Obsługa kolejki procesów gotowych

Ngo Chi Lang

18 Grudnia 2000

Spis treści

1 Wprowadzenie **3**

2 Struktury danych **3**
- 2.1 Struktura `task_struct` 3
- 2.2 Proces `init_task` 3
- 2.3 Kolejka procesów gotowych 3
- 2.4 Kolejki koncepcyjne 4

3 Algorytmy **5**
- 3.1 Algorytm `add_to_runqueue` 5
- 3.2 Algorytm `del_from_runqueue` 5
- 3.3 Algorytm `move_last_runqueue` 6
- 3.4 Algorytm `move_first_runqueue` 7

4 Dodatkowe informacje **8**

5 Zadania **8**

6 Rozwiązywania zadań **8**

1 Wprowadzenie

Kolejka procesów gotowych jest listą wszystkich procesów, które są gotowe do wykonywania (lub są aktualnie wykonywane). Kolejka ta jest realizowawa jako cykliczna lista dwukierunkowa. Początkowo znajduje się na niej jedynie proces *idle* [1]. Lista ta jest przeglądana przez programu szeregujący *schedule* w poszukiwaniu procesu, któremu zostanie przydzielony procesor.

2 Struktury danych

2.1 Struktura `task_struct`

Każdy proces jest w systemie Linux reprezentowany przez strukturę `task_struct`. Najważniejsze elementy struktury `task_struct` z punktu widzenia obsługi kolejki procesów gotowych to :

- `volatile long state` - stan procesu.
  Procesy znajdujące się w kolejce procesów gotowych mają stan `TASK_RUNNING` (chyba, że właśnie został zmieniony na inny i zaraz będzie przeszeregowanie - `schedule()`). Oznacza to, że proces jest gotowy do wykonywania lub jest w trakcie wykonywania.
- `counter` - ile kwantów czasu procesora pozostało procesowi. Gdy `counter` dojdzie do 0 następuje przeszeregowanie.
- `struct task_struct *next_run, *prev_run`
  dowiązania do następnego i poprzedniedgo procesu gotowego.
- `policy` - tryb szeregowania.
  Procesy zwykłe mają `SCHED_OTHER`, a procesy czasu rzeczywistego mają `SCHED_FIFO` lub `SCHED_RR`.
- `long rt_priority` - numer kolejki koncepcyjnej do której proces należy (0..99)

Bardziej szczegółowy opis struktury `task_struct` można znaleźć w [2].

2.2 Proces `init_task`

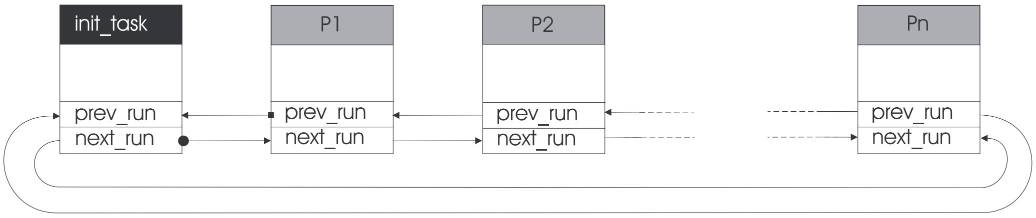
Proces `init_task` (*idle*) jest tworzony podczas startu systemu. Ma on najniższy priorytet w systemie i jest wykonywany tylko wtedy, gdy nie ma innych procesów ubiegających o dostęp do procesora. Proces ten nie robi nic (kręci się w pustej pętli).

2.3 Kolejka procesów gotowych

Jak już wspomniano we wstępie, kolejka procesów gotowych jest dwukierunkową listą cykliczną z wyróżnionym elementem - procesem *idle* (wskazywanym przez zmienną globalną `init_task`). Dowiązania do poprzedniego i następnego elementu listy są zapamiętane w polach `prev_run, next_run` struktury `task_struct` procesu. Procesy, które nie znajdują się w kolejce mają pola `next_run, prev_run` równe `NULL`.

[1] Jest to pierwszy proces utworzony podczas startu systemu

Z kolejką procesów gotowych związana jest zmienna globalna `nr_running`, której wartością jest liczba procesów ubiegających się o przydział procesora - czyli jest to liczba procesów znajdujących się w kolejce procesów gotowych nie licząc procesu *idle*.



