Szeregowanie procesów w Linuxie - trendy rozwojowe

Gabriel Kłosiński   Bartosz Łoś

6 grudnia 2007

Wstęp, Algorytmy planowania

Scheduler 2.4

Scheduler O(1)

Sprawiedliwe Schedulery: RSDL, CFS

Testy

Co to jest szeregowanie procesów?

- Problem szeregowania procesów to... problem algorytmiczny!!!

Algorytm szeregowania (ang. scheduler - planista) to algorytm rozwiązujący jedno z najważniejszych zagadnień informatyki - jak rozdzielić czas procesora i dostęp do innych zasobów pomiędzy zadania, które w praktyce zwykle o te zasoby konkurują.

Definicja

Mamy zbiór $J = \{J_1, ..., J_n\}$ n zadań do wykonania.
Dla jednego procesora jest to funkcja:

$$\sigma : \mathbb{R}^+ \rightarrow \mathbb{N}$$

taka że:

$$\forall t \in \mathbb{R}^+ \exists t_1, t_2 : [t \in \langle t_1, t_2) \wedge \forall t' \in \langle t_1, t_2) : \sigma(t) = \sigma(t')]$$

Własności algorytmów planowania

- Wykorzystanie procesora
- Przepustowość (liczba procesów kończonych w jednostce czasu)
- Czas cyklu przetwarzania (czas upływający między chwilą nadejścia procesu do systemu a chwilą zakończenia procesu)
- Czas oczekiwania (czas który proces spędza w kolejce procesów gotowych do wykonania)
- Czas odpowiedzi (czas upływający między przedłożeniem zamówienia a pojawieniem się pierwszej odpowiedzi)

Różne algorytmy planowania przydziału procesora

- **Planowanie metodą FCFS (First-Come - First Served "pierwszy nadszedł - pierwszy obsłużony")**

  Algorytm niewywłaszczający. Po objęciu kontroli nad procesorem proces utrzymuje ją do czasu, aż sam zwolni procesor wskutek albo zakończenia swego działania, albo żądania operacji wejścia-wyjścia.

Różne algorytmy planowania przydziału procesora

- **Planowanie metodą SJF (Shortest Job First "najpierw najkrótsze zadanie")**

  Gdy procesor staje się dostępny, wówczas zostaje przydzielony procesowi mającemu najkrótszą następną fazę procesora. Nie jesteśmy w stanie poznać długości następnej fazy procesora, lecz możemy spróbować oczacować jej wartość na podstawie długości faz poprzednich. Algorytm SJF może być wywłaszczający lub niewywłaszczający.

Różne algorytmy planowania przydziału procesora

- **Planowanie priorytetowe**

  Priorytety mogą być definiowane wewnętrznie (np. limity czasu, wielkość obszaru wymaganej pamięci, liczba otwartych plików, stosunek średniej fazy wejścia-wyjścia do średniej fazy procesora) albo zewnętrznie (np. ważność procesu). Planowanie priorytetowe może być wywłaszczające lub niewywłaszczające. Problemem z nieskończonym blokowaniem niskopriorytetowych procesów rozwiązuje się poprzez postarzanie procesów.

Różne algorytmy planowania przydziału procesora

- **Planowanie rotacyjne (ang. round-robin)**

  Planista przegląda kolejkę i każdemu procesowi przydziela odcinek czasu. W przypadku, gdy proces wykorzysta swój kwant czasu nastąpi przerwanie zegarowe systemu operacyjnego. Dokona się przełączenie kontekstu i proces zostanie odłożony na koniec kolejki procesów gotowych, planista zaś wybierze następny proces z kolejki

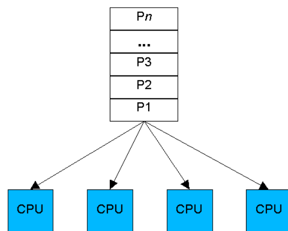
Różne algorytmy planowania przydziału procesora

- **Wielopoziomowe planowanie kolejek**

  Algorytm wielopoziomowego planowania kolejek rozdziela kolejkę procesów gotowych na osobne kolejki. Procesy zostają na stałe przypisane do jednej z tych kolejek na podstawie np. typu procesu lub rozmiaru pamięci. Każda kolejka ma własny algorytm planujący.

