Ze względu na możliwość asynchronicznego zmiany wartości pola **state** w procedurach obsługi przerwań określone jest jako `volatile`. Dzięki temu przy każdym odwołaniu do zmiennej **state** zostanie ona odczytana bezpośrednio z komórki pamięci.

Znaczenie pola **state** nie jest równoważne stwierdzeniu, że proces znajduje się w określonym stanie, ponieważ może się dopiero w nim znaleźć. Dodatkowo można logicznie odróżnić stan, w którym proces jest poza pamięcią pamięcią główną. Proces zostaje wyrzucony z pamięci głównej, jeśli brakuje pamięci. Może być wtedy uśpiony (stany `TASK_UNINTERRUPTIBLE` lub `TASK_INTERRUPTIBLE`) albo czekać na procesor w kolejce procesów gotowych.

Tablica 1.1: Stany są zdefiniowane jako stałe liczbowe

Stan	Liczba
TASK_RUNNING	0
TASK_INTERRUPTIBLE	1
TASK_UNINTERRUPTIBLE	2
TASK_ZOMBIE	4
TASK_STOPPED	8
TASK_SWAPPING	16

Dzięki takiemu określeniu stanu można sprawdzić czy proces znajduje się w danym zestawie stanów przez nakładanie masek bitowych, czyli jedno porównanie.

1.2 Opisy stanów

1.2.1 TASK_RUNNING

Proces aktualnie wykorzystującym procesor lub czeka na jego przydzielenie. W obu tych przypadkach znajduje się na liście procesów gotowych do wykonywania. O procesie, który znajduje się aktualnie w stanie `TASK_RUNNABLE` i jest tym, który ma aktualnie przydzielony procesor, mówimy, że się właśnie wykonuje. Proces taki może wykonywać się w jednym z dwóch następujących trybów:

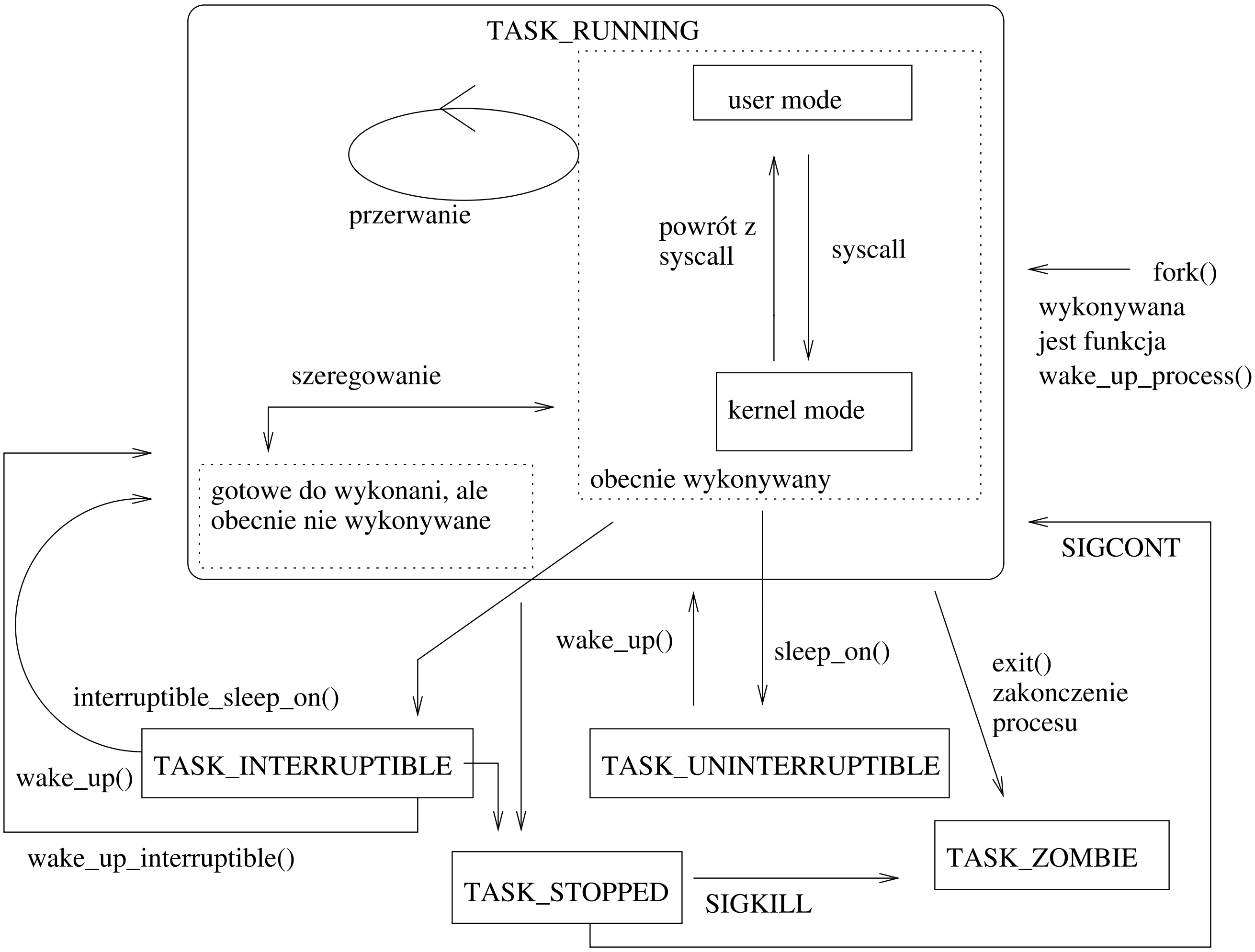
- Tryb użytkownika (user mode) - tryb podstawowy procesu.
- Tryb jądra (kernel mode) - w tym trybie możliwe jest wywoływanie funkcji systemowych (syscall). Proces w trybie jądra nie może być wywłaszczony przez jądro ze względu na możliwość modyfikowania struktur danych jądra, poprzez funkcje systemowe. Gdyby tak nie było mógłby nie dokończyć modyfikacji, będąc przerwany i spowodować niespójność tych struktur.

