Linux Scheduler
Szeregowanie procesów w systemie Linux

Andrzej Ambroziewicz
Piotr Marat
Mieszko Michalski

Zawartość prezentacji

- Linux Scheduler 2.4
- Linux Scheduler 2.6
- CFS (Completely Fair Scheduler)
- Porównanie i testy

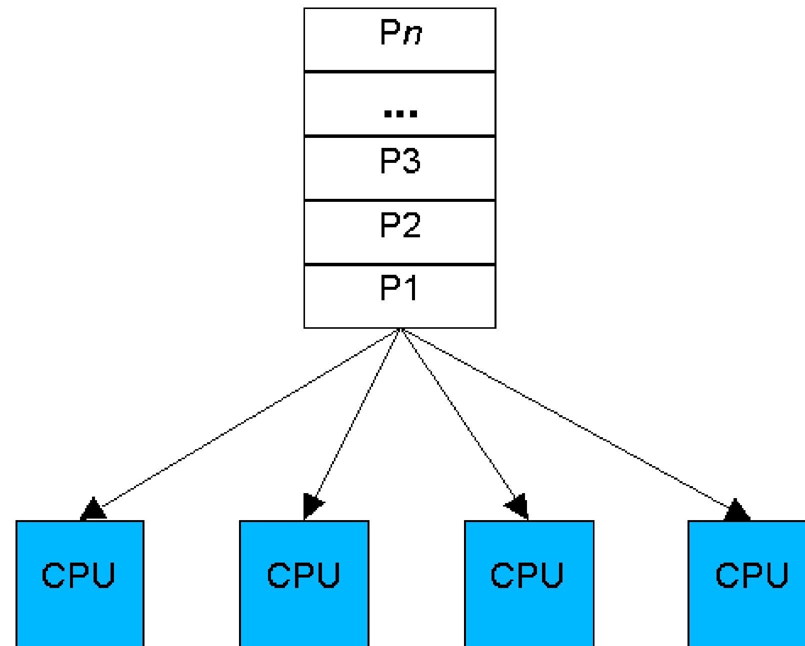
Klasy procesów

- Procesy czasu rzeczywistego SCHED_RR i SCHED_FIFO
- Zwalniają procesor gdy:
  a) W kolejce procesów gotowych pojawi się proces o wyższym priorytecie (to może być tylko proces czasu rzeczywistego)
  b) Zablokują się w oczekiwaniu na zasób
  c) Dobrowolnie zrezygnują z procesora
  d) Zakończą się
  e) SCHED_RR gdy skończy się kwant czasu
- Procesy użytkownika SCHED_NORMAL

Szeregowanie w jądrze 2.4

- Plik: kernel/sched.c (1398 linii kodu)
- Napisany całkowicie od nowa
- Złożoność O(n)
- Jedna kolejka procesów nawet dla systemów wieloprocesorowych

Szeregowanie w jądrze 2.4



Metryka procesu – task_struct

Jądro 2.4

- need_resched – flaga mówiąca czy należy wywołać schedule() “przy następnej okazji”
- counter – licznik taktów zegara pozostałych procesorowi z kwantu czasu
- nice – statyczny priorytet
- rt_priority – priorytet czasu rzeczywistego
- policy – klasa procesu (SCHED_RR, SCHED_FIFO, SCHED_OTHER + SCHED_YIELD)

Funkcja schedule()

Jądro 2.4

1) Jeśli current->active_mm == NULL podnosimy błąd (proces zawsze musi mieć to pole ustawione)

2) Zapamiętujemy na zmiennych prev i this_cpu aktualny proces działający i procesor

3) Zwalniamy główną blokadę jądra (kernel_lock)

4) Zaczynamy manipulacje na kolejce procesów gotowych: zakładamy blokadę na runqueue

5) Jeśli proces był typu SCHED_RR i wykorzystał swój kwant czasu przydzielamy mu nowy kwant I umieszczamy na koncu kolejki.

Funkcja schedule()

Jądro 2.4

6) Jeśli proces był w stanie TASK_INTERRUPTIBLE i nadszedł jakiś sygnał zmieniamy stan na TASK_RUNNABLE. Jeśli był w stanie TASK_RUNNABLE zostawiamy go, w przeciwnym wypadku jest usuwany z kolejki

7) Wybieramy proces, który ma działać następny. Domyślnie ustawiamy idle ale oceniamy go bardzo nisko (-1000). Przechodzimy listę procesów gotowych i obliczamy dla nich parametr goodness(), wybieramy najwyższy

Funkcja schedule()

Jądro 2.4

8) Funkcja goodness(task_struct * p, int this_cpu,mm_struct *this_mm)
   a) Jeśli proces sam oddał procesor (SCHED_YIELD) zwracamy -1.
   b) Jeśli jest to proces zwykły ustawiamy parametr na wartość pola, counter (jeśli to 0 to wychodzimy – proces wykorzystawł już czas)
      1) faworyzujemy procesy działające na naszym procesorze (+15)
      2) dodajemy +1 jeśli proces korzysta z załadowanej pamięci I dla wątków jądra
      3) uwzględniamy parametr nice
   c) dla procesów czasu rzeczywistego wstawiamy wartość 1000+rt_priority

9) Jeśli po przejściu całej listy maksymalna wartość wynosi 0 to znaczy, że wszystkim procesom skończył się kwant czasowy i należy obliczyć je od nowa (procesy z większym priorytetem mogą działac dłużej)

10) Jeśli trzeba, przełączamy kontekst (makro switch_to())

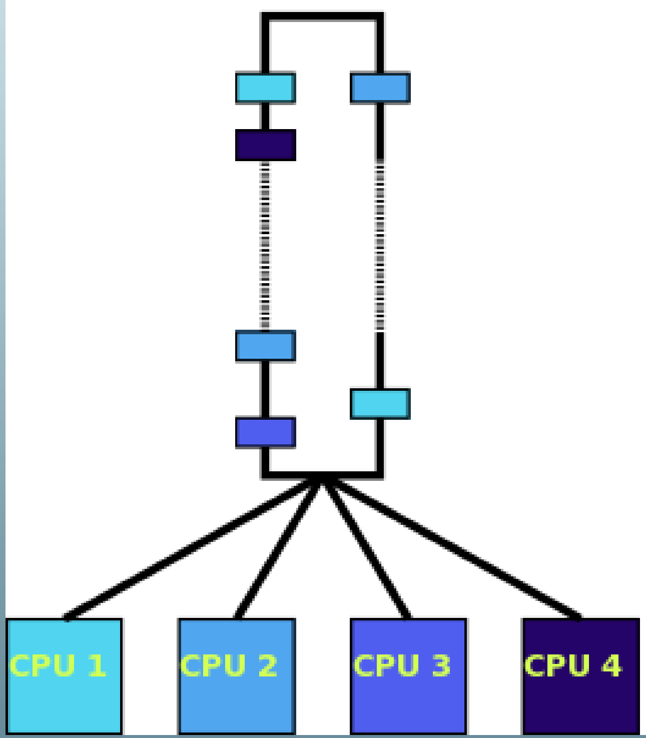
Zmiany w jądrze 2.6

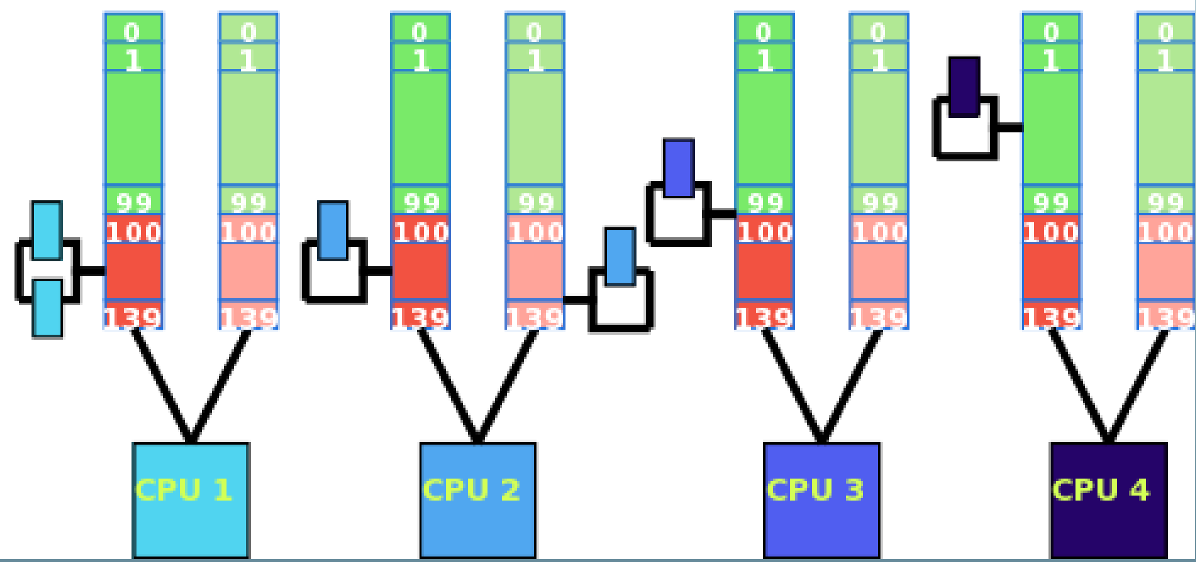
- Pliki kernel/sched.c (7200+ linii kodu)
- Napisany całkowicie od nowa
- Złożoność O(1)
- Osobna kolejka procesów dla każdego procesora

Zmiany w jądrze 2.6

- Inna metoda wyznaczania priorytetów dynamicznych
  (static_prio, prio, NICE_TO_PRIO(nice), sleep_avg, timestamp)
- 2 kolejki priorytetowe (indeksowane priorytetami 0..139)
  expired i active, pamiętana bitmapa i ilość procesów
- schedule() i scheduler_tick()
- Wywłaszczenie w trybie jądra(fully preemptive kernel)
- Wsparcie dla maszyn wieloprocesorowych – zrównoważenie obciążenia

2.4 vs 2.6





Wywłaszczanie

- Flaga need_resched mówi, że trzeba wykonać schedule() gdy tylko będzie można. Ustawiana jest w procedurach scheduler_tick() i try_to_wake_up() gdy proces budzony ma większy priorytet
- Jest trzymana w thread_info() w task_struct, bo jest tam szybciej osiągalna niż zmienne globalne