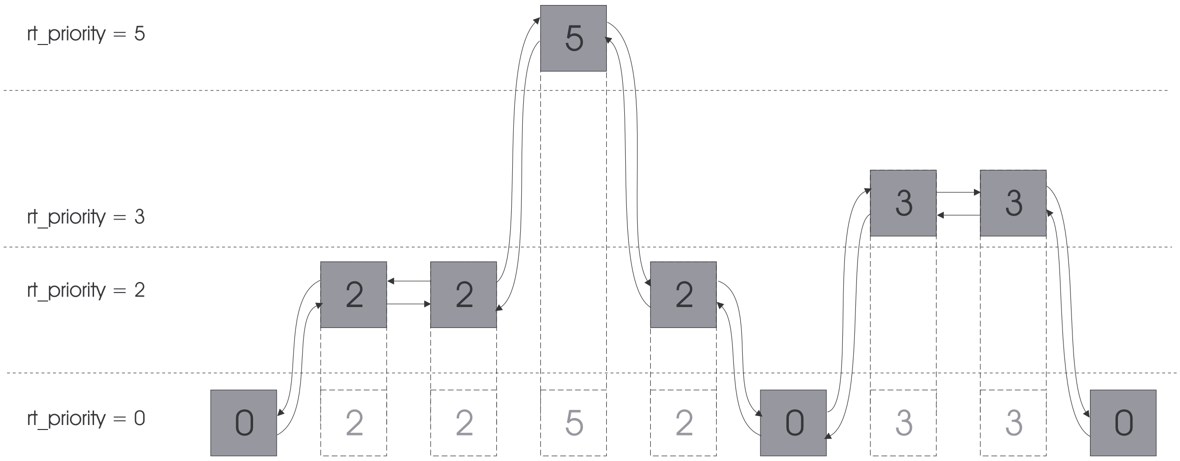
Rysunek 1. Kolejka procesów gotowych

2.4 Kolejki koncepcyjne

W Linuksie wyróżnia się dwie klasy procesów ze względu na ich priorytet wykonywania :

- procesy zwykłe
- procesy czasu rzeczywistego (ang. *real-time*)

Procesy czasu rzeczywistego wyróżniają się tym, że muszą mieć ściśle określony czas reakcji. Dlatego też w rywalizacji o dostęp do procesora mają one priorytet wyższy od wszystkich procesów zwykłych. Ponieważ w systemie fizycznie jest tylko jedna kolejka procesów gotowych wprowadza się tzw. *kolejki koncepcyjne*. W systemie znajduje się 100 kolejek koncepcyjnych ponumerowanych od 0 do 99. Kolejki koncepcyjne są realizowane poprzez nadanie zmiennej `rt_priority` struktury `task_struct` wartości z zakresu 0..99 odpowiadającej numerowi kolejki. Procesy czasu rzeczywistego są pogrupowane w kolejkach koncepcyjnych ponumerowanych od 1 do 99. Procesy zwykłe znajdują się w kolejce koncepcyjnej 0 (`rt_priority=0`).



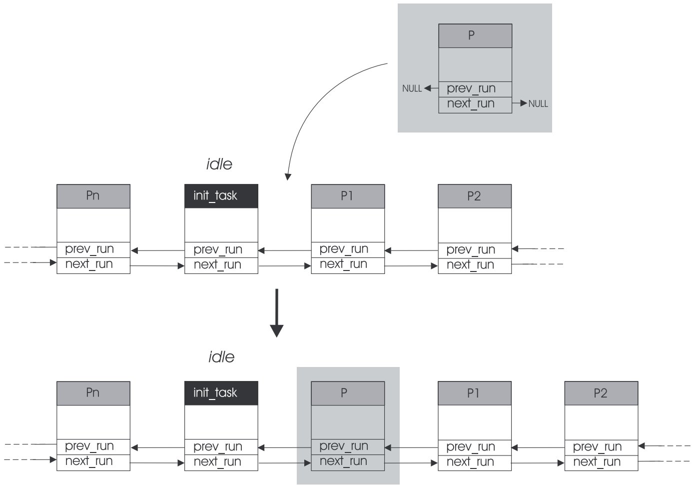
Rysunek 2. Kolejki koncepcyjne

3 Algorytmy

Do obsługi kolejki procesów gotowych dostępne są proste funkcje realizujące operacje dodawania, usuwania, przesunięcia elementów kolejki. Funkcje te nie robią nic poza manipulacją wskaźnikami `next_run, prev_run` odpowiednich elementów kolejki oraz modyfikacją globalnej zmiennej `nr_running`.

3.1 Algorytm `add_to_runqueue`

`void add_to_runqueue (struct task\_struct * p)`
Funkcja dodaje proces wskazywany przez `p` do kolejki oraz zwiększa licznik procesów gotowych `nr_running`.
Warto tu zauważyć, że proces jest wstawiany na początek kolejki - tzn. bezpośrednio za procesem idle. Umożliwia to przyjmowanie przerwań (a więc również możliwość dodania nowych procesów do kolejki) w czasie, gdy funkcja szeregująca `schedule()` wybiera proces który zaraz dostanie procesor. Wstawienie nowych procesów na początek kolejki nie zakłóca porządku aktualnie przeglądanego fragmentu kolejki, przez co nie prowadzi do błędnej sytuacji.