Przykład:
Proces wywłaszczony przez jądro, np. ze względu na przekroczenie przydzielonego mu kwantu czasu, traci procesor, ale nie zmienia stanu i dalej jest w kolejce procesów gotowych do wykonania. Proces może zostać wywłaszczony tylko jeśli jest w trybie użytkownika. Po określonym wyczerpaniu swojego kwantu czasu procesor traci procesor, dzięki czemu można zaimplementować współzawodnictwo o zasób.

1.2.2 TASK_INTERRUPTIBLE

Proces czeka na zdarzenie lub na zasoby. Ustawia się w kolejce procesów oczekujących na zadane zdarzenie. Po odebraniu sygnału powraca do stanu `TASK_RUNNING`. Nie wypisuje się od razu z kolejki gotowych tylko dopiero w wywołaniu funkcji `schedule()`, jeśli do tej pory nie otrzymał sygnału, w przeciwnym przypadku jest zmieniamy jego stan na `TASK_RUNNING` i nie musi być wpisywany do kolejki procesów gotowych ponieważ już się tam znajduje.

Przykład:
Proces może zażądać np. zapisania na dysku lub czekać na wprowadzenie danych z terminala. W takiej sytuacji jądro (zakładając, ze akurat z dysku korzysta inny proces lub że przy terminalu akurat nikt nie siedzi) ustala stan procesu na `TASK_INTERRUPTIBLE` i szereguje proces do kolejki związanej z odpowiednim zdarzeniem (dostęp do dysku, otrzymanie sygnału z terminala). Podobna jest sytuacja w chwili wywołania przerwania błędu strony. Wykorzystywane jest wtedy nie natychmiastowe wypisywanie z kolejki gotowych procesów. Jeśli strona zostanie wpisana zanim zostanie wywołana funkcja `schedule()`, wtedy wystarczy z powrotem zmienić stan procesu na `TASK_RUNNABLE`.



Rysunek 1.1: Przejścia międzystanowe

1.2.3 TASK_UNINTERRUPTIBLE

Proces czeka na odpowiednie warunki sprzętowe, a więc nie może zostać przerwany przez sygnał.

Przykład:
Kiedy proces wisi na semaforze związanym z wewnętrzna organizacja jądra (dostęp jądra do krytycznych fragmentów pamięci (zmienne)) np. przy tworzeniu procesu, znajduje się w stanie `TASK_UNINTERRUPTIBLE`. Proces w takim stanie jest w trakcie zmiany swoich struktur poprzez jądro. Struktury te mogą być niespójne i próba obsługi sygnału mogłaby napotkać na sprzeczności w tych strukturach.