Wywłaszczenie (tryb użytkownika)

- Jądro sprawdza flagę need_resched przy powrocie do trybu użytkownika w związku z zakończeniem wywoływania funkcji systemowej lub obsługi przerwania
- Jest to stan bezpieczny, możemy wywołać schedule()

Wywłaszczanie (tryb jądra)

Jądro 2.6

- Wprowadzone w 2.6, wiele systemów nie obsługuje wywłaszczania w trybie jądra (kod jądra wywołuje się aż do zakończenia)
- Możemy wywłaszczać tylko wtedy gdy jesteśmy w stanie bezpiecznym, ręcznie wywołaliśmy schedule lub aktualne zadanie oczekuje na zdarzenie.
- Jesteśmy w stanie bezpiecznym gdy nie są założone żadne blokady, zwiększamy wtedy licznik preempt_count w task_struct

Zrównoważenie obciążenia procesorów

Jądro 2.6 (SMP)

- Uruchamiamy proces stale na tym samym procesorze
- Sprawdzamy czy nie powstała przez to różnica w obciążeniu procesorów
- Jeśli tak, dokonujemy zrównoważenia
- Procedura wywoływana jest w schedule() oraz co 1 (jeśli system jest bezczynny) lub 200 milisekund
- Uruchamiana jest z wyłączonymi przerwaniami i założoną blokadą na lokalną kolejkę, aby wykluczyć jednoczesne zrównoważenia

Zrównoważenie obciążenia procesorów

Jądro 2.6 (SMP)

- Sprawdzamy czy jest kolejka dłuższa od naszej o conajmniej 25%
- Wybiera kolejke, z której wyciągane będą procesy (pierszeństwo ma expired, bo procesy z niej najczęściej nie trzymają już nic w pamięci podręcznej)
- Przerzucamy procesy o najwyższych priorytetach, przy czym sprawdzamy, czy proces nie działa na tym procesorze, czy można go przerzucić i czy nie ma niczego w pamięci podręcznej

Materiały

- R. Love – Linux Kernel Development (rozdział 3)
- http://www.informit.com/articles/article.aspx?p=101760
- http://lxr.linux.no/ - Linux Cross Reference
- Google
- students/SO – Wykład 3
- http://www.faqs.org/docs/kernel_2_4/lki-2.html
- http://kerneltrap.org/

Wady Vanilla Scheduler i CFS jako alternatywa nie posiadająca tych wad.

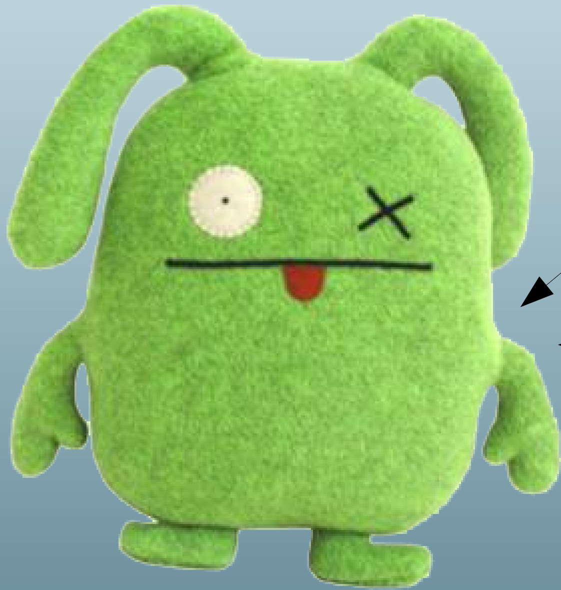
Wady poprzedniego:

- Czasami bezsensowne przydzielanie czasu względem priorytetów

Przykład: 2 wątki w systemie – dla priorytetu 0 przydzielony czas to 100 ms:

Niemiły wątek - nice=0

Milszy wątek - nice=19

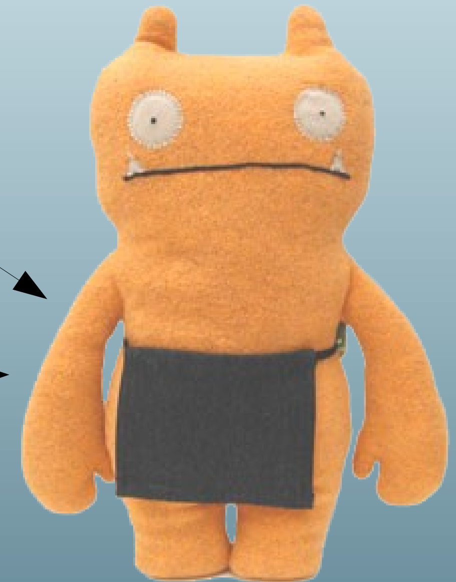


Dostanie 100 ms

100/105 czasu

Dostanie 5ms

Tylko 5/105 czasu



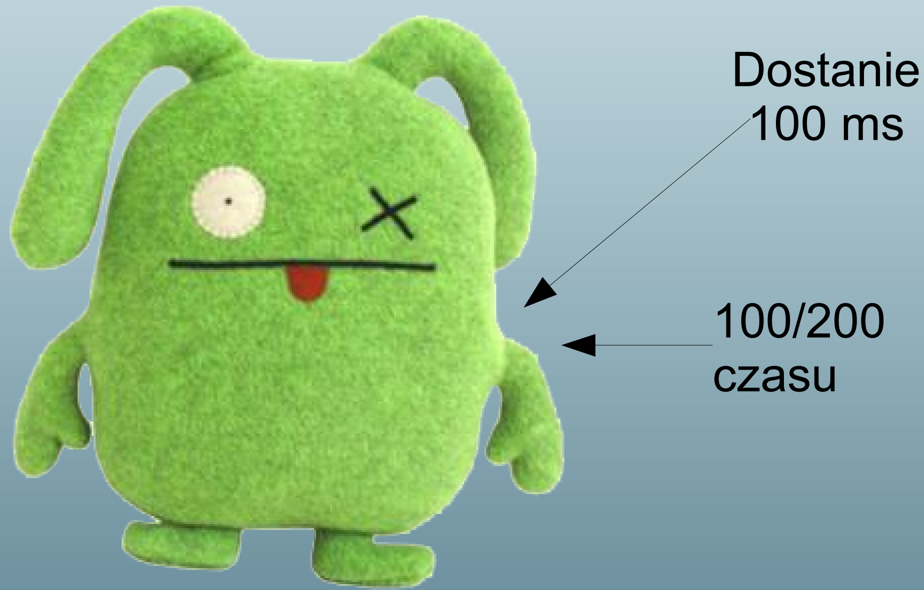
Wady poprzedniego:

- Czasami bezsensowne przydzielanie czasu względem priorytetów

Przykład: 2 wątki w systemie – dla priorytetu 0 przydzielony czas to 100 ms, tym razem równe priorytety:

Niemiły wątek - nice=0

Tak samo niemiły wątek - nice=0

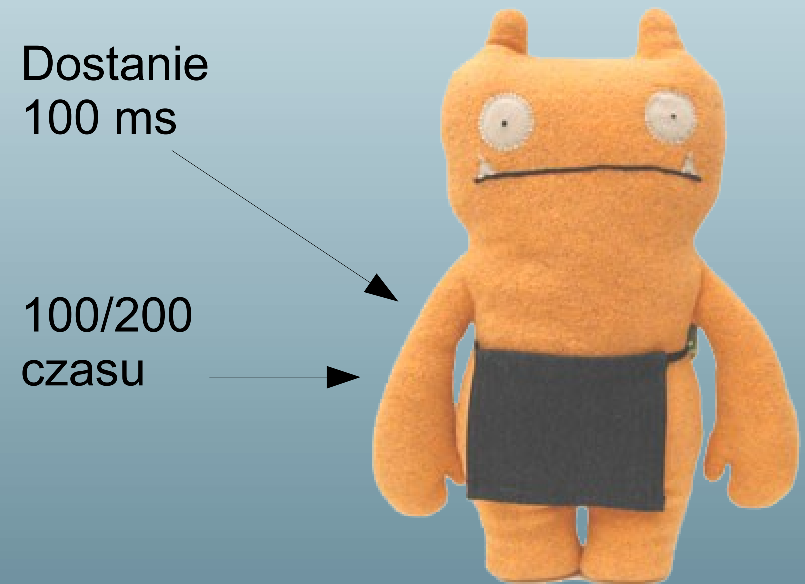


Dostanie 100 ms

100/200 czasu

Dostanie 100 ms

100/200 czasu



Ale co jeśli zamiast nice=0, każdy z nich będzie miał priorytet nice=19?

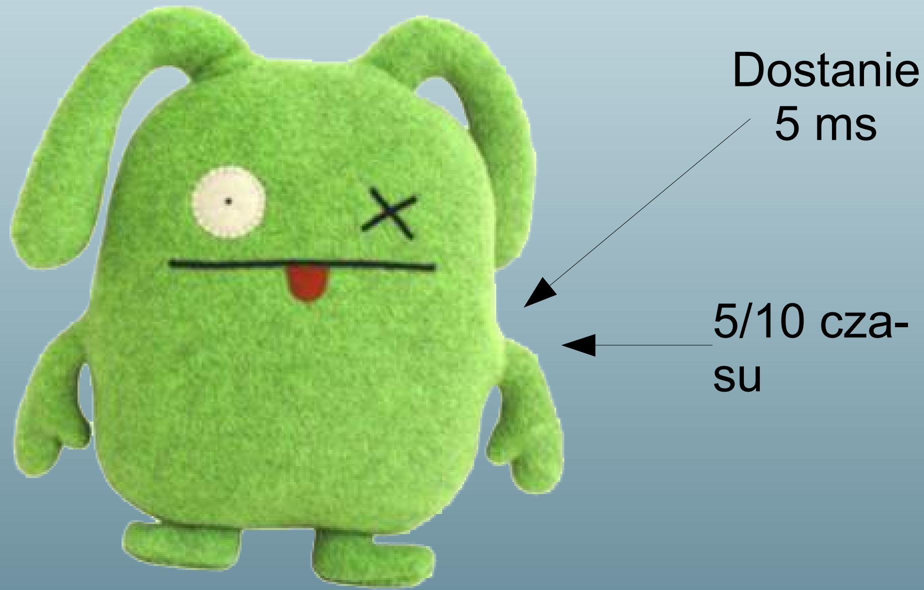
Wady poprzedniego:

