# Temat 12
# Szeregowanie procesów - algorytm `schedule()`

Rafał Łyżwa
Marek Misiowiec

20 grudnia 2000

# Spis treści

# Rozdział 1

# Szeregowanie procesów

## 1.1 Wprowadzenie

Szeregowanie procesów jest podstawową operacją umożliwiającą współbieżne wykonywanie zadań w każdym nowoczesnym, wielozadaniowym systemie operacyjnym. Algorytm szeregowania procesów, wraz z pakietem funkcji wspierających operacje wyboru i przełączania zadań[1], określany mianem **Schedulera**[2], stanowi krytyczną część podsystemu zarządzania procesami w systemie Linux. Od jego jakości bezpośrednio zależą: szybkość, wydajność i współbieżność wszelkich czynności wykonywanych przez system operacyjny, a tym samym przez użytkownika.

## 1.2 Cechy dobrego algorytmu szeregowania

Algorytm szeregowania dotyczy wszystkich stworzonych procesów a jego implementacja: funkcja `schedule()`, jest wywoływana z bardzo dużą częstotliwością. Oba wspomniane fakty określają warunki, które spełnia „dobry" algorytm szeregowania procesów:

– pochłania stosunkowo mało pamięci operacyjnej

– jest bardzo szybki; jego wykonanie jest względnie krótkie w stosunku do długości pracy pojedynczego procesu

– zapewnia dostatecznie krótki czas reakcji na zmiany stanu procesów

– gwarantuje równowagę pomiędzy procesami o wyższych i niższych priorytetach

– unika zagłodzenia, blokady, itp.

– przenosi się na systemy wieloprocesorowe

[1]W dalszej części pracy pojęcia **proces** oraz **zadanie** będą używane wymiennie. Mamy nadzieję, że nie wprowadzi to żadnej niejednoznaczności.

[2]Zdecydowaliśmy się nie korzystać z żadnych rodzimych tłumaczeń tego słowa (np. *planista*). Wersja oryginalna kojarzy się natychmiast z podprogramem szeregowania zadań. Jej tłumaczenie może zbić z tropu niedoświadczonego czytelnika.

[3]W odniesieniu do szeregowania, określeń **polityka** oraz **strategia** będziemy używać zamiennie.

## 1.3 Polityka szeregowania

Zbiór reguł pozwalających ustalić jak, kiedy i komu przydzielać czas procesora nazywany jest **polityką szeregowania**[3] *(ang. scheduling policy)*. Scheduler systemu Linux działa generalnie zgodnie z zasadą **podziału czasu** *(ang. time-sharing)*, co oznacza, iż każdy proces otrzymuje procesor jedynie na pewien okres, określany mianem **kwantu czasu**. W jądrze 2.2.17 wspomnianej regule nie podlegają jedynie procesy szeregowane w trybie `SCHED_FIFO`, co zostanie omówione nieco dalej.
W szczególności podział czasu sprawia, iż proces, w większości przypadków nie wykonuje całego zadania w jednym, spójnym przedziale czasu. Jego działanie może zostać przerwane i wznowione po chwili w niezmienionym stanie – nie wymaga to aplikowania żadnego dodatkowego kodu dla wykonującego się zadania. Niewidoczność pracy Schedulera jest dla procesów korzyścią, uzyskaną dzięki obsłudze przerwań i operacjom zmiany kontekstu.
Polityka szeregowania odnosi się także do pojęcia prioryteu procesu oraz klasyfikuje procesy posiadające wspólne, wyróżniające cechy. Priorytet, a także dynamika jego zmian, jest kluczowym czynnikem zapewniającym współbieżne i sprawiedliwe działanie procesów.

## 1.4 Typy procesów

### 1.4.1 Wykorzystanie procesora

Procesy, ze względu na częstotliwość wykorzystania procesora, można zasadniczo podzielić na dwie grupy [4]:

1. Procesy silnie wykorzystujące procesor, tj. te, których efektywne wykonanie jest bezpośrenio zależne od ciągłego przydziału procesora.
   Do grupy tej zaliczają się np. procesy wykonujące obliczenia naukowe, systemy bazodanowe, kompilatory, itp. W skrócie będziemy je nazywać **procesami obliczeniowymi** *(ang. CPU-bound)*.

2. Procesy silnie wykorzystujące urządzenia wejścia/wyjścia.
   Większość czasu ich życia stanowi oczekiwanie na operacje zapisu/odczytu z urządzeń zewnętrznych. Będziemy określać je nieco sztucznym mianem **procesów wejścia/wyjścia** *(ang. IO-bound)*. Przykładem są: interpretery poleceń, edytory tekstu, spooler drukarki, itp.

### 1.4.2 Czas reakcji

Ze względu na czas reakcji na określone zdarzenia, procesy można podzielić na:

– **interakcyjne**
  Istotą ich działania jest interakcja z użytkownikiem. W skład tej grupy wchodzą między innymi: edytory tekstu, powłoki, programy graficzne. Procesy interaktywne rzadko korzystają z procesora, ale jeżeli zajdzie taka potrzeba – czyli gdy użytkownik naciśnie klawisz, przycisk myszki, itp. – muszą zostać względnie szybko obudzone. Warunkiem dostatecznym jest zapewnienie czasu

[4] Określenia **typ procesu** będziemy dalej używać w kontekście przynależności do grupy procesów o podobnym charakterze. Nie ma ono przełożenia na pojęcie **klasy procesów**, które rezerwujemy na oznaczenie grupy procesów szeregowanych zgodnie z jednakową strategią. Klasy procesów zostały *implicite* określone przez twórców jądra systemu.

reakcji w obrębie 50–150 milisekund. Przekroczenie jego górnej granicy jest postrzegane przez użytkownika jako wyraźne opóźnienie pracy. Wariancja czasów reakcji musi być ograniczona.

– **wsadowe**
  Nie wymagają współpracy z użytkownikiem, pracują w tle. Takie procesy posiadają zwykle niższy priorytet, są tym samym osłabiane przez Scheduler. Przykładowymi zadaniami wsadowymi są kompilatory, mechanizmy baz danych, itd.

– **czasu rzeczywistego**
  Do grupy tej zalicza się między innymi: programy służące do obróbki dźwięku, grafiki, obsługujące automaty, odbierające i przetwarzające dane z czujników elektronicznych. Warunkiem ich poprawnego działania jest bezwzględnie krótki czas reakcji na zaistniałe zdarzenie, a co za tym idzie wymagają specjalnego traktowania przez funkcję szeregującą. Wyodrębnienie takich zadań wiązało się z wprowadzeniem w jądrze 2.0 specjalnej klasy – **procesów czasu rzeczywistego**. Klasa ta uwzględnia dwie różne metody szeregowania zadań, szczegółowo opisane w dalszej części pracy (wszystkie pozostałe procesy, poza wyróżnionym nieusuwalnym procesem `Idle`, zaliczają się do klasy procesów zwykłych). Warunkiem koniecznym jaki muszą spełniać obie strategie jest minimalna wariancja czasów reakcji na zdarzenie. Ponadto nie mogą być blokowane przez procesy o niższym priorytecie.

