# Szeregowanie procesów w Linuksie - trendy rozwojowe

Grzegorz Chilkiewicz, Grzegorz Łuczyna, Jakub Zwolakowski

14 grudnia 2007

# Wieloprocesowość

Co to jest i po co to jest?
Wieloprocesowość = „wiele procesów może działać równocześnie i niezależnie od siebie”.

## Z punktu widzenia procesu

1. Proces = działający program + jego kontekst.
2. Każdy proces myśli, że ma cały komputer dla siebie.

## Z punktu widzenia systemu

1. Powód: Lepsze wykorzystanie dostępnych zasobów.
2. Scheduler (planista) - przydziela procesom procesor(y).

# Wieloprocesowość

Co to jest i po co to jest?
Wieloprocesowość = „wiele procesów może działać równocześnie i niezależnie od siebie”.

## Z punktu widzenia procesu

1. Proces = działający program + jego kontekst.
2. Każdy proces myśli, że ma cały komputer dla siebie.

## Z punktu widzenia systemu

1. Powód: Lepsze wykorzystanie dostępnych zasobów.
2. Scheduler (planista) - przydziela procesom procesor(y).

# Wieloprocesowość

Co to jest i po co to jest?
Wieloprocesowość = „wiele procesów może działać równocześnie i niezależnie od siebie".

## Z punktu widzenia procesu

1. Proces = działający program + jego kontekst.
2. Każdy proces myśli, że ma cały komputer dla siebie.

## Z punktu widzenia systemu

1. Powód: Lepsze wykorzystanie dostępnych zasobów.
2. Scheduler (planista) - przydziela procesom procesor(y).

# Co robi scheduler?

1. Zarządza wszystkimi procesami w systemie - trzyma je w strukturach.
2. Określa, kiedy kto dostaje procesor - strategia przydzielania czasu CPU procesom.
3. Przełącza kontekst - pracochłonna operacja. Stara się wykonywać ją jak najrzadziej.
4. Pozwala na określanie priorytetów.

# Życie procesu

## Rodzaje procesów

- Nowy - świeżo stworzony.
- Gotowy - czekający tylko na CPU.
- Wykonywany - aktualnie działający na którymś procesorze.
- Oczekujący - czekający na jakieś zdarzenie (najczęściej na zakończenie operacji I/O).
- Zakończony - zakończył wykonanie.

## Rola schedulera

Przede wszystkim zajmujemy się przejściami między stanami Gotowy i Wykonywany, bo to tutaj trzeba przydzielać procesor.

# Życie procesu

## Rodzaje procesów

- Nowy - świeżo stworzony.
- Gotowy - czekający tylko na CPU.
- Wykonywany - aktualnie działający na którymś procesorze.
- Oczekujący - czekający na jakieś zdarzenie (najczęściej na zakończenie operacji I/O).
- Zakończony - zakończył wykonanie.

## Rola schedulera

Przede wszystkim zajmujemy się przejściami między stanami Gotowy i Wykonywany, bo to tutaj trzeba przydzielać procesor.

# Cechy idealnego schedulera

Co chcemy otrzymać z punktu widzenia użytkownika:

1. Wydajność.
2. Interaktywność.
3. Sprawiedliwość i brak zagłodzenia.
4. Dobre wykorzystanie sprzętu.

**Uwaga:**

Trzeba pamiętać, że wszystko zależy od potrzeb konkretnego użytkownika!

# Cechy idealnego schedulera

Co chcemy otrzymać z punktu widzenia użytkownika:

1. Wydajność.
2. Interaktywność.
3. Sprawiedliwość i brak zagłodzenia.
4. Dobre wykorzystanie sprzętu.

**Uwaga:**

Trzeba pamiętać, że wszystko zależy od potrzeb konkretnego użytkownika!

## Wydajność

1. Kod schedulera działa szybko - szybkie struktury danych i algorytmy.
2. Jak najmniej przełączeń kontekstu.

## Interaktywność

1. Szybki czas reakcji procesów.
2. Podział na te procesy, które muszą szybko reagować i te, które nie muszą.
   (ang. „interactive task” lub „I/O bound task” oraz „CPU hog”)
3. Sprzeczne z wydajnością.

## Wydajność

1. Kod schedulera działa szybko - szybkie struktury danych i algorytmy.
2. Jak najmniej przełączeń kontekstu.

## Interaktywność

1. Szybki czas reakcji procesów.
2. Podział na te procesy, które muszą szybko reagować i te, które nie muszą.
   (ang. „interactive task" lub „I/O bound task" oraz „CPU hog")
3. Sprzeczne z wydajnością.

