Wykład 6
Planista procesora – funkcja schedule

Typowe punkty wywołania planisty

- Procedura sleep_on (usypanie procesu).
- Powrót z wywołania systemowego lub przerwania (*ret_from_sys_call*)
  - Przerwanie musi przerwać proces wykonujący się w trybie użytkownika.
  - Planista jest wywoływany o ile ustawiona jest zmienna *need_resched*.
    - need_resched może być ustawiona przy budzeniu !!!
  - Ten punkt wywołania pozwala na wywłaszczenie procesu.
- Aktywne oczekiwanie w sterowniku urządzenia
  - Na przykład oczekiwanie na transfer danych.
  - W pętli czekającej należy wstawić instrukcję:

```
if (need_resched)
    schedule();
```

- Oraz w innych miejscach.

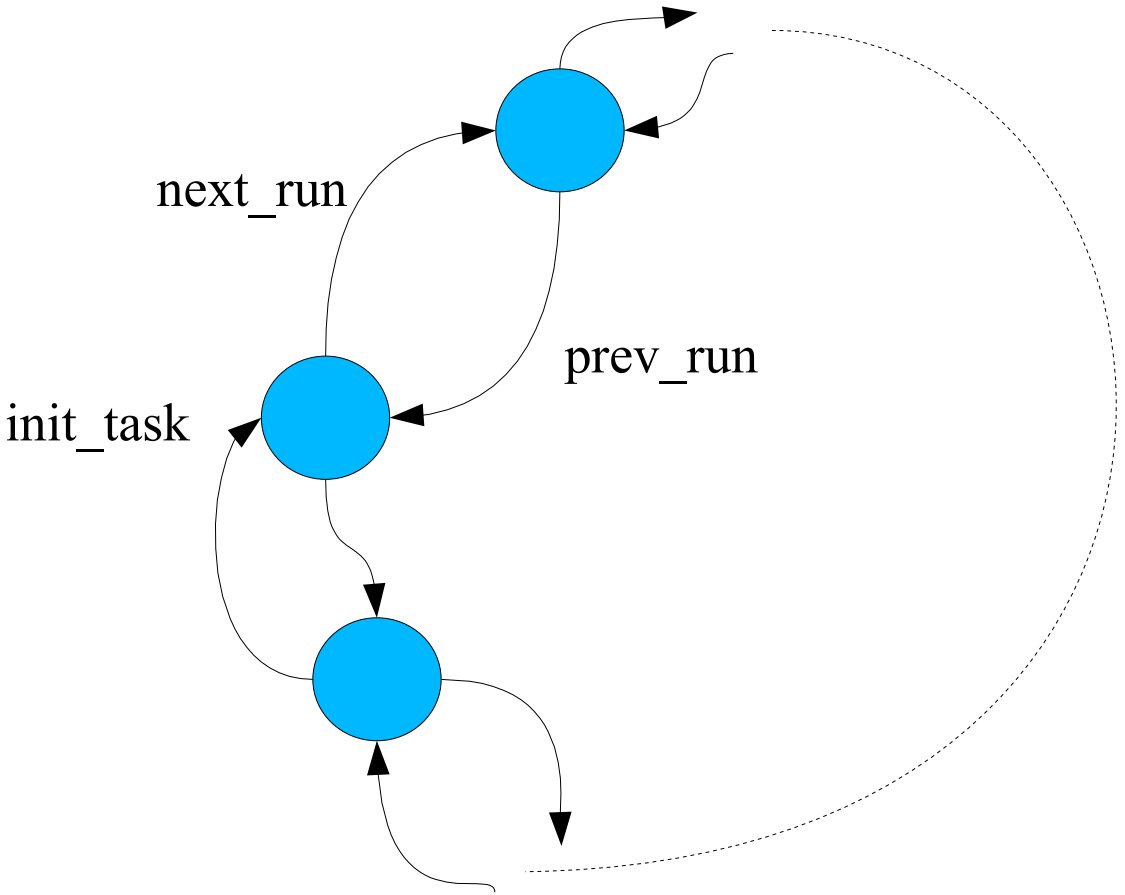
Przeznaczenie funkcji schedule – jedno zdanie

„Wybierz odpowiedni proces w stanie TASK_RUNNING i przełącz do niego kontekst (sterownie)"