### 1.4.3 Niezależność wprowadzonych typów.

Wprowadzone powyżej typy procesów przenikają się wzajemnie. Oznacza to, że nie istnieje bezwzględne przyporządkowanie pomiędzy procesami wsadowymi a obliczeniowymi oraz procesami interakcyjnymi a procesami wejścia/wyjścia. Procesy wsadowe mogą być procesami obliczeniowymi, jak kompilator, ale również procesami wejścia/wyjścia, tak jak serwer bazodanowy. Algorytm szeregowania nie odróżnia *explicite* procesów interaktywnych od wsadowych. Różnice w traktowaniu procesów są wynikiem przyjętej specyfiki szeregowania, a dokładnie wiążą się z utożsamieniem priorytetu dynamicznego procesu z czasem procesora pozostającym do wykorzystania. Ponadto użytkownik-progamista może poprzez funkcje systemowe bezpośrednio wpływać na wielkość przyznanych priorytetów. Scheduler wyróżnia jedynie procesy czasu rzeczywistego, przydzielając im odpowiednio wysoki priorytet statyczny (wyjaśnienia tych pojęć pojawią się w dalszej części pracy).

> **Przykład:**
> W systemie działają dwa procesy użytkownika: edytor tekstu (ET) oraz kompilator (K). K aktualnie korzysta z procesora. ET jako proces interakcyjny ma wyższy priorytet dynamiczny – zużywa mniej czasu procesora – ale przez większość czasu jest uśpiony. W chwilę po naciśnięciu klawisza przez użytkownika, ET jest budzony i w wyniku działania programu szeregującego K zostaje zatrzymany, a ET może zareagować na ruch użytkownika. Po wykonaniu przewidzianych operacji ET ponownie zasypia, a do działania przywrócony zostaje K.

Powyższy przykład prezentuje zasadę współdziałania procesów interakcyjnych i wsadowych. Widać, że dużą wagę przykłada się do wielkości priorytetu dynamicznego, a zatem i długości kwantu czasu. Pozostaje więc zapytać

## 1.5 Jak długi pownien być kwant czasu?

Jeżeli kwant czasu jest zbyt krótki, operacja przełączenia kontekstu staje się zbyt kosztowna czasowo.

> **Przykład:**
> Załóżmy, że kwant czasu jest porónywalnie krótki do czasu potrzebnego na zmianę kontekstu procesów. Wtedy statystycznie około 50 procent czasu procesora zużywane jest na czynności administracyjne, nie związane z potrzebami użytkownika.

Jeżeli natomiast kwant czasu jest zbyt długi, użytkownik traci poczucie współbieżności.

> **Przykład:**
> Załóżmy, że w systemie aktualnie wykonują się jedynie procesy zwykłe, tak że szeregowanie ich dostępu do procesora przypomina obsługę kolejki `FIFO`, oraz kwant czasu jest relatywnie wielki. Wtedy czas oczekiwania na kolejny przydział procesora wynosi około:
>
> `liczba_działających_procesów * kwant_czasu,`
>
> co przy długim kwancie dostatecznie oddala od siebie kolejne okresy pracy procesu. Ginie poczucie współbieżności.

Źle dobraną długość kwantu czasu możemy odczuć także w momencie równoczesnego startu dwóch procesów.

> **Przykład:**
> Załóżmy, że jednocześnie rozpoczynają pracę dwa procesy: interakcyjny (I) oraz wsadowy (W). Jeżeli procesor zostanie przydzielony W, a użytkownik zgłosi akces programowi I, to przy długim kwancie czasu odczuje opóźnienie pomiędzy wydaniem polecenia I oraz reakcją procesu.

Powyższe, nieco przerysowane przykłady wskazują na rolę jaką może odegrać rozsądny dobór długości kwantu. Twórcy Linuksa wyznają zasadę: wybierz możliwie najdłuższy okres czasu, tak by nie zachwiać szybkości reakcji systemu. Efektem ich prac eksperymentalnych, jest ustalenie kwantu czasu, w jądrze 2.2.17, na 10 milisekund.

## 1.6 Klasy procesów w Linuksie

W systemie Linux są zaimplementowane następujące klasy procesów

- **idle**
  W ten sposób nazwywa się proces określany inaczej także jako `init_task`. Jest to proces o numerze 0, tworzony bez udziału funkcji `fork()`, za to inicjalizowany wartościami zdefiniowanymi w makrodefinicji `INIT_TASK`. Proces ten wykonuje nieskończoną pętlę i cały czas znajduje się w stanie `TASK_RUNNING`. Ma najniższy priorytet w systemie i otrzymuje procesor tylko wtedy, kiedy żaden inny proces go nie potrzebuje. Z chwilą nadejścia dowolnego procesu, `idle` traci natychmiast procesor.

Czas procesora poświęcony na wykonywanie tego procesu pozwala ocenić natężenie jego użycia. Im jest on dłuższy, tym obciążenie procesora mniejsze.

Proces ten oznacza także początek kolejki procesów gotowych (patrz rozdział „Kolejka procesów gotowych i kolejki koncepcyjne").

- **procesy zwykłe**
  Do tej klasy zalicza się znakomita większość procesów. System Linux pozwala na szeregowanie tych procesów jedynie według jednej polityki szeregowania (patrz rozdział „Polityki szeregowania procesów w Linuksie"), ale jest ona na tyle elastyczna, że umożliwia znaczne zróżnicowanie poziomów uprzywilejowania procesów tej klasy. Co ważne, ich priorytety są dynamiczne, tzn. zmieniają się wraz z długością używania procesora przez proces. Dzięki temu system zwiększa sprawiedliwość i zapobiega zagłodzeniu. Dokładne omówienie tych zagadnień znajduje się w kolejnych rozdziałach.

- **procesy czasu rzeczywistego**
  Jak wspomniano już wcześniej procesy czasu rzeczywistego to procesy, które wymagają bardzo szybkiej, natychmiastowej reakcji systemu. Jeżeli ona nie nastąpi może dojść do, w najlepszym przypadku, niepożądanego zwolnienia ich pracy (np. w programach odtwarzających obraz wideo). W najgorszym razie może to spowodać zagrożenie życia ludzkiego (np. w systemach chłodzenia elektrowni atomowych).

  Choć Linux nie jest systemem dedykowanym do tego typu zastosowań, zawiera implementację procesów czasu rzeczywistego. Procesy te mają zawsze pierwszeństwo przed procesami zwykłymi, tzn. w chwili, gdy proces czasu rzeczywistego będzie potrzebował procesora, a będzie on w posiadaniu procesu zwykłego, ten ostatni zostanie natychmiast wywłaszczony. Oczywiście, tak duże możliwości procesów czasu rzeczywistego mogą w łatwy sposób doprowadzić do destabilizacji systemu (np. poprzez stworzenie takiego procesu wykonującego nieskończoną pętlę). Dlatego procesy czasu rzeczywistego może tworzyć jedynie administrator systemu.

  Podobnie jak w przypadku procesów zwykłych, także w obrębie tej klasy występuje podział na procesy bardziej lub mniej uprzywilejowane. Jednak w odróżnieniu od tamtej klasy mają priorytet statyczny, tzn. nie zmieniający się z powodu używania procesora przez proces. Dokładne omówienie tego zagadnienia znajduje się w dalszej części pracy.