## Sprawiedliwość

1. Brak zagłodzenia - kluczowe.
2. Ogólnie pojęte równe traktowanie procesów (z dokładnością do priorytetów).

## Wykorzystanie sprzętu

1. Zrównoleglenie użytkowania zasobów.
2. Dobre wykorzystanie wielu procesorów.

## Sprawiedliwość

1. Brak zagłodzenia - kluczowe.
2. Ogólnie pojęte równe traktowanie procesów (z dokładnością do priorytetów).

## Wykorzystanie sprzętu

1. Zrównoleglenie użytkowania zasobów.
2. Dobre wykorzystanie wielu procesorów.

# Priorytety

Dla nas:

1. Interaktywność
2. Sprawiedliwość
3. Wykorzystanie sprzętu
4. Wydajność

Ale np. na serwerach interaktywność jest zwykle mało istotna, a przy obliczeniach numerycznych najważniejsza jest wydajność i wykorzystanie sprzętu.

# Priorytety

Dla nas:

1. Interaktywność
2. Sprawiedliwość
3. Wykorzystanie sprzętu
4. Wydajność

Ale np. na serwerach interaktywność jest zwykle mało istotna, a przy obliczeniach numerycznych najważniejsza jest wydajność i wykorzystanie sprzętu.

# Cechy schedulera z jądra 2.4.x

## Epoki

1. Czas dzielony na epoki.
2. Scheduler iteruje po wszystkich procesach i znajduje ten z najwyższym priorytetem – czas O(n).
3. W czasie epoki każdy proces może zużyć swój timeslice.
4. Timeslice liczony na początku epoki – czas O(n).
5. (Kwant czasu - jeden tick schedulera.)

## Wsparcie interaktywności

Przy wyliczaniu nowego timeslice'a scheduler dodaje połowę czasu niewykorzystanego w poprzedniej epoce.
Jeśli proces nagle zacznie potrzebować dużo CPU, to zużyje ten zapas w ciągu jednej epoki.

# Cechy schedulera z jądra 2.4.x

## Epoki

1. Czas dzielony na epoki.
2. Scheduler iteruje po wszystkich procesach i znajduje ten z najwyższym priorytetem – czas O(n).
3. W czasie epoki każdy proces może zużyć swój timeslice.
4. Timeslice liczony na początku epoki – czas O(n).
5. (Kwant czasu - jeden tick schedulera.)

## Wsparcie interaktywności

Przy wyliczaniu nowego timeslice'a scheduler dodaje połowę czasu niewykorzystanego w poprzedniej epoce.
Jeśli proces nagle zacznie potrzebować dużo CPU, to zużyje ten zapas w ciągu jednej epoki.

# Ocena

## Plusy

1. Działa (nawet w porównaniu wypada ponoć dość dobrze).
2. Jest prosty (około 1/3 kodu następnego schedulera).

## Minusy

1. Kiepska skalowalność - to oczywiście wina czasów O(n).
2. Duże timeslices - procesy o niskim priorytecie mają problem.
3. Mało skuteczne wsparcie interaktywności.

# Ocena

## Plusy

1. Działa (nawet w porównaniu wypada ponoć dość dobrze).
2. Jest prosty (około 1/3 kodu następnego schedulera).

## Minusy

1. Kiepska skalowalność - to oczywiście wina czasów O(n).
2. Duże timeslices - procesy o niskim priorytecie mają problem.
3. Mało skuteczne wsparcie interaktywności.

# Zupełnie nowy scheduler

1. Napisany od nowa przez Ingo Molnara (stwierdził, że poprzedniego nie da się sensownie zmodyfikować).
2. Zastąpienie wszystkich algorytmów O(n) przez O(1) - stąd wzięła się nazwa.
3. Poprawione wsparcie dla interaktywności.

# Zmiany

1. Runqueues - kolejki procesów gotowych, jedna dla każdego procesora.
2. Priority arrays - pod dwie dla każdej runqueue (active i expired). Wybieranie procesu o najwyższym priorytecie w czasie O(1).
3. Nowa epoka w czasie O(1) - wysarczy zamienić tablice active i expired.
4. Dynamiczne priorytety - heurystyka wykrywania procesów interaktywnych na podstawie średniego czasu snu.
5. Interaktywne procesy nie wstawiane do expired (z dokładneością do ochrony przed zagłodzeniem).
6. (Kwant czasu - jiffie.)

# Ocena

## Plusy

1. Poprawiona wydajność i koniec problemów ze skalowalnością.
2. Poprawiona interaktywność - lepsza heurystyka.
3. Całość działa bardzo bardzo dobrze!

## Minusy

1. Problem z dużymi timeslices tak naprawdę nie został rozwiązany, tylko zmniejszony dwukrotnie.
2. Interaktywność wciąż nie zawsze idealna.

# Ocena

## Plusy

1. Poprawiona wydajność i koniec problemów ze skalowalnością.
2. Poprawiona interaktywność - lepsza heurystyka.
3. Całość działa bardzo bardzo dobrze!