- Czasami bezsensowne przydzielanie czasu względem priorytetów

Przykład: 2 wątki w systemie – dla priorytetu 0 przydzielony czas to 100 ms, tym razem równe priorytety:

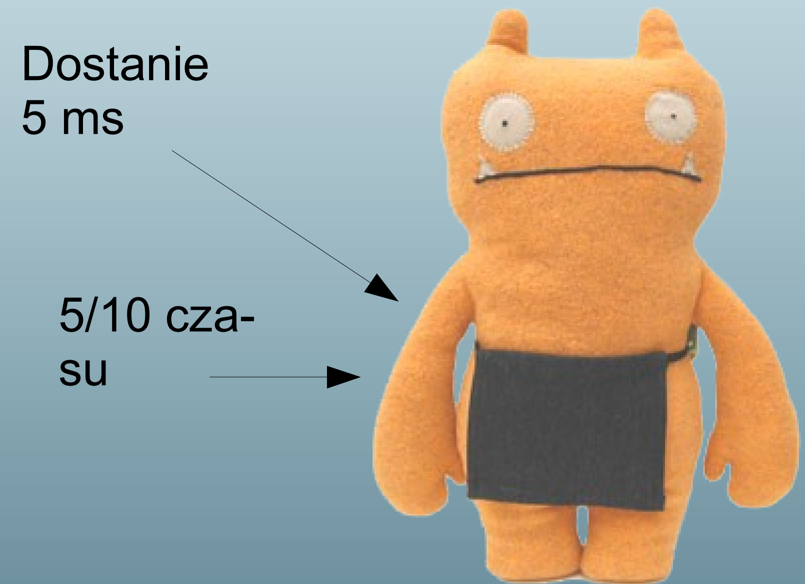
Miły wątek - nice=19

Tak samo miły wątek – nice=19



Dostanie 5 ms

5/10 czasu



Dostanie 5 ms

5/10 czasu

Choć stosunek jest taki sam, to dużo częstszy jest czas przełączania kontekstu, a co za tym idzie spada wydajność systemu.

Wady poprzedniego:

- Czasami bezsensowne przydzielanie czasu względem priorytetów
- Niesprawiedliwa zależność pomiędzy priorytetami

Przykład: 2 wątki w systemie – dla priorytetu 0 przydzielony czas to 100 ms, priorytety różnią się o 1

Niemiły wątek - nice=0

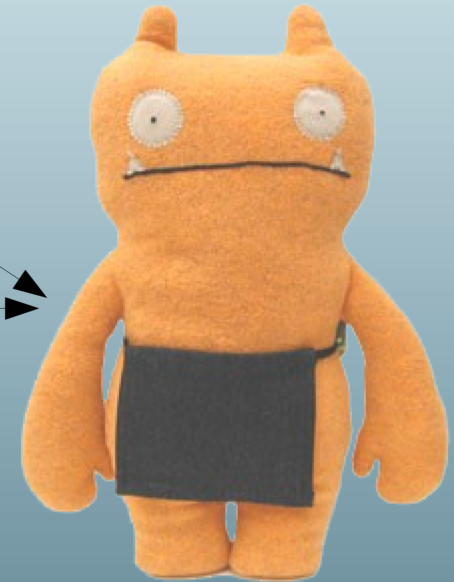
Milszy wątek – nice=1

Dostanie 100 ms

100/195 czasu

Dostanie 95 ms

95/195 czasu



Oba wątki dostaną po prawie tyle samo czasu

Wady poprzedniego:

- Czasami bezsensowne przydzielanie czasu względem priorytetów
- Niesprawiedliwa zależność pomiędzy priorytetami

Przykład: 2 wątki w systemie – dla priorytetu 0 przydzielony czas to 100 ms, priorytety różnią się o 1

Miły wątek - nice=18

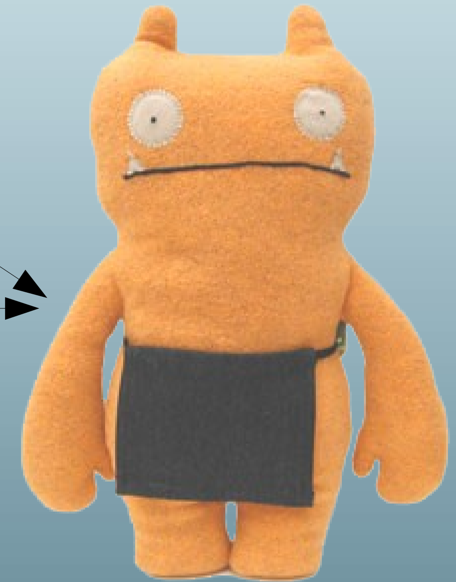
Jeszcze milszy wątek – nice=19

Dostanie 10 ms

10/15 czasu

Dostanie 5 ms

5/15 cza-su



Wątek z priorytetem 19 dostanie 2 razy mniej czasu niż ten z priorytetem 18.

Wady poprzedniego:

- Czasami bezsensowne przydzielanie czasu względem priorytetów
- Niesprawiedliwa zależność pomiędzy priorytetami
- Nieefektywne budzenie wątków - Przydziela za mało czasu procesora na takie usługi jak wyświetlanie interfejsu lub odtwarzanie multimediów.

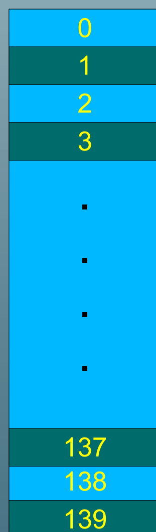
Obudzony wątek trafia do „active array” zgodnie z priorytetem, zatem wykonywany wątek, który ostatnio zasnął, może powrócić do „active array”, zamiast do „expired array”. Dopóki w „active array” są wątki, dopóty wątki z „expired array” nie otrzymają czasu procesora. Ten fakt może powodować, że ciągły strumień budzących się wątków opóźni na dowolnie długo wątki z „expired array”. Zatem można mając nawet tylko kilka wątków stworzyć sytuację, gdzie opóźnienie sięga kilku sekund.

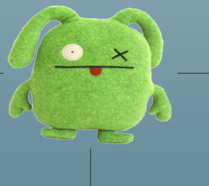
Wady poprzedniego:

- Czasami bezsensowne przydzielanie czasu względem priorytetów
- Niesprawiedliwa zależność pomiędzy priorytetami
- Nieefektywne budzenie wątków - Przydzielał on za mało czasu procesora na takie usługi jak wyświetlanie interfejsu lub odtwarzanie multimediów.

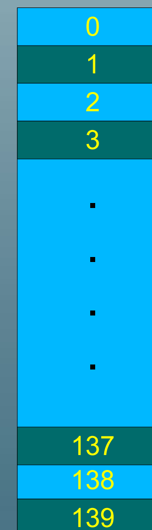
Obudzony wątek trafia do „active array” zgodnie z priorytetem, zatem wykonywany wątek, który ostatnio zasnął, może powrócić do „active array”, zamiast do „expired array”. Dopóki w „active array” są wątki, dopóty wątki z „expired array” nie otrzymają czasu procesora. Ten fakt może powodować, że ciągły strumień budzących się wątków opóźni na dowolnie długo wątki z „expired array”. Zatem można mając nawet tylko kilka wątków stworzyć sytuację, gdzie opóźnienie sięga kilku sekund.

Active array





Expired array



Wady poprzedniego:

- Czasami bezsensowne przydzielanie czasu względem priorytetów
- Niesprawiedliwa zależność pomiędzy priorytetami
- Nieefektywne budzenie wątków

Zły podział może prowadzić do zapchania systemu:

Completely Fair Scheduler

O ile 2 pierwsze problemy mogłyby być wyeliminowane poprzez pewne modyfikacje w bazowym algorytmie, o tyle z 3 był o wiele większy problem.
Zatem potrzebna była gruntowna zmiana „vanilla scheduler”.

Twórca Ingo Molnar powiedział: *"80% Of CFS's design can be summed up in a single sentence: CFS basically models an 'ideal, precise multi-tasking CPU' on real hardware."*

- W ciągu 62 godzin (środa 8 rano – piątek 22) Ingo Molnar stworzył 100K patch pozwalający Linuxowi na obsługę CFS-a, (co ciekawe nie przedyskutował swojego pomysłu wczesniej na „Linux Kernel Mailing List”) potem przez 6 godzin testował go aby nie wysadzić w powietrze komputera potencjalnego użytkownika:)
- Napisał na LKML o gotowej wersji, po czym uzgodnił szczegóły z 2 osobami, które udało mu się złapać na IRC-u. Przez następne 2 godziny dyskutował z nimi na ten temat, poczym puścił go w świat.
- W kolejnych poprawkach wyeliminowano niewielkie błędy i usprawniono jego działanie.
- Linus zaakceptował włączenie go do oficjalnego wydania jądra odrzucając tym samym konkurencyjny scheduler o nazwie SD.

Completely Fair Scheduler

•Własności:

➢Nie opiera się na kolejkach wykonań, tylko na drzewie czerwono-czarnym uszeregowanym względem czasu, na którego podstawie ustalana jest kolejność wykonania

➢Używa nanosekundowej ziarnistości, nie opierając się o detale takie jak jiffies czy HZ

➢Nie opiera się na heurystykach (odporny na ataki heurystyczne), istnieje tylko jeden główny plik konfiguracyjny /proc/sys/kernel/sched_granularity_ns pozwalający dostosować scheduler do zastosowań zarówno w komputerach domowych jak i w serwerach.

➢Bardziej silna reakcja na zmiany priorytetów

➢Implementuje zarządzanie procesami czasu rzeczywistego (SCHED_FIFO i SCHED_RR) w prostszy i bardziej efektywny sposób niż „vanilla scheduler”. Używa 100 kolejek dla wszystkich 100 priorytetów RT zamiast 140 oraz nie używa „expired array”.