Scheduler w wersji jądra 2.4

Scheduler w wersji jądra 2.4 jest dość prosty. Mamy jedną wspólną kolejkę dla wszystkich procesów gotowych (tasklist). Każdemu procesowi w kolejce przypisany jest priorytet (goodness rating).



Obliczenie priorytetów procesów

Funkcja **goodness()** wykorzystywana przez funkcję **schedule()**:

- Dla procesu, który zgłosił chęć oddania procesora (ustawiony bit SCHED_FIELD) zwraca wartość -1
- Dla procesu czasu rzeczywistego (SCHED_FIFO i SCHED_RR) zwraca wartość 1000 + rt_priority
- Dla zwykłych procesów (SCHED_OTHER), którym zakończył się już kwant czasu w tej epoce ($counter == 0$) zwraca wartość 0
- Dla zwykłych procesów zwraca wartość równą ilości taktów, które pozostały procesowi do wykorzystania w tej epoce powiększoną o wartość $20 - nice$.

Opis algorytmu szeregowania procesów dla jądra 2.4

W momencie, gdy procesor zostanie zwolniony, scheduler 2.4 przegląda kolejkę (tasklist), szukając procesu o najwyższym priorytecie i wybiera proces, który następnie zostaje uruchomiony. Kolejka ta nie jest w żaden sposób uporządkowana, zatem każda iteracja scheduler'a wymaga przeglądania całej listy procesów, w celu znalezienia najlepszego kandydata dla konkretnego procesora. Zatem algorytm wyboru kolejnego zadania działa w czasie liniowym. Po zakończeniu epoki, jeden procesor wykonuje przeliczanie priorytetów, a inne procesory czekają na zakończenie tej operacji

Pola task_struct istotne dla scheduler'a 2.4

- $p->counter$ - "dynamiczny priorytet" procesu
- $p->priority$ - statyczny priorytet procesu, zmiana: nice, sched_setparam (POSIX.1b) oraz setpriority (4.4BSD/SVR4)
- $p->rt_priority$ - priorytet czasu rzeczywistego
- $p->policy$ - klasa szeregowania procesu, zmiana: sched_setscheduler, możliwe wartości: SCHED_OTHER, SCHED_FIFO i SCHED_RR, jeśli podano OR SCHED_YIELD to proces może zostać wywłaszczony, np. przez sched_yield

Algorytm schedule() dla scheduler'a 2.4

Najważniejsze zmienne stosowane przez schedule():

- prev - deskryptor procesu, który do tej pory używał procesora
- next - na tą zmienną musi zostać zapisany wskaźnik do deksryptora procesu, który otrzyma procesor w wyniku szeregowania
- c - priorytet znalezionego do tej pory najlepszego kandydata na przydział procesora

Algorytm schedule() dla scheduler'a 2.4

- Jeżeli jesteśmy w obsłudze przerwania to błąd.
- Jeżeli poprzedni proces był procesem czasu rzeczywistego Round Robin przerzuca się go na koniec kolejki oraz jeżeli potrzeba odnawia kwant czasu.
- Jeżeli stan procesu jest różny od TASK_RUNNING to znaczy, że proces ma zostać zablokowany (tzn. usunięty z runqueue). Jednak jeżeli proces jest w stanie TASK_INTERRUPTIBLE i ma jakieś nie obsłużone sygnały, to daje się mu szansę je obsłużyc.
- Czyszczenie pola need_resched:

$$prev->need_resched = 0;$$

Algorytm schedule() dla scheduler'a 2.4

- Pierwszym kandydatem na przydział procesora jest proces idle Ma on jednak bardzo niski priorytet - każdy inny proces z kolejki runqueue będzie miał wyższy priorytet

  next = idle_task(this_cpu); c = -1000;

- Jeżeli proces ostatnio wykonywany jest dalej w trybie TASK_RUNNING to jest wybierany jako pierwszy kandydat do procesora zamiast procesu idle.
- Przeszukanie kolejki runqueue w poszukiwaniu procesu o najwyższym priorytecie. Procesowi temu zostanie przydzielony procesor.