## Minusy

1. Problem z dużymi timeslices tak naprawdę nie został rozwiązany, tylko zmniejszony dwukrotnie.
2. Interaktywność wciąż nie zawsze idealna.

# Zamieszanie

1. Miał być dalej O(1), choć znaleziono sporo „corner cases".
2. Con Kolivas próbuje dostroić O(1) i mu się to udaje. Odkrywa jednak, że cały fragment tego schedulera odpowiadający za określanie interaktwności często bardziej przeszkadza, niż pomaga.
3. Napisał więc RSDL (Rotating Staircase Deadline Scheduler), który całkowicie olewa próbę przewidywania interaktywności i skupia się na sprawiedliwości (fairness).
4. Okazuje się, że działa bardzo dobrze! Ma zostać w jądrze 2.6.23
5. Ale Ingo Molnar tworzy Completely Fair Scheduler (w 62 godziny!) bazując na pomyśle Cona Kolivasa i to CFS trafia do nowego jądra.

## Sprawiedliwość - fairness

- W teorii - całkowita.
- W praktyce - złoty środek między sprawiedliwością a interaktywnością.

## Pomysł

- Chcemy zbliżyć się jak najbardziej do „idealnego wieloprocesowego CPU”.
- Każdy z n procesów dostaje 1/n czasu procesora.
- W praktyce oczywiście niemożliwe, bo przełączanie kontekstu trwa sporo czasu.
- Zatem uruchamiamy zawsze proces, który jest najbardziej do tyłu względem czasu, jaki mu się by należał.
- (Różnica - w RDSL wliczamy czas snu, w CFS nie wliczamy.)

## Sprawiedliwość - fairness

- W teorii - całkowita.
- W praktyce - złoty środek między sprawiedliwością a interaktywnością.

## Pomysł

- Chcemy zbliżyć się jak najbardziej do „idealnego wieloprocesowego CPU”.
- Każdy z n procesów dostaje 1/n czasu procesora.
- W praktyce oczywiście niemożliwe, bo przełączanie kontekstu trwa sporo czasu.
- Zatem uruchamiamy zawsze proces, który jest najbardziej do tyłu względem czasu, jaki mu się by należał.
- (Różnica - w RDSL wliczamy czas snu, w CFS nie wliczamy.)

# Cechy CFSa

## wait_runtime

Zawiera informację, ile czasu pownien teraz działać proces, żeby odzyskać „naturalną równowagę”.

## Szukanie następnego procesu

Używane jest do tego drzewo czerwono-czarne porządkujące procesy według wait_runtime. Gdy aktualny proces przestanie mieć największą potrzebę czasu, to spada w drzewie i odbiera mu się procesor (z dokładnością do pewnej granularności).

## Granularność

Czas mierzony naturalnie - w nanosekundach.
Można ustalać granularność, aby zmienić działanie z desktopowego na serwerowe.

# Cechy CFSa

## wait_runtime

Zawiera informację, ile czasu pownien teraz działać proces, żeby odzyskać „naturalną równowagę”.

## Szukanie następnego procesu

Używane jest do tego drzewo czerwono-czarne porządkujące procesy według wait_runtime. Gdy aktualny proces przestanie mieć największą potrzebę czasu, to spada w drzewie i odbiera mu się procesor (z dokładnością do pewnej granularności).

## Granularność

Czas mierzony naturalnie - w nanosekundach.
Można ustalać granularność, aby zmienić działanie z desktopowego na serwerowe.

# Cechy CFSa

## wait_runtime

Zawiera informację, ile czasu pownien teraz działać proces, żeby odzyskać „naturalną równowagę".

## Szukanie następnego procesu

Używane jest do tego drzewo czerwono-czarne porządkujące procesy według wait_runtime. Gdy aktualny proces przestanie mieć największą potrzebę czasu, to spada w drzewie i odbiera mu się procesor (z dokładnością do pewnej granularności).

## Granularność

Czas mierzony naturalnie - w nanosekundach.
Można ustalać granularność, aby zmienić działanie z desktopowego na serwerowe.

# Cechy CFSa

## wait_runtime

Zawiera informację, ile czasu pownien teraz działać proces, żeby odzyskać „naturalną równowagę".

## Szukanie następnego procesu

Używane jest do tego drzewo czerwono-czarne porządkujące procesy według wait_runtime. Gdy aktualny proces przestanie mieć największą potrzebę czasu, to spada w drzewie i odbiera mu się procesor (z dokładnością do pewnej granularności).

## Granularność

Czas mierzony naturalnie - w nanosekundach.
Można ustalać granularność, aby zmienić działanie z desktopowego na serwerowe.