➢Uśpione procesy mają takie same prawa do czasu procesora, jak te wykonywane

➢Lepsze zarządzanie procesem STATE_IDLE i procesami o niskich priorytetach

➢Mniej kodu o ok. 700 linijek :) w porównaniu do „vanilla scheduler”

➢Przerobiony kod rozkładający obciążenie pomiędzy wieloma procesorami

Linux
Linux
DM 10,99

Completely Fair Scheduler

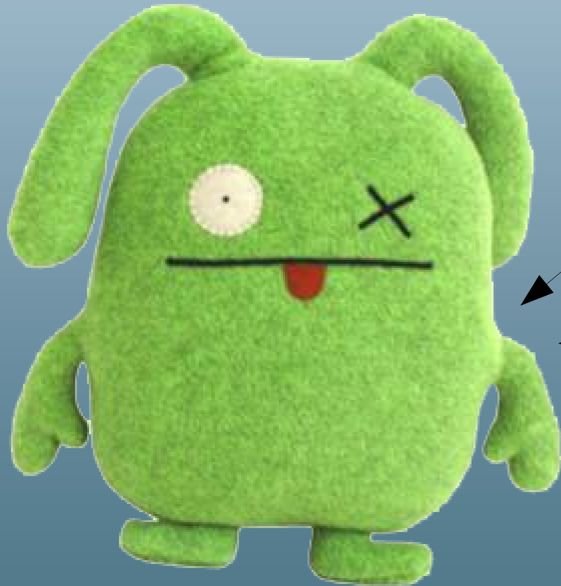
Opis działania:

- CFS zamiast przypisywać do każdego priorytetu kwant czasu, przypisuje do priorytetu wagę, która jest liczona na podstawie wykonywanych procesów. Każdy wątek otrzymuje proporcjonalną ilość czasu do swojego priorytetu podzielonego przez łączną wagę wszystkich wątków. W CFS określany jest czas pełnego obrotu „round-robin” zamiast określania czasu dla priorytetu.

Przykład: Znów mamy 2 procesy i czas tym razem na pełen obrót (a nie na 1 wątek), to 100 ms

Niemiły wątek - nice=0

Tak samo niemiły wątek – nice=0



Dostanie 50 ms

50/100 czasu

Dostanie 50ms

50/100 czasu

Completely Fair Scheduler

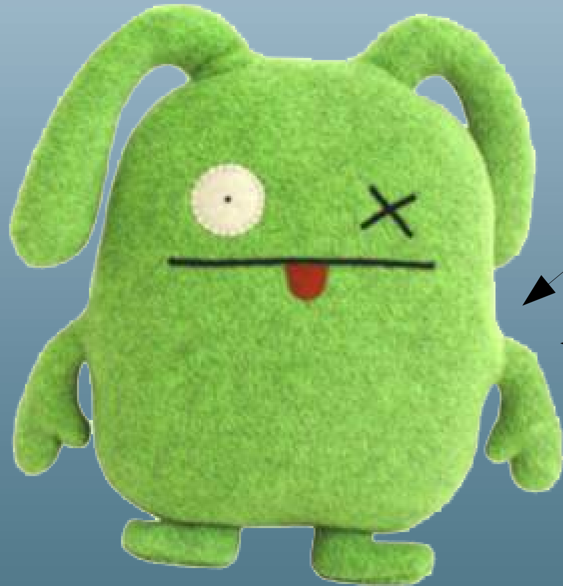
Opis działania:

- CFS zamiast przypisywać do każdego priorytetu kwant czasu, przypisuje do priorytetu wagę, która jest liczona na podstawie wykonywanych procesów. Każdy wątek otrzymuje proporcjonalną ilość czasu do swojego priorytetu podzielonego przez łączną wagę wszystkich wątków. W CFS określany jest czas pełnego obrotu „round-robin” zamiast określania czasu dla priorytetu.

Przykład: Znów mamy 2 procesy i czas tym razem na pełen obrót (a nie na 1 wątek), to 100 ms

Miły wątek - nice=19

Tak samo miły wątek – nice=19



Dostanie 50 ms

50/100 czasu

Dostanie 50ms

50/100 czasu

Completely Fair Scheduler

Opis działania:

- CFS zamiast przypisywać do każdego priorytetu kwant czasu, przypisuje do priorytetu wagę, która jest liczona na podstawie wykonywanych procesów. Każdy wątek otrzymuje proporcjonalną ilość czasu do swojego priorytetu podzielonego przez łączną wagę wszystkich wątków. W CFS określany jest czas pełnego obrotu „round-robin” zamiast określania czasu dla priorytetu.

Przykład: 2 procesy i czas na pełen obrót, to 100 ms

Niemiły wątek - nice=0

Waga = 1024

Waga = 335

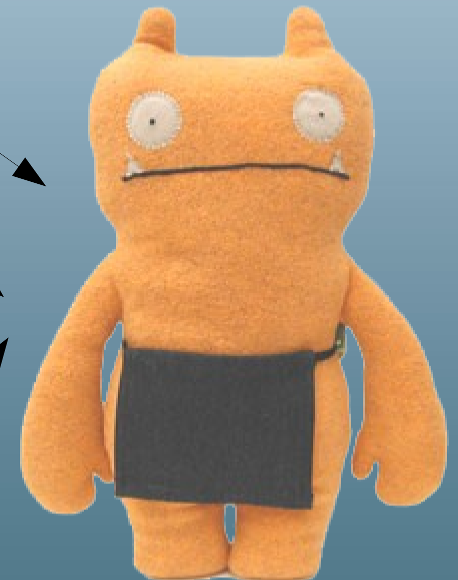
Milszy wątek – nice=5

Dostanie 75 ms

Dostanie 25 ms

1024/(1024+335) czasu

335/(1024+335) czasu



Completely Fair Scheduler

Opis działania:

- CFS zamiast przypisywać do każdego priorytetu kwant czasu, przypisuje do priorytetu wagę, która jest liczona na podstawie wykonywanych procesów. Każdy wątek otrzymuje proporcjonalną ilość czasu do swojego priorytetu podzielonego przez łączną wagę wszystkich wątków. W CFS określany jest czas pełnego obrotu „round-robin” zamiast określania czasu dla priorytetu.

Przykład: 2 procesy i czas na pełen obrót, to 100 ms

Niemiły wątek - nice=5

Waga = 335

Waga = 110

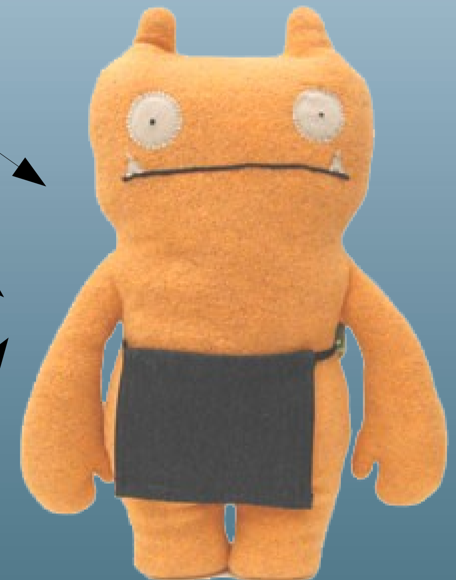
Milszy wątek – nice=10

Dostanie 75 ms

Dostanie 25 ms

335/(335+110) czasu

110/(335+110) czasu



Completely Fair Scheduler

Opis działania:

➢ CFS zamiast przypisywać do każdego priorytetu kwant czasu, przypisuje do priorytetu wagę, która jest liczona na podstawie wykonywanych procesów. Każdy wątek otrzymuje proporcjonalną ilość czasu do swojego priorytetu podzielonego przez łączną wagę wszystkich wątków. W CFS określany jest czas pełnego obrotu „round-robin" zamiast określania czasu dla priorytetu.

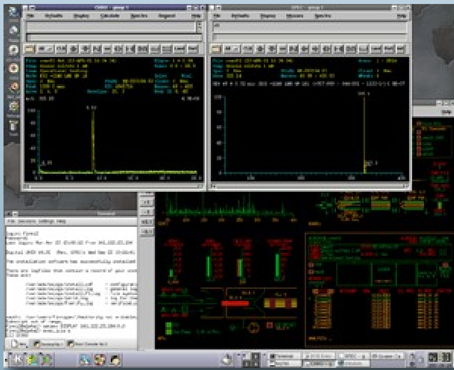
➢ Idea: Śledzić ile czasu każdy wątek był wykonywany względem wagi.
Każdy wątek o priorytecie 0 ma przyznawaną 1ns za każdą nanosekundę wykonywania, natomiast wątek o priorytecie 5 odpowiednio ma przyznawane 3ns (a dokładniej 1024/335 nanosekundy), scheduler stara się zbilansować czas wątków.
Co w takim razie dzieje się gdy przychodzi nowy proces?
Wybiera proces o najmniejszym czasie i wpisuje ten czas do nowoprzybyłego procesu (z pewną zmianą rozróżniającą to czy jest to proces nowoprzybyły, czy obudzony). W ten sposób nowe/obudzone procesy będą miały pierszeństwo, ale nie będą mogły zbyt długo być wykonywane.
Zatem musimy trzymać procesy posortowane względem czasu.
Jest to zrealizowane poprzez drzewo czerwono-czarne.

Completely Fair Scheduler

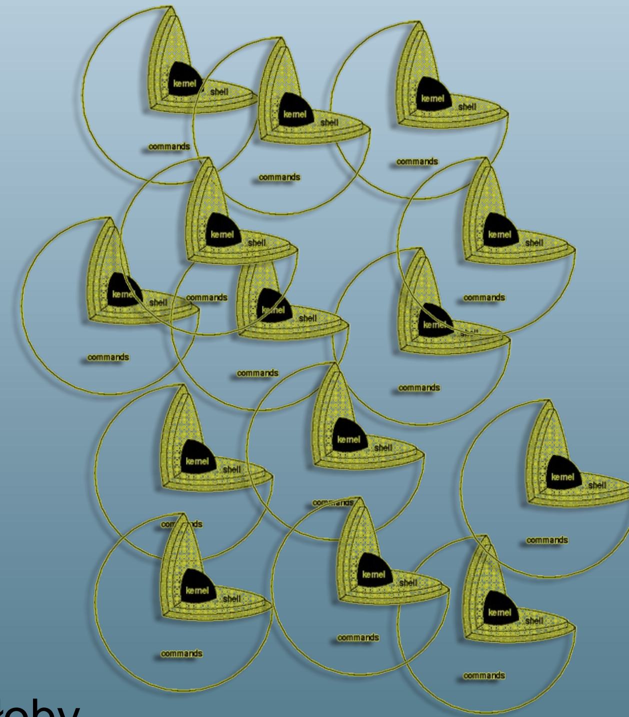
Jedne z możliwych rozszerzeń

Bolek i Lolek korzystają z jednego komputera

Bolek korzysta z X Windows



Lolek kompiluje współbieżnie 49 razy jądro linuxa



Razem – 50 procesów, zatem każdy z nich powinien dostać po 2% czasy procesora...
Bolek nie będzie zadowolony :(

Nie jest to podział sprawiedliwy, sprawiedliwiej byłoby gdyby obaj dostali po 50% czasu procesora.