Algorytm schedule() dla scheduler'a 2.4

- Sprawdzenie czy nie zakonczyła się epoka. Jeśli tak, to przydzielenie wszystkim procesom nowych kwantów czasu.
- Jeżeli nowowybrany proces jest tym samym procesem, który do tej pory był aktywny, to nic więcej już nie trzeba robić poza wyzerowaniem flagi SCHED_YIELD.
- Rozpoczęcie przełączania procesów

Porównanie scheduler'a z wersji 2.6 z 2.4

Opisany scheduler 2.4 działa według następującego algorytmu:

1. weź pierwszy proces z brzegu
2. sprawdź, jak bardzo pilne jest wykonanie
3. jeśli bardziej, niż najpilniejszego do tej pory, zapamiętaj go
4. jeśli jest jeszcze jakiś niesprawdzony proces, skocz do punktu pierwszego
5. uruchom najpilniejszy proces

Porównanie scheduler'a z wersji 2.6 z 2.4

Scheduler 0(1) ma jedną podstawową cechę: cały czas śledzi wszystkie procesy i ma ich listę, na której znajdują się priorytety poszczególnych zadań. Algorytm jest następujący:

1. pobierz priorytet najbardziej "pilnego" procesu
2. pobierz z listy pierwszy proces, który ma taki priorytet
3. uruchom zadanie

Skalowalność na maszyny wieloprocesorowe: stworzono odrębne kolejki dla każdego procesora

Podstawowe fakty

- Pełna nazwa "Nowy Skalowalny Algorytm Szeregowania Procesów"
- Twórcą schedulera jest Ingo Molnar
- 7200 linii kodu
- Osobna kolejka dla każdego procesora
- Wywłaszczanie w trybie jądra
- Zdecydowana poprawa wydajności w porównaniu z O(n)

Wady O(1)

- Przydziela zbyt mało czasu procesora na takie zadania, jak wyświetlanie interfejsu użytkownika, czy odtwarzanie muzyki.
- Obserwowano przypadki, w których nowy scheduler sobie nie radzi
- Przydzielanie czasu procesora odbywało się w sposób nieracjonalny
- Kontynuowano pracę nad poprawą schedulera O(1) - Con Kolivas

Rozwiązanie

Con Kolivas:

- Przyczyną problemów w poprzednich schedulerach jest "estymator interaktywności"
- Estymator nie działa poprawnie we wszyskich przypadkach
- Koniecznym stała się zmiana koncepcji budowy schedulera

Podstawowe informacje

- Nowa koncepcja traktowania procesów - wszystkie procesy traktujemy równo
- Wyrzucamy estymator interaktywności
- Każdy proces dostaje taki sam kwant czasu
- Con tworzy RSDL (Rotating Staircase Deadline Scheduler)
- Osiąga sukces - zdecydowana poprawa wydajności
- Radzi sobie nawet w najtrudniejszych przypadkach
- Planowany do włączenia do jądra linuksa

Podstawowe informacje

- Kolejny "sprawiedliwy scheduler"
- Pojawił się znikąd
- Twórcą był Ingo Molnar
- Pierwsza wersja po 62h pracy - 100K patch (Obecnie 290K)
- Wywołał poruszenie i gorącą dyskusję w środowisku twórców linuksa

Koncepcja sprawiedliwości

- Pomysł zaczerpnięty od Cona Kolivasa i RSDL-a
- Idealne rozwiązanie to takie, w którym procesor dzieli woją moc po równa na każdy proces
- Idealny scheduler powinien dążyć do zachowania takiej równowagi

Praktyka

- Zamiast list procesów zaimplementowano drzewa czerwono - czarne
- Powstałe opóźnienia odnotowywane i śledzone przez p->wait_runtime
- Wielkość rq->fair_clock mierzy, ile czasu procesora wykorzysta dany proces (pomocne do liczenia liczenia czasu, jaki powinien być przydzielony innym procesom)
- Kluczem w drzewie jest wartość rq->fair_clock - p->wait_runtime

