Wykład 2
Proces, stany procesu
i przejścia pomiędzy nimi.

Przypomnienie z poprzedniego semestru

- W systemach jednoprogramowych (MS-DOS) proces.
  - Wywołuje funkcję systemu w celu wykonania-operacji we-wy.
  - System operacyjny inicjalizuje operację i czeka aż się ona zakończy.
  - Następuje powrót do procesu..
- W systemach wieloprogramowych rzecz jest o wiele bardziej skomplikowana Proces A:
  - Wywołuje funkcję systemu w celu wykonania-operacji we-wy.
  - System operacyjny inicjalizuje operację, i wprowadza proces w stan zawieszenia (zapamiętanie kontekstu). Mówimy że ***jądro usypia (ang. sleep) proces A.***.
- System wybiera inny proces gotowy do wykonania i następnie przekazuje mu procesor (przełącza kontekst do tego procesu).
- Po pewnym czasie zakończenie operacji We-Wy sygnalizowane jest (na ogół) przerwaniem.
  - W handlerze obsługi przerwania ***jądro budzi (ang wakeup) proces A.***
  - Po pewnym czasie proces otrzymuje ponownie procesor.
- Cel dzisiejszego wykładu: Opisać działanie tego mechanizmu w jądrze Linuxa

Struktura task_struct

- Jest zdefiniowana w `include/linux/sched.h`.
- Zawiera wszystkie niezbędne informacje o procesie. Dla każdego procesu w Linuksie musi istnieć odpowiadająca mu instancja struktury `task_struct`
- Wektor (tablica zadań) jest zdefiniowany w tym pliku jako

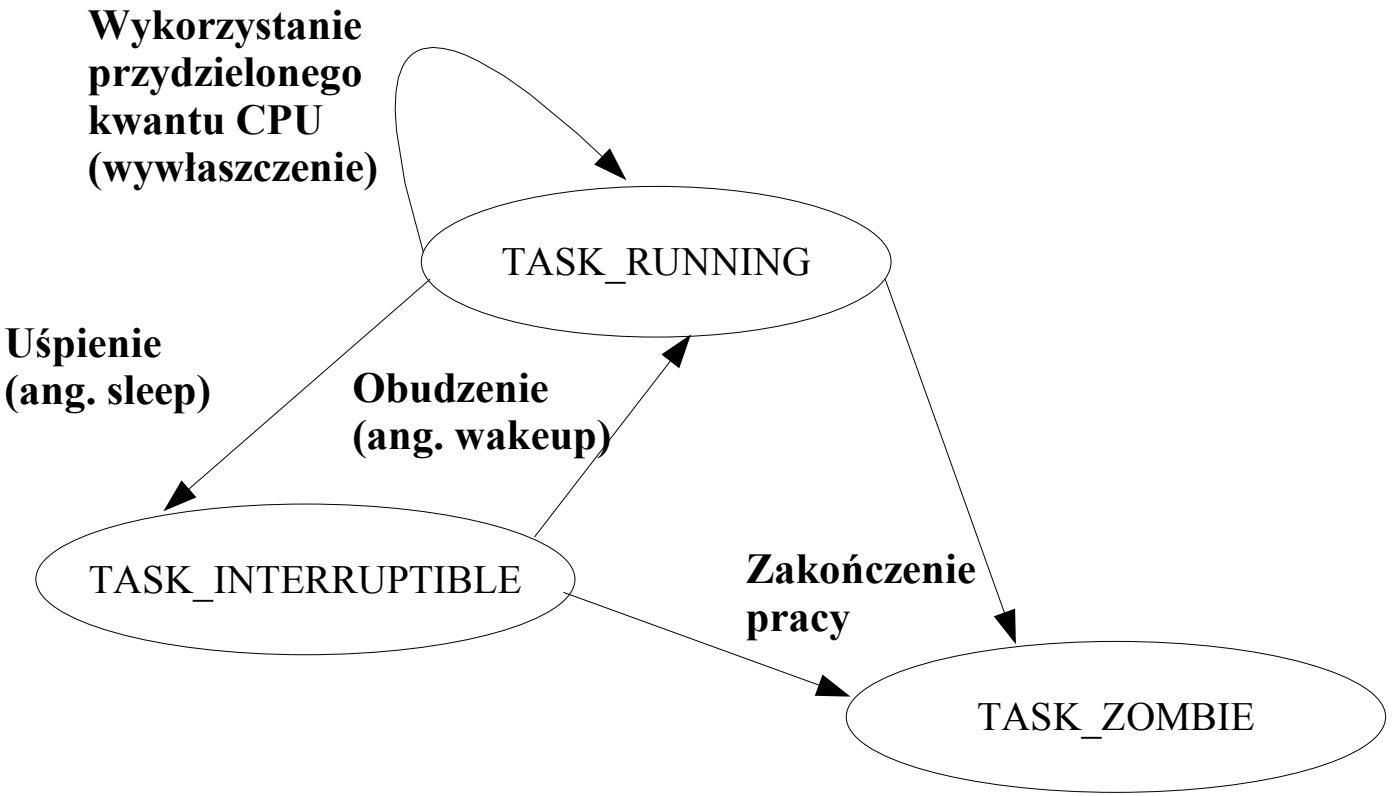
```
extern struct task_struct *task[NR_TASKS];
```