## 1.7 Zarys algorytmu szeregującego

Algorytm szeregujący dzieli czas procesora na **epoki** *(ang. epoch)*. Na początku epoki dla każdego procesu obliczany jest kwant czasu. Oznacza on maksymalny czas procesora, jaki w danej epoce może wykorzystać proces. Kiedy proces wykorzysta cały kwant, zostaje wywłaszczony. Każdy proces może być przydzielany do procesora wielokrotnie podczas jednej epoki, do czasu, gdy nie skończy się jego kwant. Na przykład, jeżeli proces zostanie zawieszony w oczekiwaniu na jakiś zasób, zachowuje pewną część swojego kwantu i może zostać wybrany do wykonywania jeszcze w tej samej epoce. Epoka kończy się, gdy wszystkie procesy gotowe do wykonania wykorzystały swój kwant czasu. Wówczas algorytm szeregujący oblicza ponownie kwant dla każdego procesu - rozpoczyna się nowa epoka.

Każdy proces ma **bazowy kwant czasu** *(ang. base time quantum)* (zwany także **priorytetem bazowym** *(ang. base priority)* - więcej na ten temat przy opisie pola *priority* struktury *task_struct*), czyli kwant czasu przyznawany przez algorytm szeregujący procesowi, kiedy wykorzystał on swój kwant w poprzedniej epoce. Więcej na ten temat można znaleźć w opisie struktur danych wykorzystwanych przy szeregowaniu.

Aby wybrać proces, który uzyska procesor, algorytm szeregujący sprawdza priorytety procesów znajdujących się w kolejce procesów gotowych. W systemie Linux istnieją dwa rodzaje priorytetów:

- **priorytet statyczny** *(ang. static priority)*
  Dotyczy procesów czasu rzeczywistego. Nie zmienia się w czasie. Więcej na ten temat przy opisie pola *rt_priority* struktury *task_struct*.

- **priorytet dynamiczny** *(ang. dynamic priority)*
  Dotyczy procesów zwykłych. Pomijając szczegóły (patrz rozdział „Działanie funkcji *goodness()*") można powiedzieć, że jest to suma priorytetu bazowego i liczby tyknięć zegara pozostałych do zakończenia kwantu czasu procesu w danej epoce.

## 1.8 Struktury danych

- stała *DEF_PRIORITY*
  Domyślna wartość priorytetu bazowego, określona wzorem:

  ```
  #define DEF_PRIORITY (20 * HZ / 100)
  ```

  gdzie `HZ` jest stałą zależną od architektury komputera, określającą częstotliwość przerwań zegara. Dla architektury Intela i386 wynosi ona 100.

- zmienna *current*
  Wskaźnik do procesu, który posiada procesor.

- struktura *task_struct*

### 1.8.1 Pola struktury task_struct istotne przy szeregowaniu

- `state`
  Stan, w jakim znajduje się proces. Możliwe wartości to:

  - TASK_RUNNING
  - TASK_INTERRUPTIBLE
  - TASK_UNINTERRUPTIBLE
  - TASK_STOPPED
  - TASK_ZOMBIE
  - TASK_SWAPPING

  Autorzy tego opracowania zakładają, że stany procesów zostały opisane w poprzednich tematach, a czytelnik posiada wiedzę tam zawartą.

- `need_resched`
Jeśli ta flaga jest ustawiona, oznacza to, że potrzebne jest wykonanie przeszeregowania procesów i system wywołuje funkcję `schedule()`. Użycie tej flagi pozwala na odłożenie szeregowania na później, jeżeli w danej w chwili mogłoby ono trwać zbyt długo. W szczególności dotyczy to procedury obsługi przerwania zegarowego.

  Flaga sprawdzana przez procedurę powrotu z przerwania (`ret_from_intr()`), a także przez niektóre sterowniki urządzeń. (więcej na temat użycia tej flagi - patrz rozdział *Wywołanie funkcji schedule()* ).

- `policy`
Polityka szeregowania. Opis polityk szeregowania zaimplementowanych w jądrze Linuxa 2.2.17 znajduje się w rozdziale „Polityki szeregowania procesów".

- `priority`
**Priorytet bazowy** *(ang. base priority)* procesu - ustawiany m.in. przez funkcję `nice()`. Określa długość kwantu czasu przyznawanego procesowi. Wewnętrznie w systemie wartość ta należy do przedziału 0..2*DEF_PRIORITY. Z zewnątrz (tzn. przy pomocy funkcji systemowych) widziana jest jako wartość z zakresu od -20 (najwyższy priorytet) do 20 (najższy priorytet). Więcej na ten temat znajduje się w opisie funkcji systemowych związanych z szeregowaniem.

- `counter`
Liczba tyknięć zegara, które pozostały procesowi do zakończenia jego kwantu. Wartość tego pola należy do zakresu 0..4*DEF_PRIORITY, czyli górna granica zakresu jest dwa razy większa niż w polu `priority`. Skąd taki stosunek wartości - patrz rozdział „Działanie funkcji szeregującej".

  Wywołanie funkcji `update_process_times()` (która jest tzw. „dolną połówką" przerwania zegarowego) zmniejsza o jeden zawartość tego pola.

  Pole to jest inicjalizowane podczas tworzenia procesu przez funkcję `do_fork()`:

```
current->counter = current->counter / 2;
p->counter = current->counter;
```

  gdzie `current` oznacza ojca, zaś `p` syna. Jak widzimy, ojciec dzieli się z synem pozostałym mu kwantem czasu. Dlaczego tak się dzieje? Mechanizm ten zapobiega przed zdobywaniem przez użytkownika dowolnej ilości czasu procesora poprzez tworzenie wielu dzieci.

  Warto zapamiętać, że przypisanie na pole `counter` odbywa się tylko w dwóch miejscach:

  - podczas tworzenia procesu (patrz powyżej)
  - przy przeliczaniu priorytetów na początku nowej epoki (patrz rozdział „Działanie funkcji szeregującej".

  Dodatkowo, dla procesów czasu rzeczywistego szeregowanych wg polityki SCHED_RR (patrz rozdział „Polityki szeregowania procesów w Linuksie") po wyczerpaniu kwantu czasu (wyzerowaniu pola `counter`) przypisywana jest wartość pola `priority`.

- `rt_priority`
**Statyczny priorytet** *(ang. static priority)* procesów czasu rzeczywistego. Może przyjmować

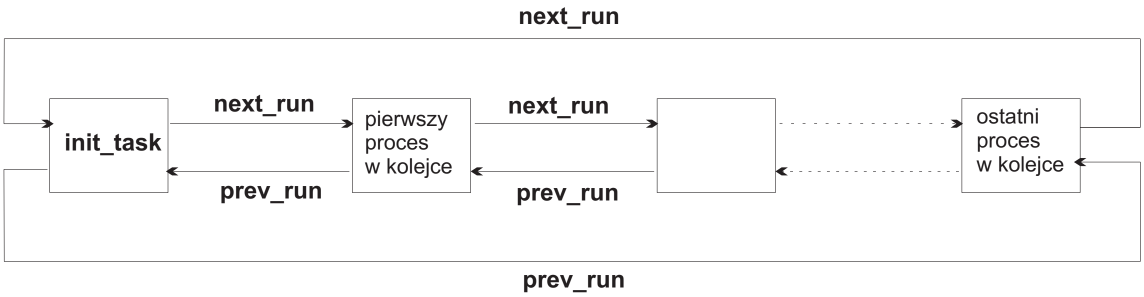
wartości z zakresu od 0 (proces zwykły) do 99 (najwyższy priorytet). Określa numer kolejki koncepcyjnej, w której znajduje się proces (patrz rozdział „Kolejka procesów gotowych i kolejki koncepcyjne").