Praktyka, ciąg dalszy

- Podczas przydzielania czasu procesora bierzemy pod uwagę czas uśpienia procesu
- Wyliczanie czasu działania procesu w nanosekundach (we wcześniejszych schedulerach - jiffies)
- Czas jest przydzielany dynamicznie, a nie statycznie
- Możliwość dopasowywania stopnia interaktywności
- Podział na moduły

Najważniejsze funkcje

- void (*enqueue_task) (struct rq *rq, struct task_struct *p);
- void (*dequeue_task) (struct rq *rq, struct task_struct *p);
- void (*requeue_task) (struct rq *rq, struct task_struct *p);
- void (*task_new) (struct rq *rq, struct task_struct *p);
- void (*task_init) (struct rq *rq, struct task_struct *p);
- void (*task_tick) (struct rq *rq, struct task_struct *p);

Najważniejsze funkcje, ciąg dalszy

- void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);
- struct task_struct * (*pick_next_task) (struct rq *rq);
- void (*put_prev_task) (struct rq *rq, struct task_struct *p);
- struct task_struct * (*load_balance_start) (struct rq *rq);
- struct task_struct * (*load_balance_next) (struct rq *rq);

Wyniki testów porównujących scheduler'a 2.6 z 2.4

Na stonie internetowej;

http://devresources.linux-foundation.org/craiger/hackbench/

przedstawiono testy ukazujące znaczącą poprawę, nie tylko w efektywnym szeregowaniu procesów dla systemu z jednym procesorem, ale także w lepszą skalowalność planisty na systemy wieloprocesorowe. Wskazuje się także na lepsze zużycie zasobów oraz pokazano, iż w wersji 2.6 z domyślnymi ustawieniami można uruchamiać znacznie więcej procesów w tym samym czasie.

Jak przeprowadzono testy?

Co to jest hackbench?
Tworzymy n procesów pisarzy oraz n procesów czytelników. Każdy pisarz pise 100 wiadomości do wszystkich czytelników. Każdy pisarz pisze zatem $100 * n$ wiadomości, a czytelnik odbiera $100 * n$ wiadomości.

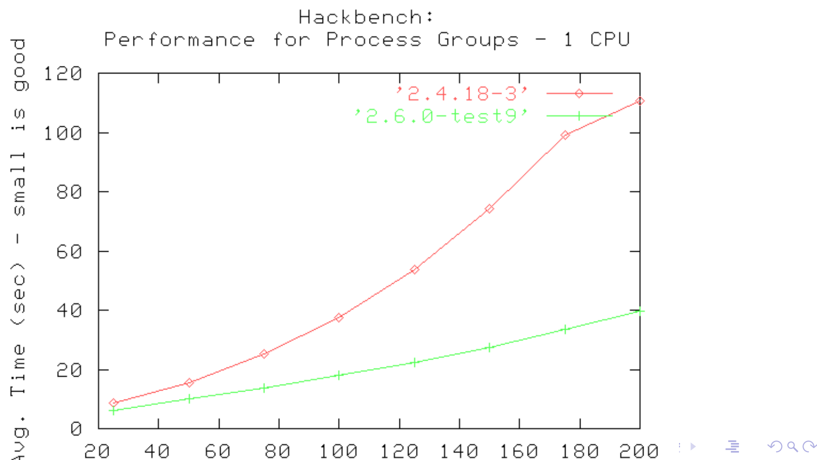
Przedstawianie wyników
Początkowo każdy test uruchamia się dla 25 procesów, zwiększając ich liczbę o kolejną wielokrotność 25 w każdym kroku, aż do maksymalnej wielkości ustawionej przez testera. Każdy zbiór procesów jest wykonywany 5 razy. Dla każdego zbioru procesów podaje się średni czas do zakończenia testu. Wyniki przedstawia się dla każdego zbioru procesów.

Jak przeprowadzono testy?