- Wartość stałej NR_TASKS zdefinowana w jako `include/linux/tasks.h` jako 512.
- Zdefiniowana jest również zmienna (tak naprawdę makrodefinicja z uwagi na systemy wieloprocesorowe SMP) `current` będąca wskaźnikiem do bieżącego procesu.
- Struktura `task_struct` zawiera bardzo wiele pól (proszę nie wpadać w panikę), część z nich zostanie omówiona na dzisiejszym wykładzie, część na kolejnych

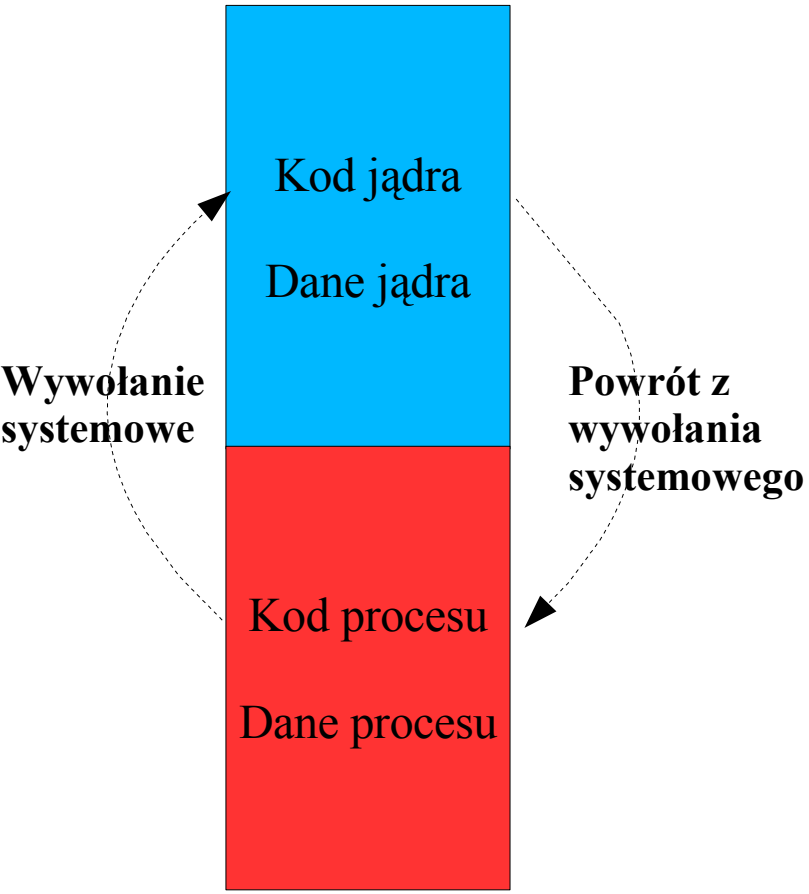
Stan procesu

- Pierwsze (i najważniejsze) pole w strukturze `task_struct` to `state`. Przechowuje ono informacje o aktualnym stanie procesu. Możliwe wartości, zdefiniowane w `include/linux/sched.h` to,
  - TASK_RUNNING : proces albo aktualnie się wykonuje na procesorze, albo jest gotowy do wykonania i czeka na otrzymanie procesora. ***Brak rozróżnienia stanu aktywnego i gotowego (ale możemy porównać się ze wskaźnikiem `current`)***.
    - Tylko procesy w stanie TASK_RUNNING są brane pod uwagę przez planistę procesora (funkcję schedule) przy poszukiwaniu procesu do którego przełączyć procesor.
  - TASK_INTERRUPTIBLE : proces oczekuje na zajście zdarzenia lub na jakiś zasób, może otrzymać sygnał.
  - TASK_UNINTERRUPTIBLE : proces oczekuje na zajście zdarzenia lub na jakiś zasób nie reaguje na sygnały.
  - TASK_STOPPED : proces został zatrzymany (np. przez sygnał SIGSTOP).
  - TASK_ZOMBIE : proces zakończył pracę, ale jego `task_struct` jest nadal obecny w pamięci. Ten dodatkowy stan jest potrzebny do poprawnego zaimplementowania funkcji systemowej `wait`.
  - TASK_SWAPPING : nie używane

Diagram zmiany stanów procesów



Tryb jądra oraz tryb użytkownika



- Proces wykonujący swoje własne instrukcje ***wykonuje się w trybie użytkownika.***
  - W tym trybie może nastąpić wywłaszczenie procesu (w wyniku przerwania zegarowego).
  - Proces ma dostęp wyłącznie do swoich danych i kodu.
  - Kod i dane jądra są zamapowane w przestrzeni adresowej procesu, jednak próba dostępu powoduje błąd.