Przypomnienie: Pola w struct task_struct

- state: bieżący stan procesu, dwa najważniejsze to:
  - TASK_RUNNING wykonywujący się lub gotowy do wykonania.
    - task_struct wykonywującego się procesu jest wskazywane przez zmienną current
  - TASK_INTERRUPTIBLE uśpiony, ale mogący odbierać sygnały
- counter, priority, rt_priority – Pola związane z priorytetami procesów (statyczny i dynamiczny)
- prev_run, next_run – wskaźniki na następny i poprzedni element na dwukierunkowej liście wszystkich procesów gotowych do wykonania.
- Policy – klasa szeregowania zgodnie ze standardem POSIX 1003.4
  - Dopuszczalne wartości to SCHED_FIFO, SCHED_RR oraz SCHED_OTHER
  - SCHED_OTHER – tradycyjne szeregowanie Uniksowe
  - SCHED_RR oraz SCHED_FIFO to szeregowanie czasu rzeczywistego.
  - Funkcja systemowa sched_setscheduler pozwala m.in na zmianę klasy szeregowania, szeregowanie czasu rzeczywistego może być wykorzystane wyłącznie przez root-a

Cykliczna lista dwukierunkowa procesów gotowych



- Podobnie lista wszystkich procesów (utrzymywana przy pomocy pól next_task i prev_task)

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

Klasy szeregowania

- SCHED_FIFO. (First In First Out – pierwszy przyszedł pierwszy odszedł). Klasa szeregowania czasu rzeczywistego. Proces o tej klasie może zostać wywłaszczony jedynie przez inny proces czasu rzeczywistego o wyższym priorytecie.
- SCHED_RR (Round Robin - w kółko Macieju) Klasa szeregowania czasu rzeczywistego. Proces tej klasy po zużyciu swego kwantu czasu jest przemieszczany na koniec kolejki procesów gotowych.
- SCHED_OTHER tradycyjna klasa procesów UNIXowych.
- Procesy klasy SCHED_FIFO i SCHED_RR wykonywane są zawsze przed procesami klasy SCHED_OTHER.
- O kolejności wykonywania procesów klasy czasu rzeczywistego decyduje
  - Priorytet (pole *rt_priority*).
  - Przy identycznym priorytecie kolejność procesów w kolejce procesów gotowych.

Priorytety (pola w struct task_struct)

- rt_priority: priorytet procesów czasu rzeczywistego.
  - Wartość z zakresu 1-99, 0 dla zwykłych procesów.
- counter – liczba taktów (przerwań zegara systemowego) pozostałych do czasu zużycia przyznanego kwantu czasu (zmniejszane o jeden po każdym przerwaniu).
  - Jeżeli counter osiągnie zero ustawiana jest flaga need_resched
  - counter jednocześnie pełni funkcję priorytetu dynamicznego dla procesów zwykłych. Jeżeli kilka takich procesów jest w stanie gotowym to zostanie wybrany proces o największej wartości counter.
- priority – priorytet statyczny, ustawiany przy pomocy funkcji nice.
  - Nice przyjmuje argument wartość od -20 do 20.
  - priority ma wartość od 0 do 2*DEF_PRIORITY, domyślnie DEF_PRIORITY
  - DEF_PRIORITY=20*HZ/100 (20 dla x86).
- Przy obliczaniu kwantu czasu dla procesów zwykłych wykorzystuje się algorytm:

  counter := priority (dla procesów w stanie TASK_RUNNING, kwant 200ms)

  counter := counter/2 + priority (dla procesów w stanie TASK_INTERRUPTIBLE)

Kod funkcji schedule

- Pomijamy systemy z wieloma procesorami (SMP).
  - Na maszynach SMP każdy procesor wykonuje funkcje schedule().
  - Pożądane jest aby proces otrzymał procesor na którym się poprzednio wykonywał (afiniczność procesorów).
    - Ten procesor może już mieć właściwe dane w swojej pamięci cache.

- Funkcje pomocnicze operujące na kolejce procesów gotowych (prev_run, next_run)
  - move_last_runqueue przesuwa proces na koniec kolejki
  - del_from_runqueue usuwa proces z tej kolejk
  - add_to_ruqueue dodaje proces do kolejki (wykorzystywana przez algorytm wakeup, więcej o niej później).