# Pliki

## Dawniej

- include/linux/sched.h
- kernel/sched.c

## Teraz

- include/linux/sched.h
- kernel/sched.c
- kernel/sched_debug.c
- kernel/sched_fair.c
- kernel/sched_idletask.c
- kernel/rt.c
- kernel/sched_stats.c

# Pliki

## Dawniej

- include/linux/sched.h
- kernel/sched.c

## Teraz

- include/linux/sched.h
- kernel/sched.c
- kernel/sched_debug.c
- kernel/sched_fair.c
- kernel/sched_idletask.c
- kernel/rt.c
- kernel/sched_stats.c

# Struktury procesów

```
struct task_struct {
  ...
  struct sched_entity se;
  ...
};
```

```
struct sched_entity {
  ...
  long wait_runtime;
  struct rb_node run_node;
  struct cfs_rq *cfs_rq;
  struct cfs_rq *my_q;
  ...
};
```

# Struktury procesów

```
struct task_struct {
  ...
  struct sched_entity se;
  ...
};
```

```
struct sched_entity {
  ...
  long wait_runtime;
  struct rb_node run_node;
  struct cfs_rq *cfs_rq;
  struct cfs_rq *my_q;
  ...
};
```

# Struktury kolejek

```
struct rq {
  ...
  struct cfs_rq cfs;
  struct rt_rq rt;
  ...
};
```

```
struct cfs_rq {
  ...
  struct rb_root tasks_timeline;
  struct rb_node *rb_leftmost;
  struct rb_node *rb_load_balance_curr;
  ...
};
```

# Struktury kolejek

```
struct rq {
  ...
  struct cfs_rq cfs;
  struct rt_rq rt;
  ...
};
```

```
struct cfs_rq {
  ...
  struct rb_root tasks_timeline;
  struct rb_node *rb_leftmost;
  struct rb_node *rb_load_balance_curr;
  ...
};
```

# Wnioski

## Wybieralność schedulerów

- Pliki z różnie działającymi schedulerami.
- `struct sched_class`
- W rzeczywistości, dla zwykłego użytkownika, tylko pozorna wybieralność schedulera.

## Złożoność

- Drzewo czerwono-czarne do przechowywania działających procesów. Algorytmy O(log n).

## Grupy procesów

- Część operacji działa na sched_entity zamiast na task_struct.
- Pole my_q w sched_entity.

# Wnioski

## Wybieralność schedulerów

- Pliki z różnie działającymi schedulerami.
- `struct sched_class`
- W rzeczywistości, dla zwykłego użytkownika, tylko pozorna wybieralność schedulera.

## Złożoność

- Drzewo czerwono-czarne do przechowywania działających procesów. Algorytmy O(log n).

## Grupy procesów

- Część operacji działa na sched_entity zamiast na task_struct.
- Pole my_q w sched_entity.

# Wnioski

## Wybieralność schedulerów

- Pliki z różnie działającymi schedulerami.
- `struct sched_class`
- W rzeczywistości, dla zwykłego użytkownika, tylko pozorna wybieralność schedulera.

## Złożoność

- Drzewo czerwono-czarne do przechowywania działających procesów. Algorytmy O(log n).

## Grupy procesów

- Część operacji działa na sched_entity zamiast na task_struct.
- Pole my_q w sched_entity.

# Testowanie

## Ogólnie

- Co testujemy?
- Jak testować wydajność?

## Hackbench

- Co to jest hackbench?
- Omówienie działania hackbench.c.

## Zarządzanie procesami

- ps
- nice
- renice

# Testowanie

## Ogólnie

- Co testujemy?
- Jak testować wydajność?

## Hackbench

- Co to jest hackbench?
- Omówienie działania hackbench.c.

## Zarządzanie procesami

- ps
- nice
- renice

# Testowanie

## Ogólnie

- Co testujemy?
- Jak testować wydajność?

## Hackbench

- Co to jest hackbench?
- Omówienie działania hackbench.c.

## Zarządzanie procesami

- ps
- nice
- renice

# Testy

## Wyniki

- Omówienie wykresu zachowania hackbench.c dla trzech schedulerów:
  1. 2.4
  2. O(1)
  3. CFS

  WYNIKI.ods
- Dlaczego scheduler 2.4 faworyzuje hackbench uruchomiony z 'nice -n 19' ?

# Pojedynek

## O(1) a CFS

- Kompilowanie jądra linuxa i jednoczesne oglądanie filmu w serwisie Youtube.com.
- Przyjaźniejsze dla użytkownika działanie schedulera CFS.
- Sposoby radzenia sobie ze schedulerem O(1).

# 'Zaawansowane' narzędzie

## PS-DOOM

- PS-DOOM, czyli narzędzie do zarządzania procesami w systemie linux.
- Szczególnie dla rozrywkowych administratorów...