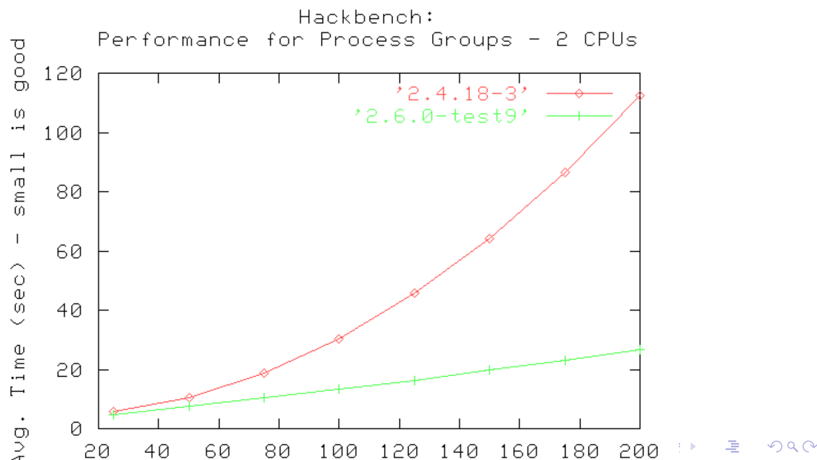
Testy przeprowadzono na następujących systemach:

- system jednoprocesorowy:
  CPU: 1GHz Pentium III w/ 256k L2 cache
  Memory: 512MB
- system z dwoma procesorami:
  CPU: 850MHz Pentium III w/ 256k L2 cache x2 (Coppermine)
  Memory: 1GB
- system z czterema procesorami:
  CPU: 700 MHz Pentium III w/ 1024k L2 cache x4 (Cascades)
  Memory: 4GB
- system z ośmioma procesorami
  CPU 700 MHz Pentium III w/ 1024k L2 cache x8 (Cascades)
  Memory: 8GB

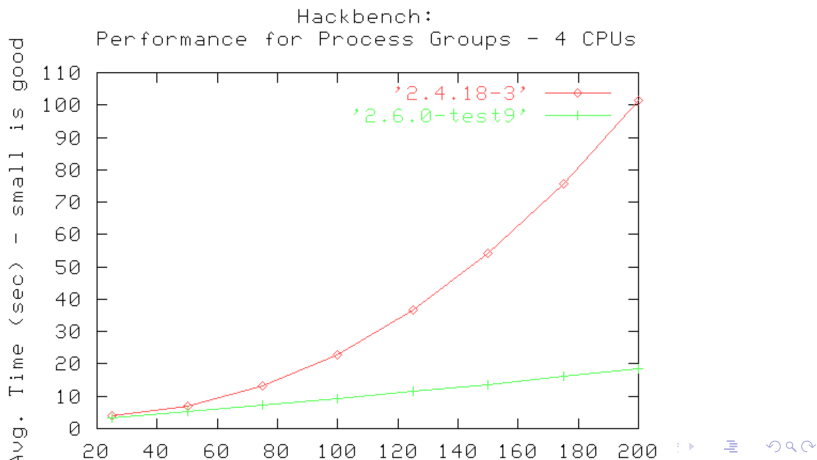
Obserwacja 1: 2.6.0 is bardziej wydajny niż 2.4.18



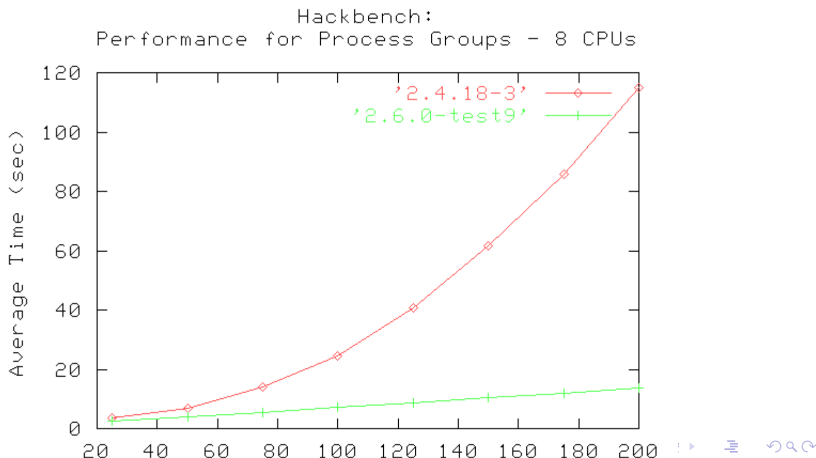
Obserwacja 1: 2.6.0 is bardziej wydajny niż 2.4.18