Funkcja pomocnicza goodness

```
int goodness(struct task_struct *p)
{
    if (p->policy != SCHED_OTHER)
        return 1000+p->rt_priority;
    return p->counter;
}
```

- Funkcja ta wykorzystana jest do selekcji procesu z kolejki procesów gotowych do wykonania.
- Procesy klasy czasu rzeczywistego są sortowane na podstawie pola rt_priority.
- Procesy zwykłe na podstawie pola counter.

  - Im więcej czasu pozostało procesowi, tym większy priorytet.

- Ponieważ counter << 1000, to procesy czasu rzeczywistego mają bezwzględny priorytet.

Funkcja schedule (1)

```
void schedule()
{
    int c;
    unsigned long timeout=0;
    struct task_struct *p, *prev, *next;
    if (bh_active & bh_mask) {
        intr_count = 1;
        do_bottom_half();
        intr_count = 0;
    }
    run_task_queue(&tq_scheduler);

    // need_resched==1 oznacza żądanie uruchomienia planisty
    need_resched = 0;
    // prev jest wskaźnikiem na poprzedni proces
    prev = current;
    cli(); // blokuj przerwania

    if(!prev->counter && prev->policy == SCHED_RR) {
        prev->counter = prev->priority;
        move_last_runqueue(prev);
    }
```

Uruchomienie dolnych połów i kolejki zadań tq_scheduler

- Jeżeli proces jest klasy SCHED_RR i zużył kwant procesora to przesuwany jest na koniec kolejki procesów gotowych i otrzymuje nowy kwant procesora.

Funkcja schedule (2)

```
switch (prev->state) {
        case TASK_INTERRUPTIBLE:
                if (prev->signal & ~prev->blocked)
                        goto makerunnable;
                timeout = prev->timeout;  // Programowana czasowo pobudka
                if (timeout && (timeout <= jiffies)) {
                        prev->timeout = 0;
                        timeout = 0;
        makerunnable:
                        prev->state = TASK_RUNNING;
                        break;
                }
        default:
                del_from_runqueue(prev);
        case TASK_RUNNING:
}
```

- Jeżeli bieżący proces jest w stanie TASK_INTERRUPTIBLE (kładzie się spać) ale:

  b) otrzymał sygnał

  c) upłynął timeout – o tym później

  to przełącz go z powrotem w stan TASK_RUNNING

- W przeciwnym wypadku usuń proces (o ile nie jest w stanie TASK_RUNNING) z kolejki procesów gotowych

Funkcja schedule (3)

```
c = -1000;
p=init_task.next_run;
next = init_task;
while (p != &init_task) {
        int weight = goodness(p, prev, this_cpu);
        if (weight > c)
                c = weight, next = p;
        p = p->next_run;
}
```

- Znajdź w kolejce procesów gotowych proces o największej “dobroci”
  - Za chwilę otrzyma on procesor.

```
if (!c) {
        for_each_task(p)
                p->counter = (p->counter >> 1) + p->priority;
}
```

- Jeżeli c==0, co oznacza że:
  - W kolejce procesów gotowych brak procesów czasu rzeczywistego.
  - Wszystkie procesy zwykłe zużyły swoje kwanty czasu

to przydziel nowe kwanty czasu (wszystkim, nie tylko gotowym) procesom

  - Możemy to nazwać przeliczeniem priorytetów dynamicznych
  - Procesy śpiące otrzymują większy kwant czasu.

Przeliczenie priorytetów (kwantów czasu)

- Wykonywane po zużyciu przez wszyskie procesy w stanie TASK_RUNNING swoich kwantów czasu.
- UWAGA: ***nowe kwanty są nadawane także procesom śpiącym*** (w stanie TASK_INTERRUPTIBLE) !
- Proces który cały czas się wykonuje (a zatem zużył swój kwant otrzyma nowy równy polu priority)
- Proces, który przez cały czas śpi otrzyma, będzie miał kwant:

```
priority                                        // na początku
priority+priority/2                             // pierwsze przeliczenie
priority+priority/2+priority/4                  // drugie przeliczenie
priority+priority/2+priority/4+priority/8      // trzecie przeliczenie

.............................

Szereg geometryczny zbieżny do 2*priority !!!
```

- W ten sposób Linuks preferuje procesy interakcyjne, to znaczy takie które (prawie) cały czas śpią (np. edytor tekstu vi)

Dygresja – oczekiwanie w jądrze (przypomnienie z poprzedniego tygodnia)

- Jednym ze sposobów realizacji aktywnego czekania w jądrze (np. w sterowniku urządzenia) jest wstawienie kodu:

```
if (need_resched)
    schedule();
```

  - Wadą tego podejścia jest bezczynne oczekiwanie w pętli i marnowanie kwantu procesora.