- `next_run`, `prev_run`
  Wskaźnik do, odpowiednio, następnego i poprzedniego procesu w kolejce procesów gotowych. Więcej na ten temat w rozdziale „Kolejka procesów gotowych i kolejki koncepcyjne".

## 1.9 Kolejka procesów gotowych i kolejki koncepcyjne

Procesy gotowe (w stanie `TASK_RUNNING`), oczekujące na procesor znajdują się w stukturze zwanej **kolejką procesów gotowych** *(ang. runqueue)*. Jest ona implementowana jako kolejka cykliczna, a do oznaczenia jej początku (a więc i końca) służy proces `init_task`. Nowe procesy wstawiane są do kolejki za tym procesem, zaś jako ostatni element kolejki uznaje się proces, którego następnikiem jest właśnie `init_task` (patrz rysunek).

Rysunek 1.1: Kolejka procesów gotowych




O ile kolejka procesów gotowych zawiera wszystkie procesy gotowe do uruchomienia, o tyle **kolejki koncepcyjne** składają się z procesów o tym samym statycznym priorytecie. Kolejki koncepcyjne są pojęciem abstrakcyjnym, które ułatwia omawianie i zrozumienie algorytmu szeregującego. Poprzez wyodrębnienie różnych poziomów „uprzywilejowania" (tzn. procesów z różnymi priorytetami statycznymi) uwidacznia się sposób działania tegoż algorytmu. Procesy z kolejek o wyższych numerach mają pierwszeństwo przed procesami z kolejek o numerach niższych. W szczególności procesy zwykłe, znajdujące się w kolejce o numerze 0, zawsze ustępują procesom czasu rzeczywistego, gdyż one należą do kolejek o numerach większych od 0.

## 1.10 Polityki szeregowania procesów w Linuksie

Jądro Linuxa w wersji 2.2.17 implementuje trzy polityki szeregowania: SCHED_FIFO, SCHED_RR i SCHED_OTHER. Dwie pierwsze są wykorzystywanie jedynie przez procesy czasu rzeczywistego, ostatnia zaś przez procesy zwykłe.

- SCHED_FIFO
W tej polityce proces, który otrzyma procesor może utracić go tylko w wypadku, gdy:
  - zgłosi żądanie wejścia-wyjścia (jest wtedy wyrzucany z kolejki procesów gotowych)
  - proces z kolejki koncepcyjnej o wyższym numerze przejdzie do stanu TASK_RUNNING
  - dobrowolnie odda procesor używając funkcji `sched_yield()`
  - zakończy działanie

  Jak łatwo zauważyć polityka ta jest niesprawiedliwa dla procesów z tej samej kolejki koncepcyjnej - teoretycznie proces, który zdobył procesor może trzymać go dowolnie długo. Oczywiście w praktyce procesy dosyć często zgłaszają żądania wejścia-wyjścia, a także dobrowolnie rezygnują z procesora.

- SCHED_RR
Po przyznaniu procesora procesowi, odmierzany jest kwant czasu. Jak już wspomniano, jego długość określa pole `priority` struktury `task_struct` procesu. Proces może utracić procesor w takich samych przypadkach jak przy polityce SCHED_FIFO, ale dodatkowo także kiedy skończy się jego kwant czasu. Po jego upływie proces jest wyrzucany na koniec kolejki procesów gotowych i przyznaje mu się nowy kwant.

  Polityka jest sprawiedliwa dla procesów z tej samej kolejki koncepcyjnej, gdyż żaden proces nie może nieskończenie długo trzymać procesora.

  **Przykład:**
  W systemie działają 3 procesy czasu rzeczywistego należące do tej samej kolejki koncepcyjnej: jeden szeregowany wg polityki SCHED_FIFO (nazwijmy go **PF**) i dwa szeregowane wg polityki SCHED_RR (nazwijmy je **PR1** i **PR2**). Załóżmy, że procesy w stan TASK_RUNNING przechodzą w kolejności **PR1**, **PF**, **PR2**. Po zakończeniu się kwantu czasu procesu **PR1** zostanie on wywłaszczony (zakładając, że wcześniej nie zgłosił żądania wejścia-wyjścia) i przesunięty na koniec kolejki procesów gotowych. Wtedy procesor przejmie proces **PF** i będzie go trzymał dowolnie długo, grożąc zagłodzeniem procesów **PR1** i **PR2**, jeżeli nie wystąpi z żądaniem wejścia-wyjścia lub dobrowolnie odda procesor. Przykład ten pokazuje ukryte „uprzywilejowanie” procesów szeregowanych zgodnie z polityką SCHED_FIFO względem szeregowanych wg polityki SCHED_RR.

- SCHED_OTHER
Polityka ta jest podobodobna do przedstawionej powyżej polityki SCHED_RR. Istotna różnica polega na tym, że priorytet procesu jest dynamiczny, tzn. zmienia się w czasie posiadania procesora przez proces.

  Na początku epoki procesowi przyznawany jest kwant czasu równy wartości pola `priority` odpowiadającej mu struktury `task_struct`. Jak już wspomniano w poprzednich rozdziałach (będzie o tym jeszcze mowa przy funkcji `goodness()`), dynamiczny priorytet jest sumą pola `priority` i czasu (mierzonego w tyknięciach zegara), jaki pozostał procesowi z jego kwantu. Zatem priorytet dynamiczny procesu posiadającego procesor zmniejsza się z każdym tyknięciem zegara.

  Podsumowując, zgodnie z polityką SCHED_OTHER proces może utracić procesor, gdy:

– zgłosi żądanie wejścia-wyjścia
– dobrowolnie odda procesor używając funkcji `sched_yield()`
– zgłosi się proces czasu rzeczywistego
– zgłosi się proces zwykły o wyższym priorytecie dynamicznym
– zakończy działanie

## 1.11 Wywołanie funkcji schedule()

Funkcja `schedule()` zajmuje się zasadniczą częścią szeregowania. Jej celem jest znalezienie „najlepszego" procesu w kolejce procesów gotowych i przydzielenie mu procesora. Funkcja ta może być wywołana przez inne funkcje systemowe na dwa sposoby: **bezpośrednio** *(ang. direct invocation)* i **pośrednio** *(ang. lazy invocation)*.

### 1.11.1 Wywołanie bezpośrednie

Funkcja szeregująca zostaje wywołania bezpośrednio, kiedy obecny proces musi zostać natychmiast zatrzymany, ponieważ wymagany przez niego zasób nie jest w danej chwili dostępny. Wówczas algorytm postępowania, implementowany przez funkcje `wait_event()` i `wait_event_interruptible()`, jest następujący:

1. Proces `current` zostaje umieszczony we właściwej kolejce procesów oczekujących *(ang. wait queue)*
2. Jego stan zostaje zmieniony na `TASK_INTERRUPTIBLE` albo `TASK_UNINTERRUPTIBLE`
3. Zostaje wywołane `schedule()`
4. Jeżeli zasób nie jest dostępny powrót do punktu 2
5. Jeśli zasób jest dostępny, proces `current` zostaje usunięty z kolejki procesów oczekujących

Funkcję `schedule()` wywołują bezpośrednio także funkcje `sleep_on()` oraz `interruptible_sleep_on()`. Działają one podobnie do funkcji `wait_event()` i `wait_event_interruptible()`, jednak w ich przypadku po odzyskaniu procesora nie jest sprawdzana dostępność zasobu - proces od razu jest usuwany z kolejki procesów oczekujących.