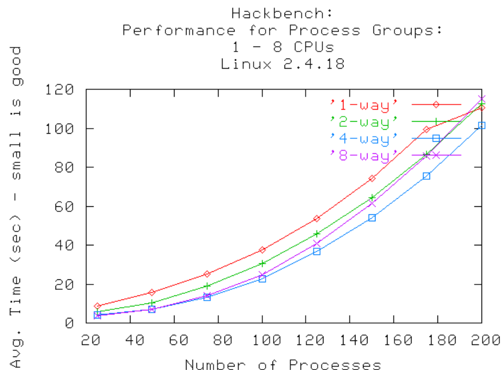
Obserwacja 1: 2.6.0 is bardziej wydajny niż 2.4.18



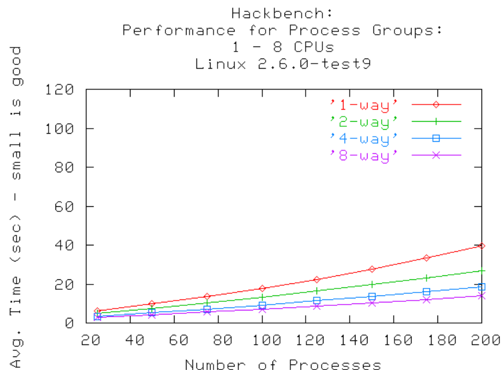
Obserwacja 1: 2.6.0 is bardziej wydajny niż 2.4.18



Obserwacja 2: 2.6.0 ma lepszą skalowalność na systemy wieloprocesorowe niż 2.4.18



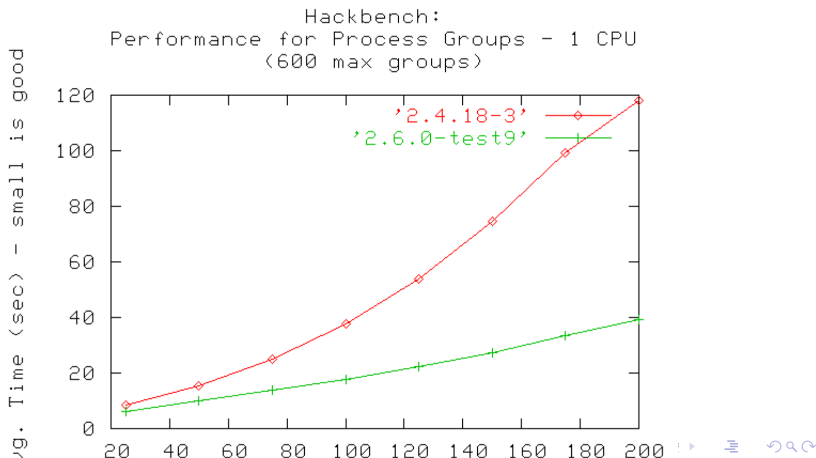
Obserwacja 2: 2.6.0 ma lepszą skalowalność na systemy wieloprocesorowe niż 2.4.18