- Alternatywne podejście polega na uśpieniu na co najmniej K milisekund:

```
current->state=TASK_INTERRUPTIBLE;
current->timeout=jiffies+K*HZ/1000;
schedule();
```

- Czas odliczany jest z dokładnością przerwań zegarowych (10ms na x86)
- Proces zostanie obudzony po zaprogramowanym czasie.
  - Ale procesor może otrzymać o wiele później !!!
  - O tym decyduje planista.

API timerów jądra (można wykorzystać w projekcie)

```
struct timer_list {
        struct timer_list *next;
        struct timer_list *prev;
        unsigned long expires; // Kiedy timer wygasa (w jiffies)
        unsigned long data; // Dane do przekazania funkcji
        void (*function)(unsigned long); // Adres handlera
};
```

- Timer jest typu single-shot
- Funkcje: init_timer, add_timer, del_timer (o ile timer zostanie odpalony, to nie ma potrzeby wołania tej funkcji)
- Działanie po wywołaniu init_timer po czasie zaprogramowanym przez expires funkcja function zostanie wywołana z argumentem równym polu data.
- Dużo bardziej “pewne” odmierzanie czasu niż w poprzenim przypadku, ponieważ handler zostanie wywołany przy najbliższej obsługi dolnej połowy.
- Haczyki:
  - Nie wiemy w kontekście jakiego procesu – nie można sięgać do danych w pamięci procesu !!!
  - Synchronizacja => dolne połowy mogą przerwać kod jądra (w szczególności kod sterownika), a timer jest wywoływany z dolnych połów => potrzeba cli()

Funkcja schedule (4)

- W tym momencie zmienna next wskazuje na nowy proces.

```
if (prev != next) {
        struct timer_list timer;

        kstat.context_swtch++;
        if (timeout) {
                init_timer(&timer);
                timer.expires = timeout;
                timer.data = (unsigned long) prev;
                timer.function = process_timeout;
                add_timer(&timer);
        }
        switch_to(prev,next); // przełącz kontekst
        // powrót nastąpi dopiero po ponownym przełączeniu kontekstu
        if (timeout) // Możliwość obudzenia przed wywołaniem timera
                     // Np sygnał
                del_timer(&timer);
 }
```

Jeżeli ustawiliśmy timeout (patrz dygresja) to wykorzystaj timer jądra do wywołania funkcji process_timeout budzącej proces po zaprogramowanym czasie.

- Funkcja przełączająca kontekst jest napisana w assemblerze
- add_timer – pozwala na zaprogramowanie timerów jądra – funkcji

Funkcja process_timeout

- Proces uśpiony nie czekał na żadnej kolejce (wait_queue) – więc wystarczy przełączyć stan na TASK_RUNNING i dodać do kolejki procesów gotowych..

```
inline void wake_up_process(struct task_struct * p)
{
        unsigned long flags;
        save_flags(flags);
        cli();
        p->state = TASK_RUNNING;
        if (!p->next_run)
                add_to_runqueue(p);
        restore_flags(flags);
}

static void process_timeout(unsigned long __data)
{
        struct task_struct * p = (struct task_struct *) __data;
        p->timeout = 0;
        wake_up_process(p);
}
```

- cli() - ponieważ kod funkcji może być przerwany przez przerwanie (lub dolną połowę, o ile nie jest wywołana ona z dolnej połowy), a przerywający kod może operować na kolejce procesów gotowych.

Funkcja add_to_runqueue

- Jest wywoływana zawsze przy dodawaniu procesu do kolejki procesów gotowych (ang. run queue), między innymi przez funkcję rodziny wakeup. Pomijamy kod SMP i kontrolę błędów.