Poza przedstawionymi wyżej możliwościami również wiele sterowników urządzeń bezpośrednio wywołują funkcję `schedule()`. Na ogół dotyczy to sterowników działających w pętli. W każdym obrocie pętli sprawdzana jest flaga `need_resched` aktywnego procesu i jeżeli jest ona ustawiona, to sterownik wywołuje funkcję `schedule()`.

### 1.11.2 Wywołanie pośrednie

Wywołanie pośrednie polega na ustawieniu flagi `need_resched` aktywnego procesu. Flaga ta jest ustawiana, gdy:

- Proces wykorzystał przyznany mu kwant czasu. Warunek ten jest sprawdzany w funkcji `update_process_times()`, która jest „dolną połówką" przerwania zegarowego.

- Obudzony proces ma wyższy priorytet od bieżącego. Sprawdza to funkcja `reschedule_idle()`, wywoływana przez funkcję `wake_up_process()`, która z kolei jest wywoływana przez funkcję `wake_up()`.
- Przy wywołaniu funkcji `sched_setscheduler()` i `sched_yield()`. Funkcje te zmieniają znacząco stan procesu, dla którego są wywoływane, w związku z czym konieczne jest przeszeregowanie. Szerzej na ten temat piszemy w rozdziale dotyczącym funkcji systemowych.

## 1.12 Działanie funkcji szeregującej

Funkcja `schedule()` działa wg następującego algorytmu:

1. Zanim funkcja przystąpi do czynności związanych z właściwym szeregowaniem, wywołuje zadania pozostawione przez inne funkcje jądra w różnych kolejkach. Najpierw wywołuje funkcję:

   ```
   run_task_queue(&tq_scheduler)
   ```

   która obsługuje zadania pozostawione przez inne funkcje jądra. Następnie obsługiwane są wszystkie aktywne niemaskowane dolne połówki procedur:

   ```
   if (bh_active & bh_mask)
     do_bottom_half();
   ```

   Mechanizmy **kolejek zadań** *(ang. task queues)* i **dolnych połówek** *(ang. bottom halves)* procedur jest wykorzystywany do odłożenia realizacji zadań, jeżeli w danej chwili nie można tego zrobić. W szczególności mechanizmem tym posługują się procedury obsługi przerwań, odkładając na później zadania zajmujące zbyt wiele czasu - procedura obsługi przerwania musi się wykonywać szybko.

2. Jeśli funkcję `schedule()` wywołano przy obsłudze przerwania, kończy ona działanie i wypisuje komunikat o błędzie. Wywołanie takie jest poważnym niedopatrzeniem programisty i stanowi zagrożenie dla stabilności systemu.

3. Jeśli obecny proces jest procesem czasu rzeczywistego szeregowanym według strategii `SCHED_RR`, który wykorzystał swój kwant czasu, to przyznaje mu nowy i umieszcza na końcu kolejki procesów gotowych:

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

4. Jeśli obecny proces jest w stanie `TASK_INTERRUPTIBLE` i nadszedł nieblokowany sygnał, to ustawia go na `TASK_RUNNING`. W przeciwnym przypadku usuwa proces z kolejki procesów gotowych.

5. Następnie funkcja dokonuje wyboru procesu, który otrzyma procesor. Służy do tego funkcja goodness() opisana w dalszej części pracy. Funkcja ta zwraca aktualną „wartość" procesu, tzn. ustala, który proces powinien otrzymać proces przyznając im liczby. Im liczba jest większa, tym proces jest bardziej wartościowy.
   Jeżeli aktualny proces jest w stanie TASK_RUNNING, sprawdza się, czy proces dobrowolnie zrezygnował z procesora korzystając z funkcji sched_yield(). Jeżeli tak, procesowi przydzielana jest (bez wywoływania funkcji goodness() wartość 0. W przeciwnym przypadku wywoływana jest dla niego funkcja goodness():

```
if (prev->state == TASK_RUNNING) {
  next = prev;
  if (prev->policy & SCHED_YIELD) {
    prev->policy &= ~SCHED_YIELD;
    c = 0;
  } else
    c = goodness(prev, prev);
} else {
  c = -1000;
  next = &init_task;
}
```

6. Następnie dla każdego procesu z kolejki procesów gotowych obliczana jest wartość funkcji goodness(). Spośród nich wybierany jest ten z wartością największą. Jeżeli wszystkie procesy mają wartość równą 0, tzn. że nie ma w kolejce procesów gotowych procesów czasu rzeczywistego, a wszystkie procesy zwykłe wyczerpały swój kwant. Oznacza to koniec epoki, zatem konieczne jest przeliczenie dynamicznych priorytetów wszystkich procesów, wg wzoru:

```
counter = counter / 2 + priority;
```

   Zauważmy, że mamy tu do czynienia z mechanizmem noszącym nazwę **postarzania** *(ang. aging)*, który zapobiega zagłodzeniu procesów. Ponieważ przeliczanie dynamicznych priorytetów dotyczy wszystkich procesów (a nie tylko tych z kolejki procesów gotowych), więc zyskują procesy znajdujące się poza kolejką procesów gotowych, gdyż mają one na ogół niezerowe pole counter.

   Doszliśmy do miejsca, w którym należy wyjaśnić, dlaczego wartość pola counter stuktury task_struct może być dwukrotnie większa od wartości pola priority tejże struktury. Wartość początkowa pola counter jest równa priority, a wzór na postarzanie opisuje szereg geometryczny zbieżny do 2 * priority.

7. Po wybraniu procesu, który ma otrzymać procesor następuje ostatnia część funkcji schedule(). Jeżeli wybrany został proces inny niż aktualnie wykonywany, funkcja wykonuje przełączenie kontekstu:

```
if (prev != next) {
  kstat.context_swtch++;
  switch_to(prev,next);
}
```

   Funkcja switch_to() jest zależna od architektury komputera. Dla Intela jest to makro.

## 1.13 Opis funkcji goodness()

Spójrzmy na algorytm funkcji `goodness`. W wierszach 1-2 sprawdzamy, czy proces jest procesem czasu rzeczywistego. Jeżeli tak, funkcja kończy działanie zwracając wartość, która wielokrotnie przekracza wartości dopuszczalne dla procesów zwykłych.

W dalszej części funkcji zajmujemy się już tylko procesami zwykłymi. Najpierw w wierszach 5-6 sprawdzamy, czy proces wyczerpał swój kwant. Jeżeli tak zwracamy 0. W przeciwnym przypadku w wierszach 8-9 sprawdzamy, czy proces był poprzednio wykonywany na procesorze, dla którego dokonuje się szeregowanie. Jeżeli tak, zwiększamy zwracaną wartość o stałą `PROC_CHANGE_PENALTY`, gdyż istnieje prawdopodobieństwo, że w cache'u procesora pozostały wpisy związane z tym procesem (szerzej tym tematem zajmujemy się w rozdziale „Linux/SMP"). Następnie w liniach 11-12 sprawdzamy, czy proces korzysta z tej samej przestrzeni adresowej, co proces aktualny. Jeżeli tak, to zwiększamy zwracaną wartość o 1. Takie rozwiązanie może, w przypadku procesu aktualnego, zapobiec przełączaniu kontekstu (promowany jest proces aktualny), a w odniesieniu do pozostałych procesów, może przyspieszyć tę operację. Na koniec w wierszu 14 dodajemy do zwracanej wartości liczbę tyknięć zegara, która pozostała procesowi do zakończenia jego kwantu czasu.