1.2.4 TASK_STOPPED

Proces może zostać zatrzymany przez właściciela grupy do której należy. Proces został zatrzymany po otrzymaniu jednego z następujących sygnałów: `SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU` lub będąc monitorowanym przez inny proces przy użyciu wywołania systemowego `ptrace` przekazał sterowanie do procesu monitorującego. Sygnały powyższe oznaczają zatrzymanie procesu lub obsługę przechodzenia w tło. Proces może się znaleźć w tym stanie przechodząc ze stanu, w którym może odbierać sygnały tzn. `TASK_RUNNING` (będąc w kolejce procesów gotowych do wykonania) lub `TASK_INTERRUPTIBLE`. Funkcja obsługi przerwania zajmuje się przesłaniem sygnału. Opuszczenie tego stanu możliwe jest jedynie po otrzymaniu sygnału kontynuacji `SIGCONT` lub zakończenia `SIGKILL`.

1.2.5 TASK_ZOMBIE

Proces się zakończył, ale w tablicy procesów wciąż znajduje się odpowiadająca mu struktura `task_struct`, która jest później wykorzystana w wywołaniach systemowych `exit()` i `wait()` procesu ojca.
Przykład:
Proces może chcieć zakończyć się tylko, gdy wszystkie procesy przez siebie stworzone zakończą się. Użyta funkcja `wait()` wykorzysta stan procesu dziecka, aby poinformować o tym proces ojca.

1.2.6 TASK_SWAPPING

Praktycznie nie używany stan od systemu 2.0. Tylko w dwóch miejscach występuje do niego odwołanie przy sprawdzaniu stanu i ustalaniu maski bitowej dla wartości zmiennej `state`, ale nigdy nie może znaleźć się w tym stanie.

1.3 Algorytmy

1.3.1 Wprowadzenie

Często zdarza się, że działający proces musi zostać wstrzymany i może być kontynuowany dopiero po zajściu konkretnego zdarzenia. Takim zdarzeniem może być zwolnienie się zasobu, operacja wejścia wyjścia, podniesienie semafora, pojawienie się informacji w łączu, zakończenie pracy procesu potomnego lub inne. W Linuksie procesy oczekujące na konkretne zdarzenie umieszczane są w osobnych jednokierunkowych kolejkach cyklicznych `wait_queue`.

1.3.2 Usypianie

Następujące funkcje pozwalają na usypianie procesów w oczekiwaniu na zdarzenie.

```
void interruptible_sleep_on(struct wait_queue **p)
void sleep_on(struct wait_queue **p)
long interruptible_sleep_on_timeout(struct wait_queue **p, long timeout)
long sleep_on_timeout(struct wait_queue **p, long timeout)
```

Algorytm 1 {interruptible_}sleep_on{_timeout}

Dane wejściowe: kolejka **p** związana z określonym zdarzeniem
Wynik: Brak

1: zmiana stanu obecnego procesu na
- `TASK_INTERRUPTIBLE` dla {interruptible_}
- `TASK_UNINTERRUPTIBLE` w przeciwnym przypadku
2: zapamiętanie poziomu pracy procesora i zablokowanie przerwań
3: zablokowanie kolejki **p** do pisania
4: wstawianie procesu do kolejki **p**
5: odblokowanie kolejki **p**
6: wywołanie szeregowania (funkcja: `schedule()` lub `schedule_timeout()` dla `{_timeout}`)
7: zablokowanie kolejki **p** do pisania
8: usunięcie procesu z kolejki **p**
9: odblokowanie kolejki **p**
10:przywrócenie początkowego poziomu pracy procesora

Uwagi:

- Funkcja `sleep_on()` dodaje proces do danej kolejki procesów uśpionych na lokalnym stosie danych, w bloku pamięci dostępnej dla wszystkich procesów.
- Funkcja `sleep_on()` nie jest odpowiedzialna za usunięcie procesu z kolejki procesów gotowych tylko funkcja `schedule()`, w której proces oddaje procesor.
- Funkcje z przyrostkiem `_timout` mają taką samą semantykę. Różnią się tylko tym, że wywołują nie funkcję `schedule()`, tylko `schedule_timeout()`. Dzięki temu po okresie czasu `time_out` zostają wypisane z kolejki procesów oczekujących i wracają do stanu `TASK_RUNNING`.