Rysunek 3. Dodawanie do kolejki procesów gotowych - `add_to_runqueue`

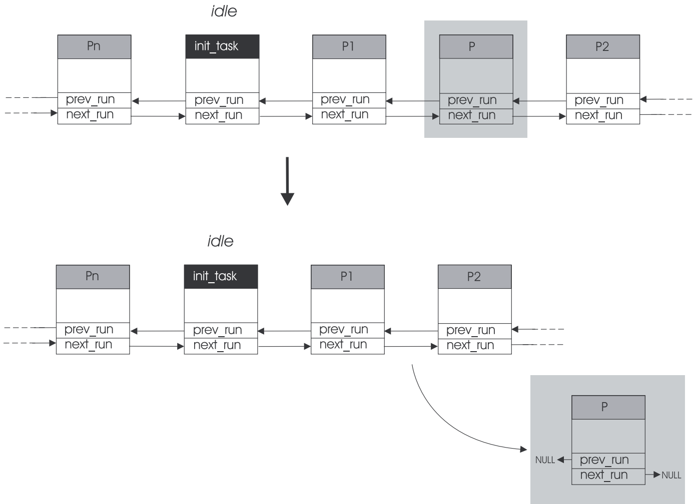
Gdzie jest wywołany :

- `wake_up_process(struct task\_struct *p)`
  Funkcja `wake_up_process`, dla danego procesu `p` sprawdza czy on już znajduje się na kolejce procesów gotowych; jeśli nie to `p` zostanie dodany do kolejki - `add_to_runqueue`.

3.2 Algorytm `del_from_runqueue`

`void del_from_runqueue (struct task_struct * p)`
Funkcja usuwa dany proces `p` z kolejki procesów gotowych i zmniejsza licznik (`nr_running`).
Gdzie jest używany :

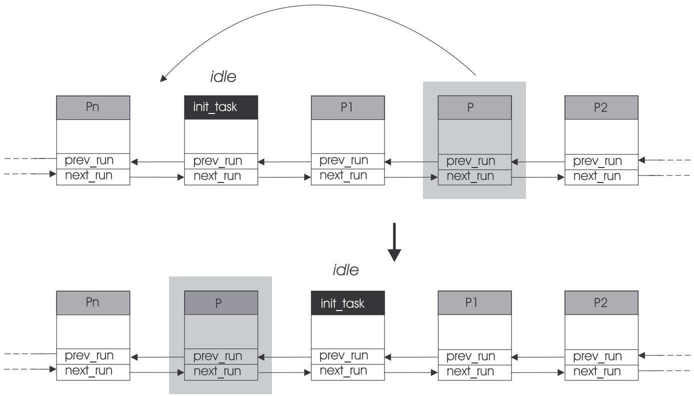
- `schedule()`
  Jeśli program szeregujący *schedule* przeglądając kolejkę procesów gotowych napotka na proces, którego status jest różny od `TASK_RUNNING` albo status jest równy `TASK_INTERRUPTIBLE` a nie ma przychodzącego do niego sygnału, to proces ten zostanie usunięty z kolejki procesów gotowych - `del_from_runqueue`.



Rysunek 4. Usuwanie z kolejki procesów gotowych - `del_from_runqueue`

3.3 Algorytm `move_last_runqueue`

`void move_last_runqueue (struct task_struct * p)`
Funkcja przesuwa dany proces `p` na koniec kolejki procesów gotowych, bezpośrednio przed procesem *idle*.



Rysunek 5. Przesuwanie na koniec kolejki procesów gotowych - `move_last_runqueue`

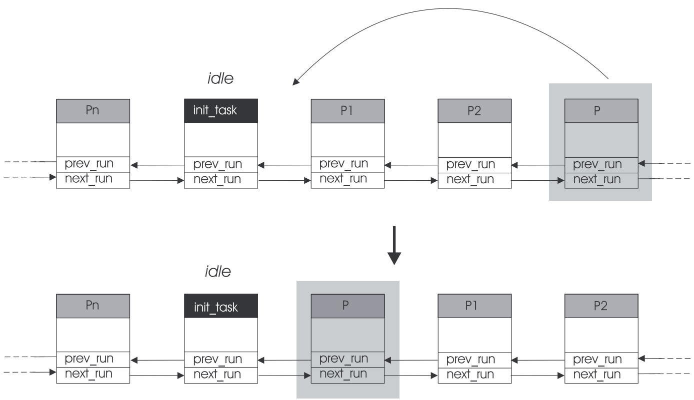
Gdzie jest używany :

- `void schedule(void)`
  Program szeregujący *schedule* przeglądając kolejkę procesów gotowych może napotkać na proces, któremu skończył się kwanty czasu (`counter=0`) oraz jest on szeregowany w trybie *Round-Robin* (`policy=SCHED_RR`) to przyznany mu zostanie nowy kwant i przzesuwany jest na koniec kolejki - `move_last_runqueue`.