---

**Algorytm 1** Goodness

---

**Dane wejściowe:** prev - proces, który aktualnie ma procesor
  p - proces, którego „wartość" obliczamy
  this_cpu - numer procesora, którego dotyczy szeregowanie
**Wynik:** wartość procesu
1: **jeśli** p jest procesem czasu rzeczywistego **to**
2:   zwróć wartość 1000 + rt_priority
3: **koniec jeśli**
4: przypisz na zmienną weight wartość pola counter procesu p
5: **jeśli** wartość pola counter dla p wynosi zero **to**
6:   zwróć zero
7: **koniec jeśli**
8: **jeśli** p był wykonywany poprzednio na procesorze this_cpu **to**
9:   dodaj do weight stałą PROC_CHANGE_PENALTY
10: **koniec jeśli**
11: **jeśli** p i prev mają tę samą przestrzeń adresową **to**
12:   dodaj do weight 1
13: **koniec jeśli**
14: dodaj do weight wartość pola priority procesu p
15: zwróć wartość zmiennej weight

---

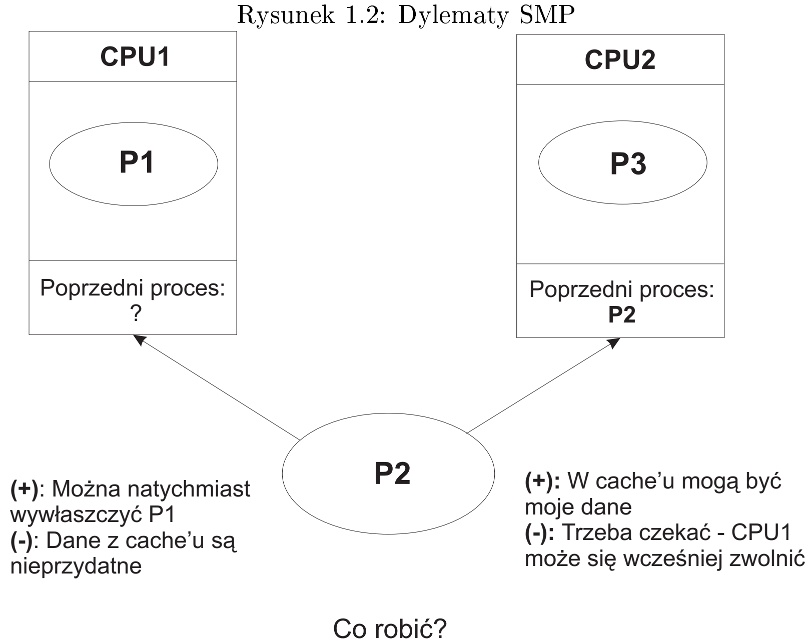
## 1.14 Linux/SMP (Symmetric MultiProcessor)

Rozszerzenie algorytmu szeregowania na systemy wieloprocesorowe stanowi w jądrze 2.2.17 kompromis pomiędzy prostotą a efektywnością. Niewielkim kosztem „zrównoleglono" wersję jednoprocesorową poprzez dodanie analizy ostatnio używanego procesora oraz szacowanie kosztu nadpisania cache'u procesora.

Wywołanie funkcji `schedule()` jest związane z konkretnym procesorem, ale aby móc realnie korzystać z dobrodziejstw wieloprocesorowości pod uwagę brana jest również informacja o pozostałych procesorach. Istotnym rozszerzeniem jest także zwielokrotnienie liczby procesów `Idle` - na każdy procesor przypada jedno takie zadanie.
Poniższy przykład prezentuje podstawowy dylemat szeregowania SMP:

> **Przykład:**
> Jednostka jest wyposażona w dwa procesory: `CPU1, CPU2`. W systemie działają trzy procesy: `P1, P2, P3` o priorytetach uporządkowanych zgodnie z ich numeracją (największy priorytet ma `P3`). `P1` wykonuje się na `CPU1, P3` na `CPU2.` `P2` poprzednio wykonywał się na `CPU2`, po czym przeszedł w stan uśpienia, z którego właśnie został obudzony. Co powinien zrobić Scheduler? Wywłaszczyć `P1` i przydzielić `P2` do `CPU1` – tracimy informację, która może znajdować się w cache'u `CPU2` i mogłaby się przydać `P2`, gdyby powrócił do ostatnio używanego procesora? Poczekać aż `P3` zwolni `CPU2` i dopiero wtedy przydzielić `P2` do `CPU2` – tracimy współbieżność, zwłaszcza gdy `CPU1` zwolni się przed `CPU2` i zostanie obsadzone przez proces `Idle`?

Rysunek 1.2: Dylematy SMP




### 1.14.1 Struktury danych dla SMP

```
– struct schedule_data {
      struct task_struct *curr;     #  aktualnie wykonywany proces
      unsigned long last_schedule;  #  czas objęcia procesora przez curr
```

`} aligned_data[NR_CPUS]`

Tablica struktur – dla każdego procesora przechowuje informację o bieżącym procesie oraz czasie rozpoczęcia jego pracy

– `cacheflush_time`
  Szacowany czas pełnego nadpisania cache'u procesora. Bezpośrednio zależy od częstotliwości oraz wielkości cache'u procesora i jest wyliczany zgodnie ze wzorem:

  ```
  ((wielkość_cache'u) / 5000 ) * częstotliwość_procesora
  ```

  Wielkość ta waha się w granicach $50$–$100$ milisekund dla Intel Pentium.

– `this_cpu`
  Określa logiczny numer procesora.

Oraz w strukturze task_struct każdego procesu:

– `avg_slice`
  Średni okres jednego wykorzystania procesora (to nie jest to samo, co kwant czasu!), wyliczany na podstawie wzoru:

  ```
  (avg_slice + this_slice) / 2,
  ```

  gdzie `this_slice` jest ostatnim okresem pobytu. Jak widać jest to średnia ważona elementami ciągu geometrycznego wszystkich poprzednich okresów wykorzystania procesora.

– `processor`
  podobnie jak `this_cpu`, tylko dla konkretnego procesu

### 1.14.2 Algorytm szeregowania SMP

Algorytm szeregowania SMP jest zatopiony w wersji algorytmu dla maszyn jednoprocesorowych. Wynika to z jego względnej prostoty. Rozbicia opisu na dwa przypadki dokonaliśmy celowo ze względu na przejrzystość podanych algorytmów.

1. Część wstępna, jak w zwyklym `schedule()`
2. W zmiennej lokalnej `this_cpu` zapamiętywany jest identyfikator aktualnie wykonującego się procesu (`prev->processor`)
3. Inicjalizacja zmiennej `sched_data` (struktura `schedule_data`)
4. Przejście kolejce procesów gotowych i wykonanie dla każdego z procesów funkcji `goodness()` — funkcja ta bierze pod uwagę również identyfikator procesora, na którym dany proces był ostatnio wykonywany - jeżeli jest to `this_cpu` procesor, do wartości `goodness` dodawany jest stały bonus (`PROC_CHANGE_PENALTY`, zazwyczaj $15$)
5. Podobnie jak wcześniej, w razie potrzeby przeliczane są priorytety wszystkich istniejących procesów.