1.3.3 Budzenie

Budzenie procesów ze stanów TASK_UNINTERRUPTIBLE, TASK_INTERRUPTIBLE:

```
void wake_up(struct wait_queue **q)
void wake_up_interruptible(struct wait_queue **q)
```

Algorytm 2 wake_up{_interruptible}

Dane wejściowe: kolejka **q**, z której mają być obudzone procesy
Wynik: Brak
1: zablokowanie kolejki **q** do czytania
dopóki istnieje nieprzetworzony proces z kolejki **q** będących w stanie `TASK_UNINTERRUPTIBLE` { i w `TASK_INTERRUPTIBLE`}
wykonuj
2: wywołanie funkcji `wake_up_process` dla tego procesu
koniec dopóki
3: odblokowanie kolejki **q**

Uwagi:

- Budzi wszystkie procesy z kolejki, choć mogą rywalizować o blokadę i wiele może wrócić do stanu uśpienia.
- Funkcja `wake_up()` nie usuwa procesu z kolejki procesów oczekujących. Dzieje się to w odpowiedniej funkcji `sleep_on()` w momencie gdy proces ponownie dostaje sterowanie po wykonaniu przez schedule.
- W funkcji `wake_up()` jest blokowana kolejka tylko do czytania, ponieważ w tej funkcji nie są zmieniane struktury kolejek procesów czekających. Każdy proces musi sam usunąć się z kolejki, na której czekał, gdy wróci do wykonywania procedury `sleep_on()`, w której zasnął.
- Funkcja `wake_up()` nie budzi procesów zastopowanych (będących w stanie `TASK_STOPPED`). Muszą one być obudzone przez sygnały.

```
void wake_up_process(struct task_struct * p)
```

Funkcja wake_up_process() jest wywoływana między innymi w funkcjach fork(), schedule(), wake_up().

Algorytm 3 wake_up_process

Dane wejściowe: wskaźnik **p** do struktury `task_struct` procesu
Wynik: Brak
1: zapamiętanie poziomu pracy procesora i zablokowanie przerwań
2: zmiana stanu procesu **p** na `TASK_RUNNING`
3: dodaj proces **p** do kolejki procesów gotowych
4: przywrócenie początkowego poziomu pracy procesora
5: wywołanie funkcji udostępniającej nam procesor, jeśli jest wykonywany proces idle

Uwagi:

- Jeśli jest wykonywany proces `idle` tzn. nie ma żadnego innego procesu gotowego do wykonania.

1.4 Zadania

1. Co się zdarzy, gdy proces zaśnie i nie będzie gotowych.
2. W algorytmach sleep_on i wake_up_process jądro podnosi poziom pracy procesora, aby zapobiedz przerwaniom. W jaki sposób mogłoby zostać rozspójnione struktury jądra, gdyby się tego nie zrobiło.
3. Proces próbuje zasnąć na kolejce zdarzeń i zostanie wywołane budzenie procesów z tej kolejki jeszcze zanim proces podniesie poziom pracy procesora. Co się stanie wtedy z tym procesem?
4. Jakie są możliwe różne sytuacje, w których proces może opuścić kolejkę procesów oczekujących na zdarzenie?

itd.

1.5 Rozwiązania zadań

1. Będzie wykonywany proces `idle` wykonujący nieskończoną pętlę.
2. Dodawanie i usuwanie procesów z kolejki oczekujących na zdarzenie musi być sekcją krytyczną. Zmienianie listy równocześnie mogłoby ją rozspójnić.
3. Proces nie wypisuje się z kolejki procesów gotowych w funkcji `sleep_on()`, ale dopiero w funkcji `schedule()`. Tam jest sprawdzane czy już nie nadszedł sygnał. Jeśli nadszedł to proces nie zaśnie, ale uzyska stan `TASK_RUNNING` i po otrzymaniu procesora wychodząc z funkcji `sleep_on` sam się wypisze z kolejki procesów oczekujących na dane zdarzenie.
4. Proces może być:
   - Obudzony przez funkcję `wake_up()`
   - Jeśli był w stanie `TASK_INTERRUPTIBLE` tzn. zasnął w funkcji `interruptible_sleep_on()` to może obudzić go sygnał
   - Jeśli zasnął w funkcji `sleep_on _timeout()` lub `interruptible_sleep_on_timeout()` to po upłynięciu czasu timeout zostanie samoczynnie obudzony.

1.6 Słownik

- **poziom pracy procesora** - definiuje jakie przerwania są w obecnej chwili zablokowane
- **{}** - oznacza opcjonalność

Literatura

[Bach 95] Bach M.: *Budowa systemu operacyjnego Unix.* WNT, 1995

[Kernel] Źródła jądra:
linux/kernel/sched.h
linux/kernel/sched.c

[Projekty Linux] *Projekty Linux na Systemach operacyjnych na MIMUW*
rainbow.mimuw.edu.pl/SO