- Z chwilą wywołanie systemowego, zaczynją wykonywać się instrukcję jądra. Pomimo, że CPU wykonuje kod jądra, to mówimy, że ***proces wykonuje się w trybie jądra***.
  - W tym trybie możliwy jest dostęp zarówno do kodu i danych procesu oraz jądra.
  - Proces nie może zostać wywłaszczony.
  - Problemy z bezpieczeństwem, np. Co się stanie gdy jako argument funkcji read podamy adres wskazujący na pamięć jądra

Listy dwukierunkowe

- Wszystkie procesy w systemie tworzą listę dwukierunkową.
  - Pola `prev_task` oraz `next_task` w strukturze `task_struct` to wskaźniki na poprzedni i następny element tej listy. Początek listy to
- Wszyskie procesy w stanie `TASK_RUNNING` tworzą listę dwukierunkową.
  - Pola `prev_run` oraz `next_run` w strukturze `task_struct` to wskaźniki na poprzedni i następny element tej listy. Początek listy to
  - Lista ta wykorzystywana jest przez planistę procesora (funkcja `schedule`).
- Pozycja o numerze 0 w tablicy wskaźników `task` jest zarezerwowana dla procesu systemowego bezczynnego. Wskazuje ona na zmienną `init_task`.
- `init_task` jest pierwszym i ostatnim elementem dla obydwu list dwukierunkowych. Stąd np. makrodefinicja iterująca wszyskie procesy ma postać:

```
#define for_each_task(p) \
        for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

Kolejki procesów oczekujących na zdarzenie (ang. wait queues)

- Uśpione procesy oczekujące na zdarzenie przechowywane są w kolejce. W plikach `/include/linux/wait.h` oraz `include/linux/sched.h` zdefiniowane są:

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

- Kolejka procesów oczekujących na zdarzenie jest zaimplementowana jako lista cykliczna takich struktur. Aby wykorzystać z kolejki należy zadeklarować zmienną `struct wait_queue * p` będącą (wkaźnikiem na) początkiem kolejki. Do operacji na kolejkach służą następujące funkcje:

```
void init_waitqueue(struct wait_queue **q) // inicjalizacja kolejki

// Dodanie elementu wait do kolejki p;
void add_wait_queue(struct wait_queue ** p, struct wait_queue * wait)

// Usunięcie elementu wait z kolejki p;
void remove_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
```

- ***Praca domowa***: przeanalizuj kod tych funkcji i zauważ że blokują one przerwania (funkcja `cli()`) na czas modyfikacji kolejki (dlaczego jest to konieczne ?)

Usypianie procesu - scenariusz

1. Proces wywołuje funkcje systemową read() - przypuśćmy że jest to odczyt znaku z konsoli.
2. Jądro wywołuje sterownik konsoli.
3. Sterownik konsoli wywołuje funkcję `interruptible_sleep_on(console_queue)`, gdzie `console_queue` jest kolejką procesów oczekujących na dane z konsoli.
4. Uwaga: ten punkt zostaje wykonany dopiero gdy:
   - Proces zostanie obudzony (np. w procedurze obsługi przerwania klawiatury)
   - Planista (funkcja schedule) wybierze ten proces do wykonywania.

   Następuje skopiowanie znaków z bufora klawiatury do przestrzeni użytkownika i powrót z wywołania systemowego.

Pomiędzy punktami 3 i 4 zachodzą więc dwa przełączenia konteksktu (od procesu usypianego i ponownie do procesu usypianego)

interruptible_sleep_on

- Pełna definicja w kernel/sched.c, uproszczony kod ma postać:

```
void interruptible_sleep_on(struct wait_queue **q) {

    //element kolejki reprezentujący bieżące zadanie
    struct wait_queue entry={current, NULL};

    current->state=TASK_INTERRUPTIBLE;
    add_wait_queue(q,&entry);
    schedule(); // BARDZO WAŻNE: Planista procesora
    remove_wait_queue(q,&entry); // ***
}
```

- Funkcja `schedule` (a) wyszukuje inny proces o stanie TASK_RUNNIING i (b) przełącza do niego kontekst (kod częściowo zależny od sprzętu).
- Aby proces kontynuował wykonanie (od linii //***)
  - Stan procesu musi zostać zmieniony ponownie na TASK_RUNNING.
  - Musi zostać ponownie wywołana funkcja `schedule` np. W sytuacji gdy inny proces wykona `*_sleep_on` lub inny proces zużyje swój kwant czasu procesora, co jest sprawdzane przy obsłudze przerwania zegarowego.
  - Ponadto funkcja `schedule` musi zdecydować, że procesor należy się właśnie temu procesowi.

Budzenie procesu - scenariusz

- Następuje przerwanie klawiatury.
- Procedura obsługi przerwania klawiatury sprawdza, że nastąpiło wprowadzenie nowego znaku i wywołuje funkcję `wake_up_interruptible(console_queue)` budzącą wszystkie procesy czekające na kolejce `console_queue`. Procesy te otrzymują stan TASK_RUNNING.
- Oznacza to, że mogą one zostać wybrane do wykonywania przy kolejnym wywołaniu planisty `schedule`.
- Mamy do czynienia z asymetrią:
  - Uśpienie procesu => utrata procesora
  - Obudzenie procesu => to wcale nie jest przydzielenie procesora (bieżący proces wskazywany przez `current` kontynuuje pracę). ***To danie szansy na przydział procesora przy następnym wywołaniu schedule.***

wake_up_interruptible

```
void wake_up_interruptible(struct wait_queue **q)
{
        struct wait_queue *next;
        struct wait_queue *head;

        next = *q;
       head = WAIT_QUEUE_HEAD(q);
        while (next != head) {
                struct task_struct *p = next->task;
                next = next->next;
                if (p->state == TASK_INTERRUPTIBLE) {
                                unsigned long flags;