- `int sys_sched_yield(void)`
  Proces czasu rzeczywistego może sam się wywłaszczać, zwalniając dostęp do procesora. Używa on do tego celu funkcję `sys_sched_yield()`, która wywłaszcza bieżący proces, jednocześnie przesuwając go na koniec kolejki procesów gotowych - `move_last_runqueue`.

3.4 Algorytm `move_first_runqueue`

`void move_first_runqueue (struct task_struct * p)`
Funkcja przesuwa dany proces `p` na początek kolejki procesów gotowych, bezpośrednio za procesem *idle*.



Rysunek 6. Przesuwanie na początek kolejki procesów gotowych - `move_first_runqueue`

Gdzie jest używany :

- `int setscheduler(pid_t pid, int policy, struct sched_param *param)`
  Funkcja `setscheduler` zmienia sposób szeregowania danego procesu. Służy ona do tworzenia procesów czasu rzeczywistego. Jako parametr podaje się `pid` procesu (dla `pid=0`, zmienia sposobu szeregowania procesu wywołującego tą funkcję), metoda szeregowania (`policy`), oraz numer kolejki koncepcyjnej (`param.sched_priority`) na której ma być umieszczony. Po zmianie odpowiednich elementów struktury `task_struct` danego procesu, wywołana jest funkcja `move_first_runqueue` umieszczając nowo "utworzonego" procesu na początek kolejki procesów gotowych.

4 Dodatkowe informacje

Bardziej szczegółowe informacje o funkcjach `schedule, wake_up_process, sys_sched_yield, setscheduler` można znaleźć w [2].
Wszystkie omawiane powyżej funkcje i struktury są zdefiniowane w plikach `kernel/sched.c, include/linux/sched.h` :

- `kernel/sched.c`
  funkcje `add_to_runqueue, del_from_runqueue, move_last_runqueue, move_first_runqueue, wake_up_process, schedule, sys_sched_yield, setscheduler`, zmienna `nr_running`
- `include/linux/sched.h`
  struktura `task_struct`, definicja procesu `INIT_TASK` (*idle*).

Do przeglądania plików źródłowych bardzo pomocna jest strona
`http://lxr.linux.no/`

5 Zadania

1. Czy w trakcie pracy systemu kolejka procesów gotowych może być pusta ?
2. Jak można szybko sprawdzić czy proces znajduje się w kolejce procesów gotowych ?
3. Załóżmy, że do implementacji kolejki procesów gotowych zamiast listy dwukierunkowej używa się kolejkę priorytetową realizowaną na drzewie zrównoważonym. Wtedy w algorytmie szeregującym [2] zamiast przeszukiwać całej listy w aby znaleźć proces o najwyższym priorytecie bierze się po prostu wierchołek drzewa. Jak opisana zmiana według ciebie wpłynie na wydajność systemu ?

6 Rozwiązywania zadań

1. Nie. Podczas startu systemu kolejka procesów gotowych jest "zainicjonawa" procesem `init_task` (proces *idle*). Podczas pracy systemu proces *idle* nigdy nie jest usuwany z kolejki i dlatego kolejka nigdy nie jest pusta. Gdy nie ma procesów chętnych do korzystania z procesora to zmienna `nr_running` ma wartość zero, ale w kolejce procesów gotowych znajduje się proces *idle*, więc jemu zostanie przydzielony procesor.
2. Łatwo można stwierdzić czy proces znajduje się w kolejce procesów gotowych sprawdzając czy w jego strukturze `task_struct`, pole `next_run` (lub `prev_run`) jest różne od `NULL`.
3. Autorowi pracy wydaje się, że opisana zmiana raczej nie poprawiłaby wydajność systemu bo uzyskane korzyści w postaci przyspieszenia wyboru procesu nie są współmierne do ponoszonych kosztów operacji dodawania, przesuwania proces do kolejki. Wprowadzenie drzewa znacznie skomplikowałoby też kod jądra oraz wymagało więcej pamięci do reprezentowania tej struktury.

[2] Więcej szczegółów można znaleźć w [2]

Bibliografia

[1] Źródło Linux'a, 2.2.17

[2] *Linux Podręcznik*, Rozdział 2, Zarządzanie procesami

[3] *Linux Prezentacje*, Procesy

[4] David A Rusling : *The Linux Kernel*

[5] Michael K. Johnson : *Linux Kernel Hacker's Guide*

[6] Maurice J. Bach : *Budowa systemu operacyjnego Unix*, WNT 1995