6. Ustawiany jest wskaźnik `sched_data->curr (next)`
7. Ustawiane są pola struktury next: `has_cpu` na wartość 1 oraz processor na `this_cpu`
8. W zmiennej `t` zapamiętywany jest bieżący czas systemowy
9. W zmiennej `this_slice` zapamiętywany jest czas ostatniego wykorzystania procesora przez proces `curr` (różnica pomiędzy wartościami zmiennych `t` i `last_schedule`)
10. Zmiennej `sched_data->last_schedule` przypisywana jest wartość zmiennej `t`
11. Uaktualniana jest wartość zmiennej `prev->avg_slice:  (prev->avg_slice + this_slice)/2`
12. W razie potrzeby następuje zmiana kontekstu
13. Tutaj proces dochodzi dopiero w momencie ponownego przydziału procesora. Jeżeli proces, który go właśnie zwalnia ma się dalej wykonywać i nie jest procesem `Idle`, to wołana jest dla niego funkcja `reschedule_idle()`.
14. Procesowi prev zabierany jest procesor: `has_cpu` $= 0$

### 1.14.3 Funkcja reschedule_idle()

Funkcja `reschedule_idle()` zarządza dalszym losem świeżo wywłaszczonego procesu. Podstawową operacją z jakiej korzysta funkcja jest sprawdzenie zależności pomiędzy procesem wywłaszczonym a procesami wykonującymi się na wszystkich pozostałych procesorach. Zależność ta jest określona poprzez makro `related()`. Mówimy mianowicie, że dwa procesy `P1` i `P2` są zależne, gdy:

– `P1` wykonuje się na pewnym `CPUx`

– `P2` został właśnie wywłaszczony

– `P2` jest procesem szeregowanym zgodnie z polityką `SHED_OTHER`

– oba procesy żądają blokady jądra

– wartość `cacheflush_time` przewyższa `avg_slice` procesu `P1`

Badanie procesów zależnych jest wynikiem dwóch spostrzeżeń. Nie warto wywłaszczać procesu, który właśnie się wykonuje na pewnym procesorze i wiemy, że jego średni czas pracy jest krótszy niż czas nadpisania pamięci podręcznej. Podobna sytuacja powstaje, gdy dwa procesy żądają blokady jądra. Ponieważ tylko jeden proces mógłby z niej skorzystać, lepiej jest, gdy drugi z nich poczeka.
Właściwy algorytm prezentuje się następująco:
`P` jest wywłaszczonym procesem.

1. Jezeli `P` jest procesem czasu rzeczywistego zawsze próbuj wywłaszczyć jeden z procesów działających na innym procesorze – przejdź do 3. kroku.
2. Jeżeli istnieje procesor `CPU1` z wykonywanym procesem `P2`, takim że `P` i `P2` są zależne w myśl operacji `related()` zrezygnuj z poszukiwania nowego miejsca dla `P`. W przeciwnym przypadku: przejdź do kroku 3.

3. Jeżeli procesor, na którym `P` ostatnio pracował jest wolny (dokładniej: wykonuje się na nim proces `Idle`), `P` może rozpocząć na nim pracę.
4. W przeciwnym przypadku: dla wszystkich procesów `P2` wykonujących się na kolejnych procesorach policz różnicę:

   `preemption_goodness = goodness(P2, P) - goodness(P2, P2)`

   i wybierz ten o największej dodatniej wartości `preemption_goodness`.
5. Dla wybranego procesu `P2` ustawiana jest `flaga need_resched` i do odpowiadającego mu procesora `CPU` wysyłany jest sygnał `smp_reschedule()`. Procesor `CPU` został wybrany na miejsce pracy dla procesu `P`.

## 1.15 Funkcje systemowe związane z szeregowaniem

Poniższe funkcje służą do zmiany priorytetów procesów i ich polityki szeregowania. Zwykli użytkownicy mogą jedynie **obniżać** priorytety **swoich procesów**. Natomiast administrator *(ang. root)* może **dowolnie zmieniać** priorytety **dowolnych procesów**.

- **nice(int increment)**
  Funkcja ta służy procesowi do zmiany jego priorytetu bazowego, czyli do modyfikacji pola `priority` struktury `task_struct` procesu.

  Parametr `increment` może mieć dowolną wartość, ale i tak jego jest ona obcinana do zakresu -40..40. Ujemne wartości oznaczają podwyższenie priortytu, zaś dodatnie obniżeniu.

- **getpriority(int which, int who)**
  **setpriority(int which, int who, int niceval)**
  Funkcja `nice()` pozwala jedynie na zmianę priorytetu procesu ją wywołującego. Natomiast funkcje `getpriority()` i `setpriority()` mogą działać na większym zbiorze procesów.
  `getpriority()` zwraca wartość największego priorytetu bazowego procesów należących do danego zbioru, powiększoną o 20. `setpriority()` ustawia priorytet bazowy na daną wartość wszystkim procesom należącym do danego zbioru procesów.

  Parametr `which` określa, jaki zbiór procesów. Może mieć następujące wartości:

  `PRIO_PROCESS`
  Procesy o tym samym identyfikatorze (czyli naprawdę jeden proces).

  `PRIO_PGRP`
  Procesy należące do tej samej grupy.

  `PRIO_USER`
  Procesy należące do tego samego użytkownika.

  Parametr `who` zawiera identyfikator procesu, grupy lub użytkownika określający dany zbiór procesów.

  Parametr `niceval` oznacza nowy priorytet bazowy procesów należących do zbioru. Dopuszczalne wartości należą do zakresu od -20 (najwyższy priorytet) do 20 (najniższy priorytet). Podobnie jak w przypadku funkcji `nice()` liczba ta jest następnie konwertowana do wewnętrznej postaci jądra.

- **sched_yield()**
  Funkcja ta służy procesowi do oddania procesora na własne życzenie bez usypiania. Proces pozostaje w stanie `TASK_RUNNING`, ale zostaje przemieszczony na koniec kolejki procesów gotowych. Dzięki temu procesy o takim samym priorytecie dynamicznym mają możliwość zdobycia procesora.

  Dla procesów zwykłych funkcja ustawia w polu `policy` struktury `task_struct` bit `SCHED_YIELD`. Dzięki temu przy następnym wywołaniu funkcji `schedule()` proces będzie traktowany, jak te, które wyczerpały swój kwant czasu.

  Funkcja przesuwa dany proces na koniec kolejki procesów gotowych i ustawia flagę `need_resched` aktualnego procesu.

- **sched_getscheduler(pid_t pid)**
  Funkcja ta zwraca politykę szeregowania procesu określonego przez jego identyfikator. W szczególności pozwala stwierdzić, czy proces jest procesem zwykłym, czy też procesem czasu rzeczywistego.

## 1.16 Funkcje systemowe związane z procesami czasu rzeczywistego

- **sched_setscheduler(pid_t pid, int policy, struct sched_param *param)**
  Funkcja ta pozwala ustawić dla procesu o podanym identyfikatorze jego politykę szeregowania i statyczny priorytet procesów czasu rzeczywistego (pole `rt_priority` struktury `task_struct`). Dzięki tej funkcji można więc sprawić, że proces zwykły stanie się procesem czasu rzeczywistego, ustawiając politykę szeregowania na `SCHED_FIFO` lub `SCHED_RR`. Można także zmienić proces czasu rzeczywistego na zwykły.