                                 save_flags(flags);
                                 cli();
                                 p->state = TASK_RUNNING;
                                 if (!p->next_run)
                                     add_to_runqueue(p);
                                 restore_flags(flags);

                }
        }
}
```

Generalnie algorytm polega na nadaniu wszystkim procesom w kolejce stanu TASK_RUNNING. Funkcja `add_to_runqueue` umieszcza proces na dwukierunkowej liście procesów gotowych do wykonania pola next_run oraz struktury .

Synchronizacja kodu jądra - wersja jednoprocesorowa

- Często do danych struktur jądra np. dane sterownika możemy odwoływać się z np. (A) różnych ścieżek wykonania jądra (np. dwa procesy wykonujące się z trybu jądra piszą do tego samego urządzenia) (B) ze ścieżki jądra (proces pisze do urządzenia) i z handlera przerwania. W takiej sytuacji musimy zapewnić synchronizację dostępu do danych jądra - tworząc sekcje krytyczne.
- Linuks w wersji 2.0.x wykorzystuje rozwiązanie znane z klasycznego Uniksa.
- W przypadku (A) synchronizację zapewnia prosta zasada: ***proces wykonujący się w trybie jądra nie może zostać wywłaszczony.*** Przez wywłaszczenie rozumiemy odebranie procesu bez jego zgody i wiedzy. Proces może jedynie zrzec się procesora wywołując funkcję *schedule*.
  - Oczywiście powyższa zasada nie dotyczy procesów wykonujących się w trybie użytkownika.
  - Uwaga niektóre funkcje np. kmalloc z drugim argumentem GFP_KERNEL mogą uśpić proces wykonując algorytm _sleep_on i i pośrednio wywołać *schedule*.
  - W takiej sytuacji możemy skorzystać z semaforów jądra.
- W przyadku (B) sychronizację zapewnia możliwość wykorzystania pary funkcji *cli()*/*sti()* blokujących i odblokowujących przerwania.

Przykład wykorzystania operacji usypiania i budzenia: semafory jądra

- Jest to jeden z mechanizmów jądra wykorzystywany do ochrony krytycznych danych. Rozważmy sytuacje w której proces, wykonywujący się w trybie jądra, jest w trakcie modyfikacji jakiejś struktury danych, z pewnych przyczyn (nie mówimy jakich) zasypia. Wówczas możliwy jest scenariusz, w którym planista `schedule` przekaże procesor innemu procesowi, który wywoła funkcje systemową, wejdzie w tryb jądra i spróbuje zmodyfikować tę samą strukturę.
- Rozwiązanie: chronić dostęp do danych krytycznych przy pomocy semforów.
- API semaforów jest zdefiniowane w `/include/asm-i386/semaphore.h`. Składa się z następujących funkcji:

```
void down(struct semaphore * sem);
void up(struct semaphore * sem);
```

oraz z makrodefinicji MUTEX inicjalizującej strukturę `struct semaphore`. Funkcja `down` rezerwuje, a `up` zwalnia semafor.

Implementacja operacji semaforowych

- Rzeczywista implementacja uwzględnia systemy wieloprocesorowe i jest zoptymalizowana w assemblerze, przy założeniu jednego procesora możemy ją uprościć do postaci:

```
struct semaphore {
   int count;
   struct wait_queue wait;
}

void down(struct semaphore *sem) { // wejście do semafora
    while(sem->count<=0)
        sleep_on(sem->wait);
    sem->count--;
}

void up(struct semaphore *sem) { // wyjście z semafora
    sem->count++;
    wake_up(&sem->wait);

}
```

- Przypominam, że proces wykonujący się w trybie jądra nie może zostać wywłaszczony. Stąd gwarancja atomiczności operacji `down` oraz `up`.

Przykład wykorzystania operacji semaforowych

- Pierwszy proces w trybie jądra:

```
down(&semafor);
// Kod sekcji krytycznej
.................
// kmalloc może uśpić proces
kmalloc(n,GFP_KERNEL)
// Ciąg dalszy sekcji krytycznej
.................
// Koniec sekcji krytycznej
up(&semafor);
```

- Drugi proces w trybie jądra:

```
down(&semafor);
// Kod sekcji krytycznej
.................
// Koniec sekcji krytycznej
up(&semafor);
```

- Gdyby nie kmalloc lub inne tego typu funkcje, nie ma potrzeby stosowania semaforów (jądro nie jest wywłaszczalne !!!).
- W tej chwili gdy proces po lewej stronie zostanie uśpiony w wyniku wywołania funkcji kmalloc (w oczekiwaniu na odzyskanie wolnej pamięci), proces po prawej stronie zostanie uśpiony na semaforze - i nie wejdzie do sekcji krytycznej.
- Gdy pamięć się znajdzie (jeżeli nie, to blokada :)) proces pierwszy zostanie obudzony, wróci z funkcji malloc, wyjdzie z sekcji krytycznej i pozwoli procesowi drugiemu (czekającemu na semaforze) na wejście

Dalsze pola struktury task_struct (1)

```
long counter;
long priority;
```

Wykorzystywane przez planistę `counter`.

```
unsigned long signal;
unsigned long blocked;
```

Obsługa sygnałów POSIX. Maska bitowa zgłoszonych i zablokowanych sygnałów.

```
unsigned long flags;
```

Flagi procesu (proces monitorowany, ropoczynany, kończony).

```
int errno;
```

Kod błędu ostatniego wywołania systemowego.

```
int pid,pgrp,session,leader;
```

Identyfikator procesu, grupy procesów, sesji i lidera sesji.

```
Unsigned short uid,gid,euid,egid;
```

Identyfikator użytkownika i grupy oraz dwa identyfikatory rzeczywiste.

Dalsze pola struktury task_struct (2)

```
struct wait_queue *wait_child_exit;
```

Kolejka wykorzystywana przez funkcje rodziny wait czekające na zakończenie procesu potomnego

```
struct files_struct *files;
```

Informacje o otwartych plikach postaci:

```
struct files_struct {
        int count;
        fd_set close_on_exec;
        fd_set open_fds;
        struct file * fd[NR_OPEN];
}
```

NR_OPEN=256, `count` jest liczbą aktualnie otwartych plików, `open_fds` jest maską bitową otwartych plików, `close_on_exec`l maska bitowa plików które muszą być zamknięte po wywołaniu exec, ma rozmiar 32 bajtów, typ `fd_set` (256 bitów)

Dalsze pola struktury task_struct (3)

```
init-+-5*[agetty]
     |-bash---mc-+-bash---pstree
     |           `-cons.saver
     |-crond
     |-gpm
     |-kerneld
     |-kflushd
     |-klogd
     |-kswapd
     |-syslogd
     `-update
```

- Powyższy listing to wynik polecenia pstree.

```
struct task_struct *p_opptr,*p_pptr,*p_cptr,*p_ysptr,*p_osptr;
```

- Wskaźniki do oryginalnego rodzica, rodzica, najmłodszego potomka, najmłodszego brata i najstarszego brata procesu.

Dygresja – wątki jądra (ang. kernel thread)

- Wątek jądra – proces “nie mający części użytkownika”
  - Nie posiada pamięci użytkownika – zatem wykonuje się w trybie jądra i jest częścią jądra.
  - Posiada własny task_struct i podlega planowaniu przez planistę jak zwykłe procesy.
  - Widoczny jest po wykonaniu polecenia ps.
- Tworzymy go przy pomocy funkcji:

```
pid_t kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
```

- Wątek jądra rozpoczyna się od funkcji fn.
- argument arg zostanie przekazany funkcji fn.
- Jeżeli wątek jądra zostanie stworzony z kontekstu zwykłego procesu (a nie innego wątku jądra), to pamięć tego procesu zostanie zwolniona dopiero wtedy, gdy wątek oraz proces zakończą pracę !!!

Następny wykład

- Sygnały
- Sterownik urządzenia znakowego