Completely Fair Scheduler

„Group schedule”

- Hierarchiczny podział na grupy umożliwi równy podział czasu procesora względem grup na tym samym poziomie w drzewie
- Powstaje pojęcie „schedule entity”, czyli scheduler niekoniecznie zarządza wątkiem.
- Standardowo proces jest na szczycie hierarchii i każdy jest zarządzany niezależnie, ale proces może zostać przeniesiony do grupy, w której będzie zarządzany w ramach tej grupy. Proces jest widziany przez zarząd-ce poprzez grupę nadrzędną.
- Kiedy scheduler wykonuje następny wątek i wybierze grupę, to schodzi w dół drzewa i w ramach tej grupy wybiera następny wątek.. tą operację schodzenia po drzewie wykonuje tak długo, aż dostanie się do procesu.

Testy planistów

- Planista z jądra 2.4 – Old
- Planista z jądra 2.6 < 2.6.23 – O(1)
- Planista Completely Fair Scheduler (CFS)
- Planista Rotating Staircase Deadline (RSDL lub SD)

Twórcy planistów

- Con Kolivas
- Ingo Molnar
- Roman Zippel

Jak testować wydajność?

- Hackbench
- lat_ctx
- pipe_test
- wrażenia użytkownika

hackbench

- Twórcą hackbench'a jest Rusty Russell
- Określił on go jako 'chat benchmark'
- Nie korzysta z wątków ani semaphorów
- Hackbench tworzy podaną liczbę grup procesów (20) które wysyłają do siebie wiadomości (100) za pomocą łącz nienazwanych (pipe). Mierzy czas ich wykonania.
- Mierzy skalowalność planisty

pipe_test

- prosty program mierzący czasy zmiany kontekstu
- tworzy dziecko I wymiania z nim po jednej informacji. (czyli 2 odczyty I 2 zapisy)

```c
int main (void)
{
        unsigned long long t0, t1;
        int fd1[2], fd2[2];
        int m = 0, i;

        pipe(fd1);
        pipe(fd2);

        if (!fork()) {
                for (;;) {
                        t0 = GET_TIME();
                        for (i = 0; i < LOOPS; i++) {
                                read(fd1[0], &m, sizeof(int));
                                m = 2;
                                write(fd2[1], &m, sizeof(int));
                        }
                        t1 = GET_TIME();
                        printf("%.2f usecs/loop.\n",
                                (double)(t1-t0)/(double)LOOPS);
                }
        } else {
                for (;;) {
                        m = 1;
                        write(fd1[1], &m, sizeof(int));
                        read(fd2[0], &m, sizeof(int));
                }
        }

        return 0;
}
```

lat_ctx

- część pakietu do testowania systemu LMBENCH
- mierzy czasy zmiany kontekstu
- tworzy pierścień procesów połączonych pipe'ami
- każdy proces czyta z pipe'a, “coś” wykonuje I pisze do kolejnego procesu.
- to “coś” to funkcja bread.
- zjawisko “cache pollution”.

wrażenia użytkownika

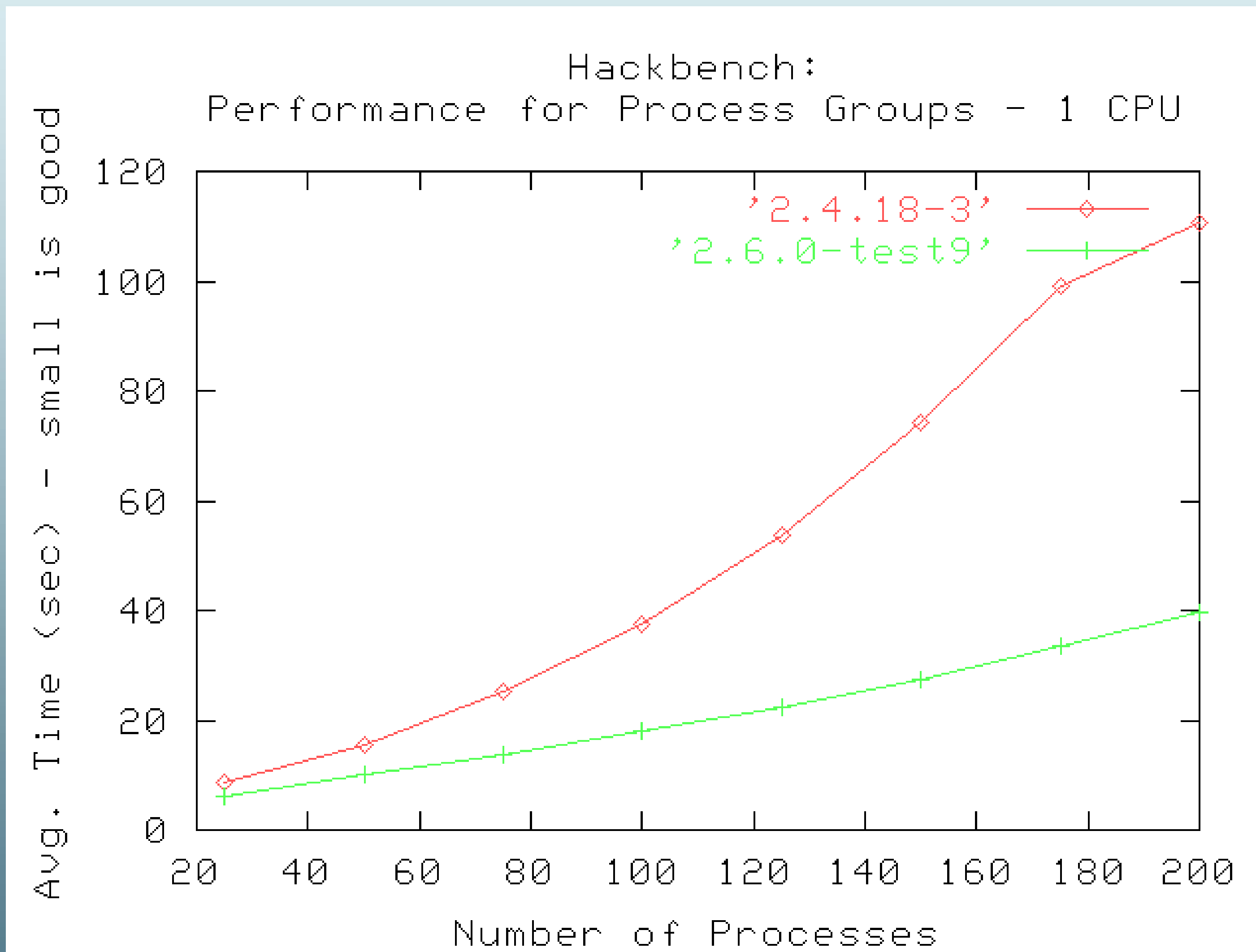
- Jak płynna jest praca z systemem?
- Czy interaktywne aplikacje nie mają opóźnień?
- Jak pracuję się gdy w tle działa “ciężki” proces?

Old vs O(1)

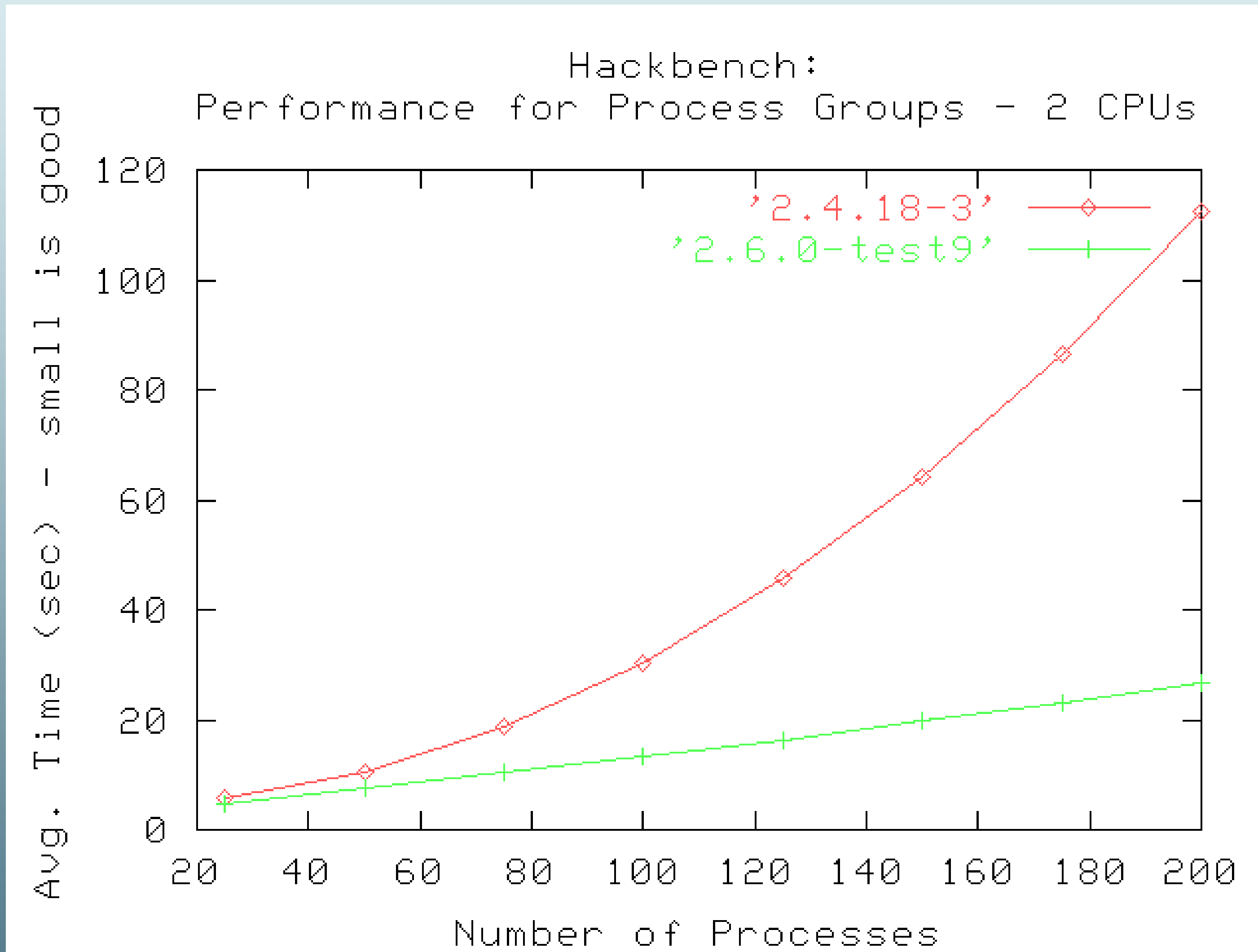
- Szybki dla niedużej liczby procesów.
- Rzadkie przeliczanie priorytetów.
- Brak zagłodzenia.
- Dobra obsługa procesów czasu rzeczywistego.