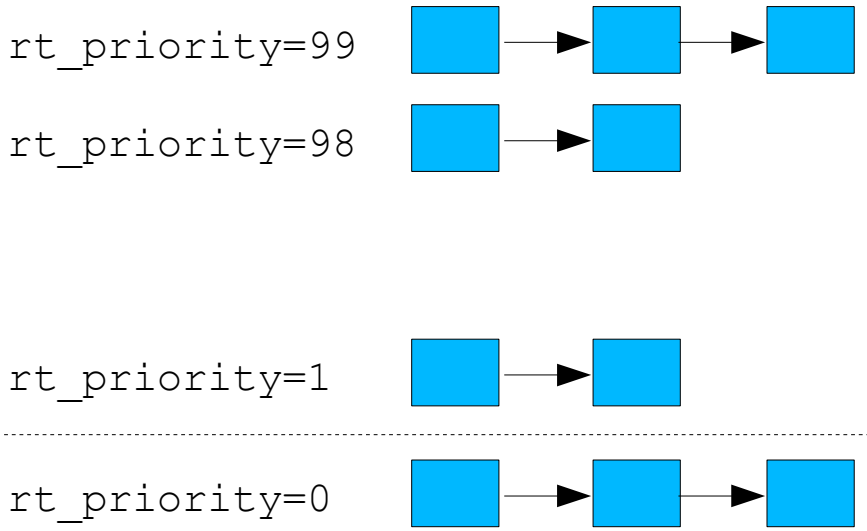
```
static inline void add_to_runqueue(struct task_struct * p)
{
   // Tu robi sie ciekawe
   if (p->policy != SCHED_OTHER || p->counter > current->counter + 3)
      need_resched = 1;
   nr_running++;
   // Dodanie do cyklicznej listy dwukierunkowej (kod szybki ale nieczytelny)
   (p->prev_run = init_task.prev_run)->next_run = p;
   p->next_run = &init_task;
   init_task.prev_run = p;
}
```

- Jeżeli nowo dodany proces jest czasu rzeczywistego, lub ma większy (co najmniej o 40ms) kwant czasu niż proces bieżący to uruchamiany jest planista (i oczywiście wywłaszczy proces bieżący, jeżeli nie jest on czasu rzeczywistego lub ma mniejszy kwant).
- Cel: przyspieszenie reakcji na procesy interakcyjne, spełnienie standardu POSIX dla procesów czasu rzeczywistego.

Cechy algorytmu szeregowania

- Procesy zwykłe mogą zostać zagłodzone przez procesy czasu rzeczywistego.
  - Wysokopriorytetowy proces klasy SCHED_FIFO, który wpadnie w pętlę nieskończoną może zawiesić system.
- Jeżeli brak procesów czasu rzeczywistego, to nie może dojść do zagłodzenia (dlaczego ?)
- Algorytm jest sprawiedliwy i (prawie) nie faworyzuje żadnego z procesów z grupy procesów o tym samym priorytecie.
- Jeżeli proces wyczerpie swój kwant czasu to musi czekać na wyczerpanie kwantu czasu pozostałych procesów.
  - Może to sprawiać problemy w przypadku procesów interaktywnych (ale takie procesy rzadko wyczerpują swój kwant czasu).
- Brak rozróżnienia pomiędzy procesami interaktywnymi i obliczeniowymi.
  - Jeżeli system jest obciążony procesami obliczeniowymi, to utrudnia to pracę procesom interaktywnym.

Kolejka wielopoziomowa



Powyżej tej linii procesy czasu rzeczywistego

- 100 wartości rt_priority symuluje podział kolejki procesów gotowych na 100 poziomów kolejek „koncepcyjnych”, przy czym procesy są wybierane z najwyższego możliwego poziomu.

Wybrane funkcje systemowe związane z szeregowaniem

- int nice(int increment) - zmiana wartości priorytetu statycznego o podaną wartość
  - Wartość ujemna oznacza zwiększenie, może z niej korzystać tylko root.
  - Wartość nice 0-20 jest przeliczana na wartość priority 0-2*DEFPRIORITY
  - Na wartości priorytetu operują również funkcje getpriority/setpriority
- int sched_yield() - oddanie procesora na własne życzenie
  - Proces jest jednocześnie przesyłany na koniec kolejki procesów gotowych.
- int sched_setscheduler(pid_t pid, int policy, struct sched_param *param) - ustawienie klasy szeregowania dla procesu o numerze pid, (pid==0 oznacza bieżący proces).
  - policy może przyjmować wartość SCHED_FIFO, SCHED_RR, SCHED_OTHER.
  - sched_param posiada tylko jedno pole sched_priority. Jest to priorytet czasu rzeczywistego
  - sched_getscheduler pobiera klasę szeregowania