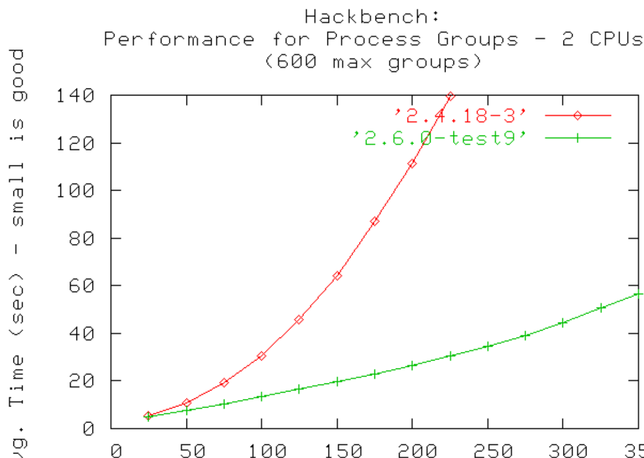
Obserwacja 3: 2.6.0 ma bardziej wydajne zużycie zasobów

Następna grupa testów miała na celu określenie maksymalnej liczby procesów, które mogą być wykonywane do momentu w którym zasoby systemu zostaną wyczerpane. Testy uruchamiano dla coraz większej maksymalnej liczby grup procesów do momentu, w którym system zgłosił błąd związany z brakiem zasobów.

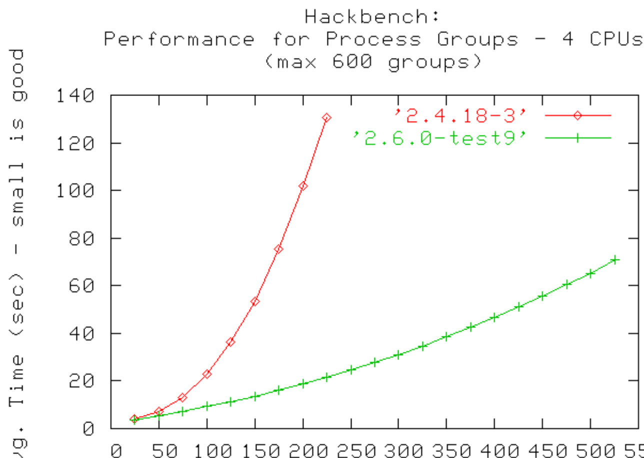
Obserwacja 3: 2.6.0 ma bardziej wydajne zużycie zasobów



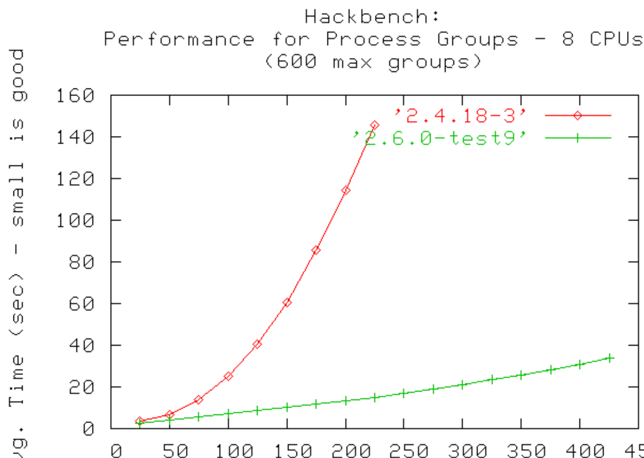
Obserwacja 3: 2.6.0 ma bardziej wydajne zużycie zasobów



Obserwacja 3: 2.6.0 ma bardziej wydajne zużycie zasobów



Obserwacja 3: 2.6.0 ma bardziej wydajne zużycie zasobów



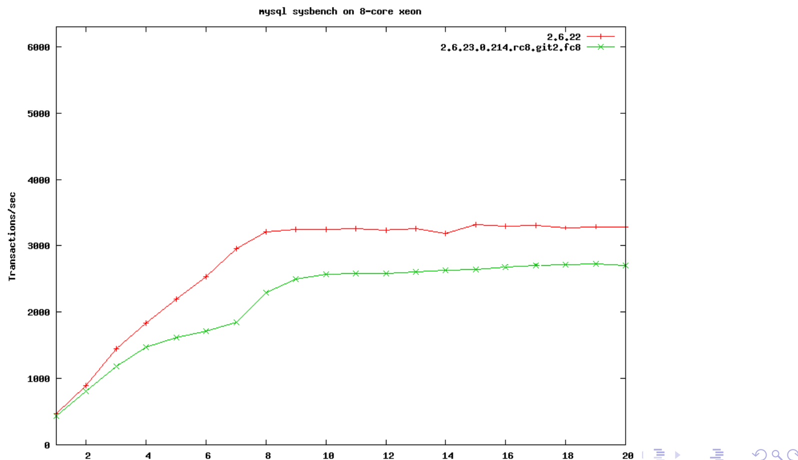
RSDL a CFS

Dyskusja o tym, który z planistów: RSDL czy CFS jest lepszy pojawiała się na wielu listach dyskusyjnych użytkowników linuksa. Między innymi na stronie internetowej:

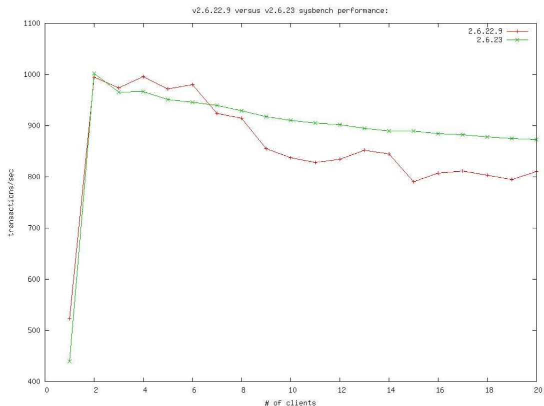
www.kerneltrap.org/CFS

przedstawiano różne wyniki świadczące o wyższości jednego schedulera nad drugim.

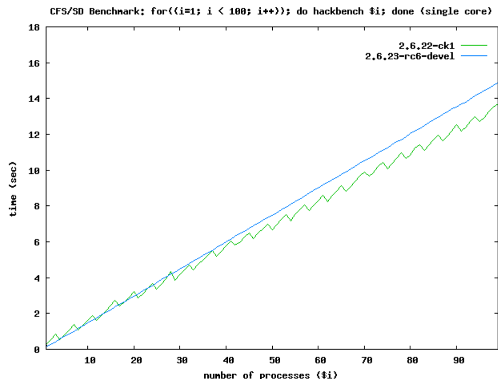
Wyniki testów na serwerze MySQL (Nicholas Miel)



Wyniki testów na serwerze MySQL (Ingo Molnar)



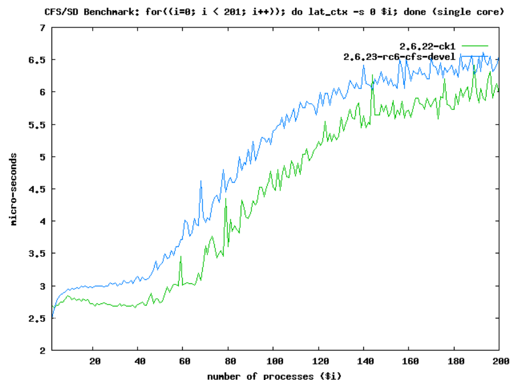
Testy przy użyciu hackbench



Testy mierzące czas przełączania kontekstu

lat_ctx
Mierzy czas przełączania kontekstu dla podanej liczby procesów i wielkości czasu. Tworzy *n* proxesów i łączymy je za pomocą łącz nienazwanych (pipe) w pierścień. Processy przekaziją sobie żeton. Proces odbiera żeton i wykonuje pewnien kod (parametr testu), następnie przekazuje żeton następnemu procesowi w pierścieniu.

Testy mierzące czas przełączania kontekstu



CFS a gry 3D

Niektóre zgłaszane obawy na temat CFS dotyczyły tego, iż może on radzić sobie gorzej z grami 3D niż SD. Ingo Molnar przedstawił wyniki testów, świadczące o tym, iż poprawiony CFS działa nie gorzej lub nawet lepiej niż SD.

- Porównanie: 2.6.22-ck1 (SD) z 2.6.22-cfsv19 (CFS)
- Gra: Quake 3 Arena Demo under Wine (0.9.41)
- Przedstawiono zależność między liczba zadań w tle a liczbą klatek na sekundę.

CFS a gry 3D