- Skalowalność
- Wydajność SMP
- Zrównoważenie obciążenia
- Poprawa interaktywności

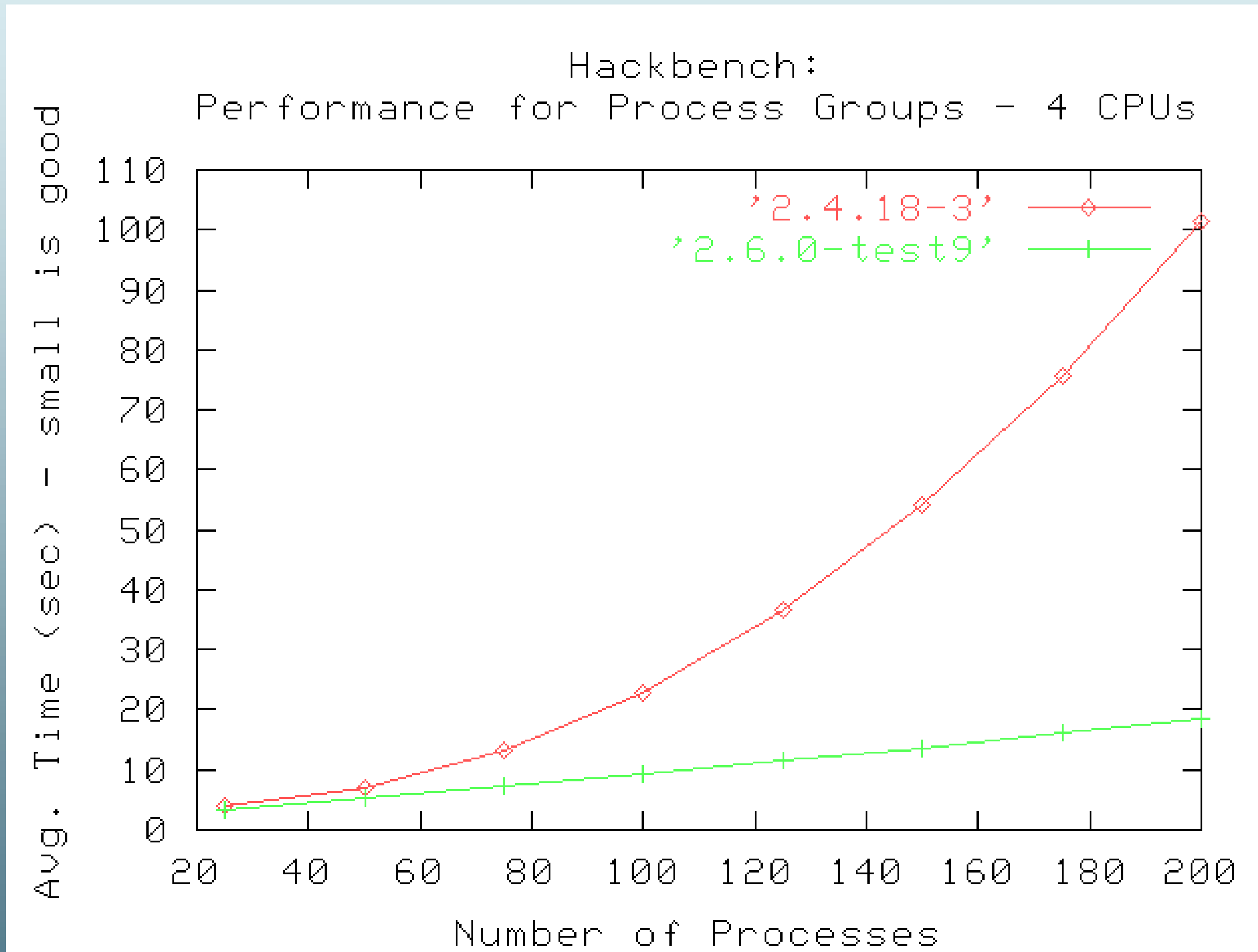
Porównanie Old I O(1)



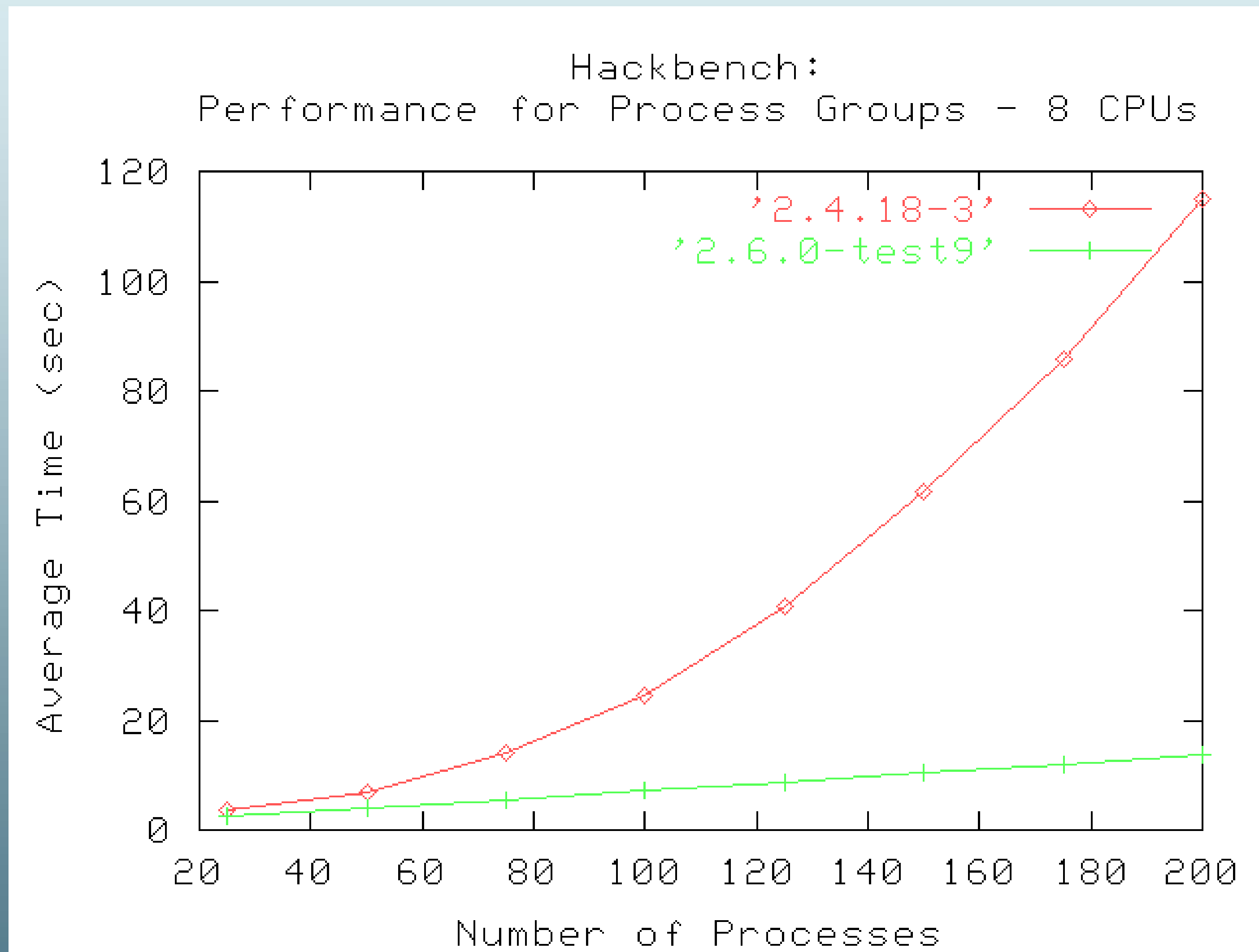
Porównanie Old I O(1)



Porównanie Old I O(1)



Porównanie Old I O(1)