  Struktura `sched_param` ma tylko jedno pole: `sched_priority`, określające statyczny priorytet czasu rzeczywistego procesu.

  Parametry procesu zmienić może właściciel, efektywny właściciel lub administrator. Dodatkowo, aby ustawić politykę szeregowania na odpowiadającą procesom czasu rzeczywistego lub zmienić statyczny priorytet potrzebne są uprawnienia administratora.

  Funkcja przesuwa dany proces na koniec kolejki procesów gotowych i ustawia flagę `need_resched` aktualnego procesu.

- **sched_getparam(pid_t pid, struct sched_param *param)**
  **sched_setparam(pid_t pid, struct sched_param *param)**
  Funkcja `sched_getparam()` zwraca w strukturze `param` dla procesu określonego przez `pid` parametry szeregowania procesu (w obecnej wersji jądra jedynie statyczny priorytet procesów czasu rzeczywistego).

  Funkcja `sched_setparam()` jest podobna do funkcji `sched_setscheduler()` (zresztą obecnie implementowana jest jako jej wywołanie). Jedyna różnica polega na tym, że przy pomocy tej funkcji nie można zmienić polityki szeregowania procesu, a jedynie zmienić jego statyczny priorytet.

- **sched_get_priority_min(int policy)**
  **sched_get_priority_max(int policy)**

Zwracają, odpowiednio, najmniejszy i największy możliwy priorytet statyczny procesów czasu rzeczywistego, szeregowanych według podanej polityki.

Mimo że w obecnej wersji jądra liczby te są stałe, użycie powyższych funkcji służy większej przenośności programów.

## 1.17 Cechy algorytmu szeregowania.

### 1.17.1 Wady strategii szeregowania

- **Algorytm nie jest dobrze skalowalny.**
  Przy dużej liczbie procesów znajdujących sie w stanie `TASK_RUNNING` koszt przeliczania priorytetów może byc znaczący. W starszych wersjach systemu UNIX priorytety dynamiczne obliczane były co sekundę, zauważalnie obniżając efektywność pracy systemu. Twórcy Linuksa postanowili zmniejszyć narzut systemowy związany z szeregowaniem. Priorytety sa przeliczane rzadko, ale operacja ta odnosi się do licznej grupy procesów, a przez to jest dość kosztowna. Prostota rozwiązania wdrożonego w Linuksie niestety powoduje zwiększenie czasu reakcji procesów interaktywnych, co przy dużej liczby zadań aktywnych może okazać sie szczególnie uciążliwe dla użytkowników.

- **Ustalona wartość kwantu czasu może być zbyt wielka.**
  Interakcyjność systemu w dużej mierze zależy od jego obciążenia, które mierzymy liczbą działających procesów. W przypadku dużego obciążenia, kwant czas przeznaczony dla pojedynczego procesu może okazać się zbyt długi, wpływając negatywnie na interakcyjność systemu. Wlasność ta nie jest zauważalna przy mniejszym wykorzystaniu systemu.

- **Preferowanie procesów interaktywnych nie jest optymalną strategią.**
  Preferowanie procesów interaktywnych poprawia szybkość reakcji na żądania użytkownika. Niestety, wiele programów wsadowych, których specyfika działania wiąże się z operacjami wejścia/wyjścia także otrzymuje bonus wynikający ich charakteru. Jednakowoż, procesy te nie są interaktywne i nie wymagają od użytkownika żadnej współpracy. Przykłdem mogą być mechanizmy bazodanowe, które przez większość czasu wykonują operacje dyskowe.

- **Zaburzenia w pracy procesów czasu rzeczywistego.**
  Jądra niewywłaszczalne *(ang. nonpreemptive kernels)* nie są dobrze przystosowane dla aplikacji działających w czasie rzeczywistym, gdyż procesy mogą wiele czasu poświęcać na obsługę przerwania, odbywającą się w trybie jądra. W tym czasie proces rzeczywisty nie może byc kontynuowany. Stanowi to zasadniczy problem dla procesów tego typu, które niemal z definicji wymagają przewidywalnie krótkich czasów reakcji. Stąd biorą się propozycje, by w przyszłości uczynić jądro w pełni wywłaszczalnym lub zastosować technikę „stałych punktów wywłaszczania", znana m.in. z `SVR4`.
  Nie jest to jednak jedyny kłopot związany z poprawnym działaniem procesów czasu rzeczywistego. Poważne problemy sprawia także oczekiwanie na zasoby wykorzystywane w danym momencie przez proces(y) o niższym priorytecie. Zjawisko to jest często nazywane „odwróceniem priorytetów". Co więcej, zdarza się, że skorzystanie z usługi jądra jest uzależnione od reakcji niżej notowanych procesów. Mamy tu do czynienia z tzw. „ukrytym szeregowaniem". Powyższe zagadnienia stanowią przedmiot pracy programistów związanych z rozwojem jądra.

### 1.17.2 Zalety strategii szeregowania

- **Prostota**
  Nie wymaga skomplikowanych operacji, nie tworzy złożonych struktur danych. Rozszerzenia względem systemu wieloprocesorowego nie komplikują znacznie algorytmu.

- **Poprawność**
  Dzięki stosowaniu priorytetów dynamicznych i wprowdzeniu pojęcia epoki, gwarantuje, iż nie wystąpi blokada. Nie zapewnia natomiast tego, że procesy zwykłe nie zostaną zagłodzone przez procesy czasu rzeczywistego.

- **Sprawiedliwość**
  Dostęp do procesora jest uzależniony od priorytetów, które definiują wagę procesu. Postarzanie sprawia, że procesy uśpione zyskują w stosunku do aktywnych i po przebudzeniu mają teoretycznie większe szanse dostępu do procesora niż pozostałe procesy. Proces potomny uzyskuje priorytet o połowę niższy od swojego rodzica, dzięki czemu próba przechwycenia procesora przez specjalnie utworzoną grupę procesów jest skazana na porażkę.

- **Efektywność**
  Priorytety procesów są rzadko przeliczane. W przypadku systemu wieloprocesorowego bonus otrzymują procesy ostatnio wykonywane na bieżącym procesorze.

# Literatura


[1] Pliki źródłowe jądra Linuksa 2.2.17: `sched.c, sched.h`

[2] Strony podręcznika systemowego:
`nice (2), setpriority (2), getpriority (2), sched_setscheduler (2), sched_getscheduler (2), sched_getparam (2), sched_setparm (2), sched_get_priority_max (2), sched_get_priority_min (2), sched_yield (2), idle (2)`

[3] Krzysztof Arciszewski, Krzysztof Sobusiak – *Projekt-Linux*, rozdział 2.3.1: *Sposoby szeregowania procesow* – 1997

[4] Grzegorz Całkowski – *Lab-Linux*, rozdział 3.3.1: *Szeregowanie Procesów* – 1998

[5] Krzysztof Lichota – *Linux Prezentacje*, część V: *Scheduler* – 1999

[6] David A. Rusling – *The Linux Kernel*, rozdział: *Scheduling* – 1998

[7] Maurice J. Bach – *Budowa systemu operacyjnego UNIX* – 1995

[8] Tigran Aivazian – *Linux Kernel Internals* – rozdział 2: *Process And Interrupt Management* – 2000