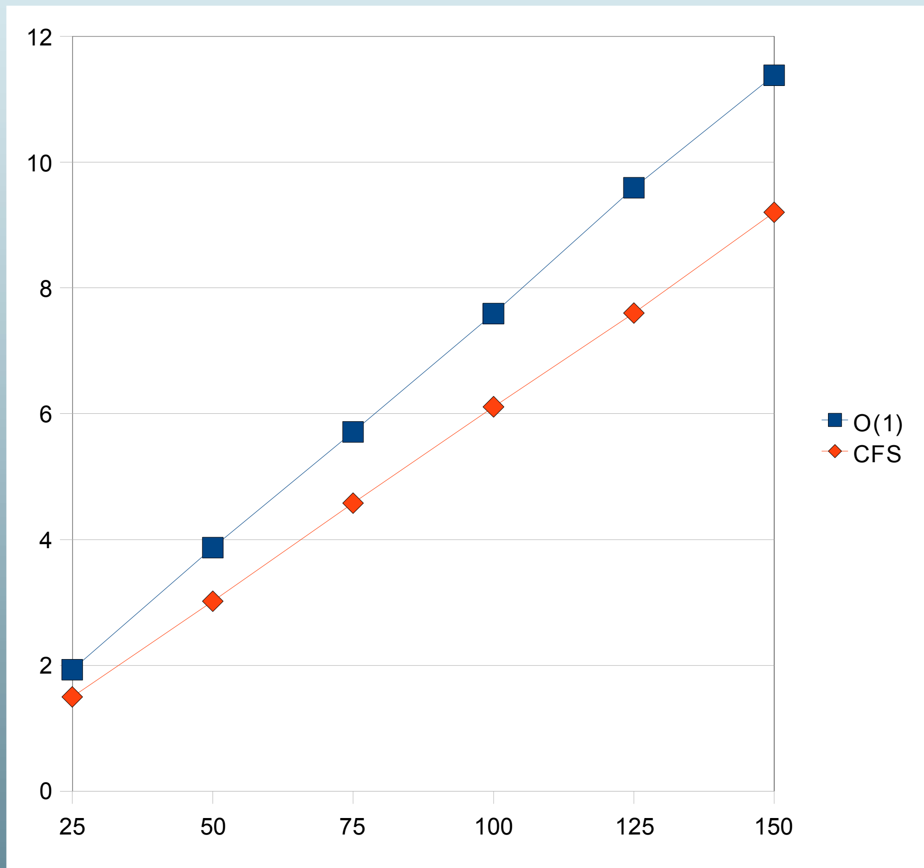


CFS vs O(1)

- Bardziej sprawiedliwy
- Bardziej uniwersalny
- Lepsza skalowalność
- Szybszy
- Efektywne budzenie wątków

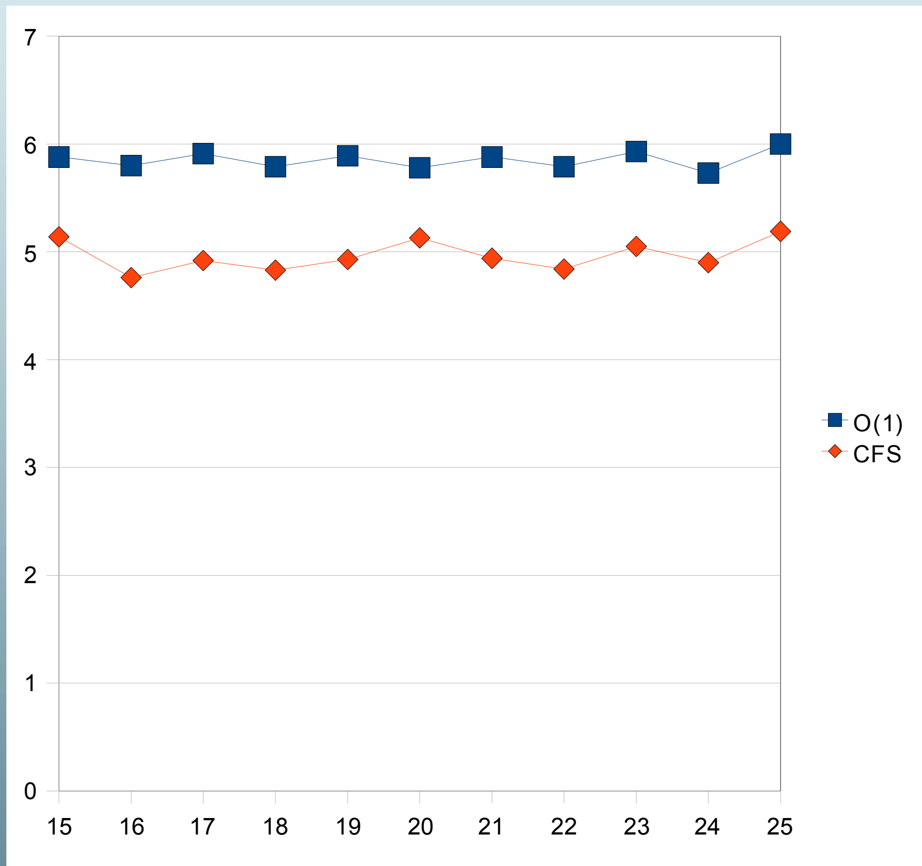
- Jest O(1)?
- Jest na straconej pozycji

CFS vs O(1) hackbench



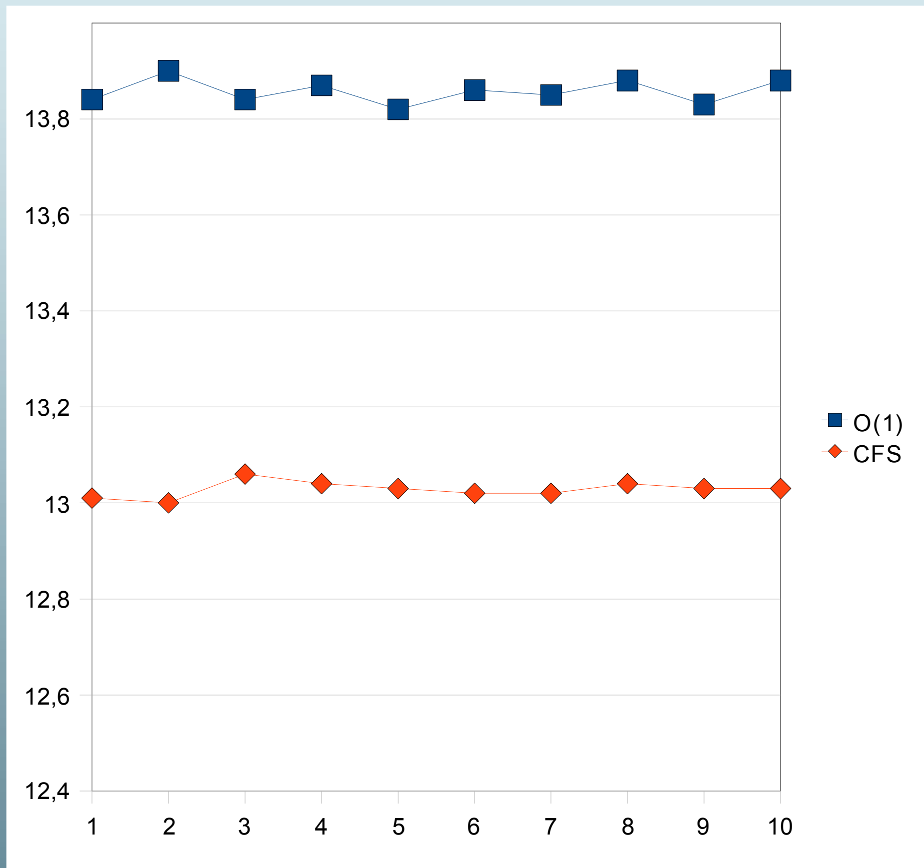
- Widać sporą przewagę CFS nad O(1) sięgającą 15%
- Lepsza skalowalność CFS
- Wartości średnie dla 10 prób

CFS vs O(1) lat_ctx



- Przewagę CFS nad O(1) około 10%
- Szybsze przełączanie kontekstu
- Wartości dla 15-25 procesów

CFS vs O(1) pipe_test



- Ponownie przewaga CFS
- Wartości dla 10 prób

CFS vs O(1)

- Z punktu widzenia normalnego użytkownika, wzrost interaktywności w CFS wobec O(1) jest znaczący - zwłaszcza na słabszych komputerach

O co tyle krzyku?

- Kontrowersje z nowym schedulerem
- Scheduler O(1) jest dość “młody”
- Problem dotyczy wybrania następcy O(1) z dwóch poważnych kandydatów
- CFS vs RSDL
- Con Kolivas rezygnuje z pracy nad jądrem

RSDL

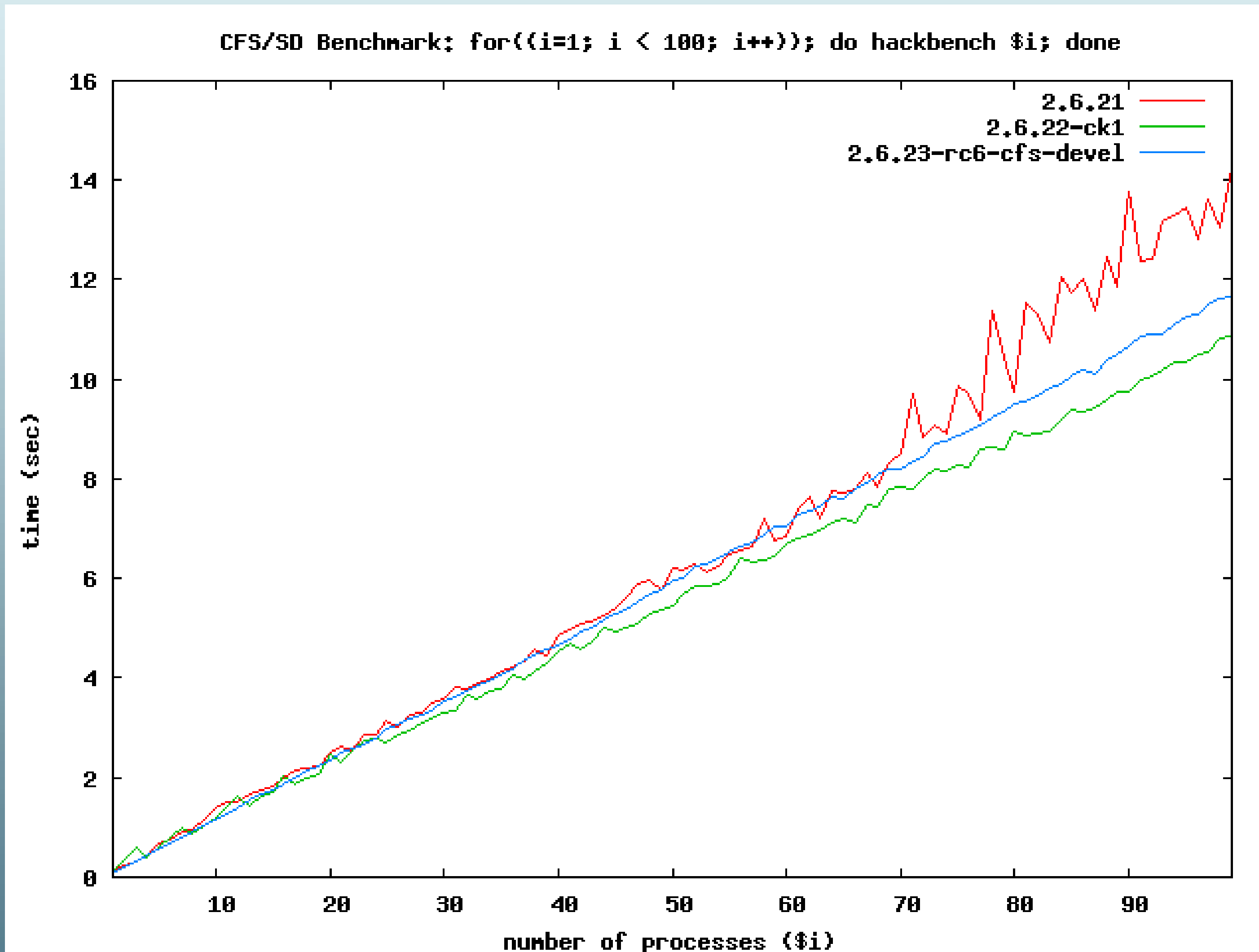
- Działa w O(1)
- Odporny na zagłodzenia
- Brak wsparcia dla interaktywynych procesów
- Niskie czasy oczekiwania
- Pomimo braku wsparcia, wysoka interaktywność

CFS vs RSDL

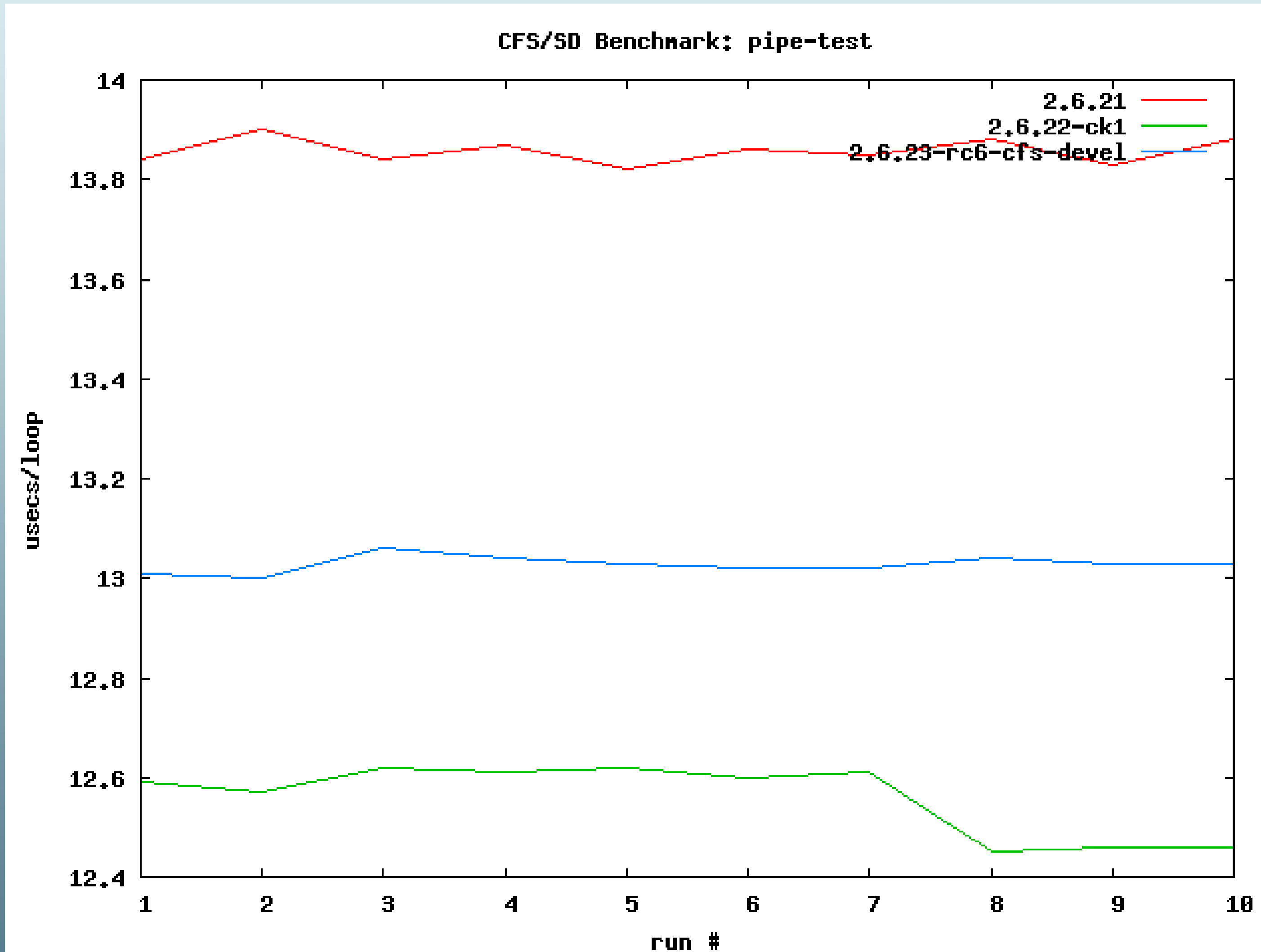
- Niewiele wolniejszy od RSDL
- Wsparcie wielu znaczących programistów
- Bardziej uniwersalny

- W wielu pojedynczych testach jest szybszy
- Działa w O(1)

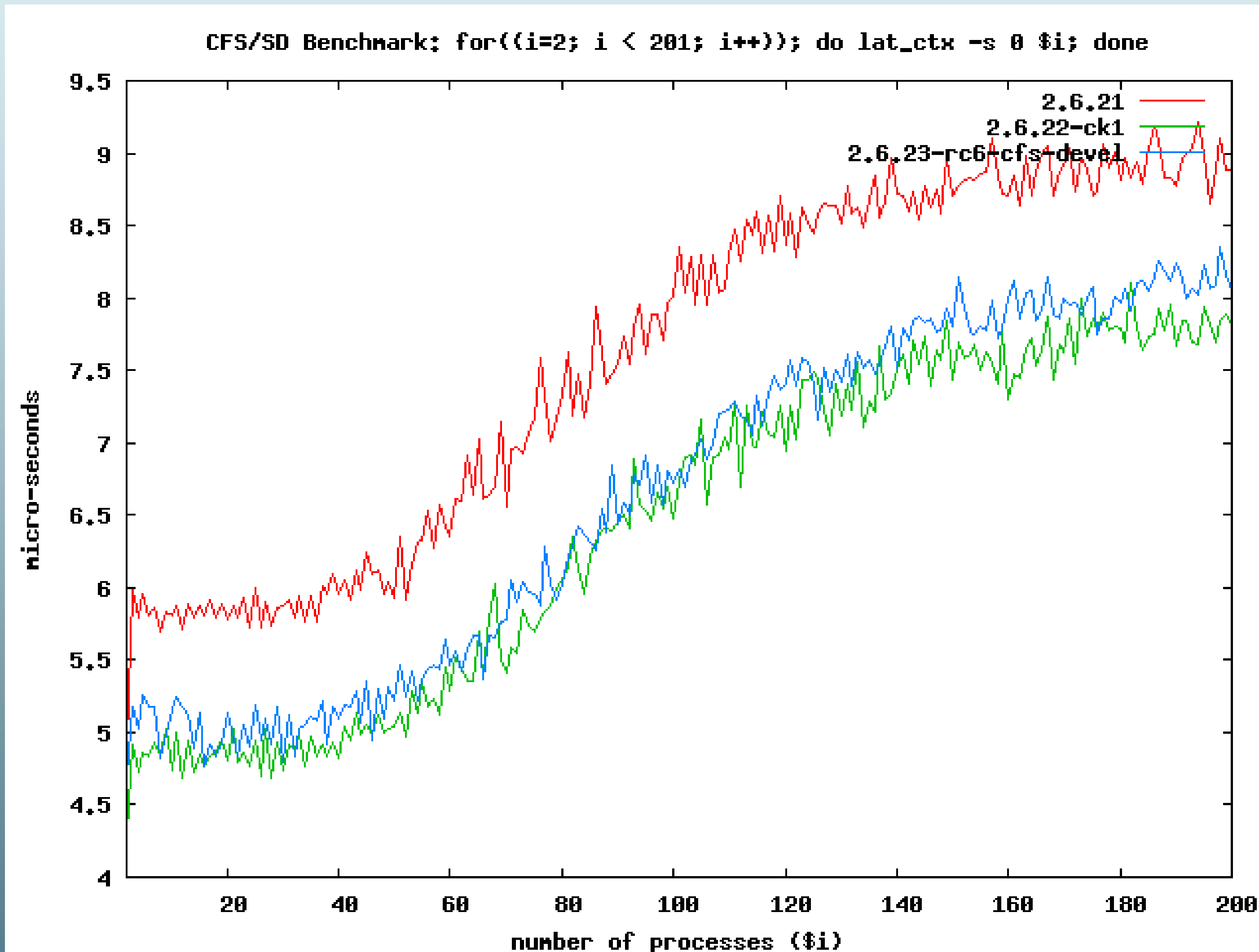
CFS vs RSDL



CFS vs RSDL



CFS vs RSDL



Rodzaj planisty jako opcja?

- Rozdzielenie planistów na do serwerów I do desktopów
- Umożliwienie użytkownikowi wyboru planisty, który bardziej spełnia jego oczekiwania

- Użytkownicy są za głupi, nie powinni podejmować takich decyzji więc decydujemy za nich.
- Linus uważa, że planista powinien być taki sam na desktopie I na serwerze.

Planista przyszłości - PiS

- Niewielkie koszty funkcjonowania
- Mozliwie rzadkie przełączanie kontekstu
- Zapewnienie interaktywności
- Efektywność, także przy dużym obciążeniu
- Niedopuszczenie do zagłodzenia
- Efektywna obsługa systemów wieloprocesorowych
- Jak najlepsza obsługa procesów czasu rzeczywistego

Przydatne linki

- http://ck.wikia.com/wiki/RSDL
- http://kerneltrap.com
- http://people.redhat.com/mingo/cfs-scheduler/