Szeregowanie procesów w Linuksie - trendy rozwojowe

Szymon Gwóźdź
Tomasz Klukowski
Przemysław Kosiak

4.12.2007

Plan prezentacji

Scheduler w jądrze 2.4

Scheduler w jądrze 2.4

Pola w task structure

- need_resched - określa czy schedule() powinno być wywołane przy “następnej okazji”
- counter - “dynamiczny priorytet”; liczba tików procesora pozostałych temu wątkowi w tym “scheduling slice”. Kiedy wartość tego pola staje sie mniejsza lub równa 0 to jest resetowana na 0 i p->need_resched jest ustawiane. Proces może sobie zmieniać wartość tego pola.
- priority - statyczny priorytet; zmienialny tylko poprzez wywołania systemowe np. nice()

Pola w task structure

- policy - określa do której “scheduling class” należy to zadanie
  - SCHED_OTHER - traditional UNIX process
  - SCHED_FIFO - POSIX.1b FIFO realtime process
  - SCHED_RR - POSIX round-robin realtime process
- mm - obszar pamięci używany przez proces

Pola w task structure

- processor - wskazuje, który procesor był ostatnio używany przez ten proces
- cpus_runnable, cpus_allowed - cpus_runnable ma wartość różną od zera jeśli proces nie jest uruchomiony na żadnym procesorze . Pola te służą do wnioskowania, czy proces może zostać uruchomiony na danym procesorze przy pomocy makra can_schedule:

```
#define can_schedule(p,cpu) \
        ((p)->cpus_runnable & (p)->cpus_allowed & (1 << cpu))
```

Schedule() - algorytm

- Sprawdzamy czy przypadkiem nie zachodzi current->active_mm=NULL. Jeśli tak to bląd.
- Ustawiamy zmienną prev na aktualny proces, a zmienną this_cpu na aktualny procesor.
- Sprawdzamy czy schedule() zostało wywołane przez "interrupt handler- jeśli tak to zgłaszamy bląd.
- Zwalniamy globalną blokadę jądra (release_kernel_lock)
- Inicjalizujemy sched_data na wskaźnik do pola sched_data procesora.

Schedule() - algorytm

- Zakładamy spin-locka runqueue_lock
- Jeśli prev jest procesem typu SCHED_RR oraz prev->counter==0 to przesuwamy prev na koniec runqueue
- Sprawdzamy czy prev jest w stanie TASK_INTERRUPTIBLE. Jeśli tak jest to: jeśli proces jest w trakcie przetwarzania sygnału to zmieniamy stan na TASK_RUNNING, w przeciwnym wypadku usuwamy prev z runqueue

Schedule() - algorytm

- Zmienna next jest ustawiana na idle task tego procesora. Dobroć tego zadania jest ustawiana na -1000 w nadziei, że znajdzie sie jakiś lepszy wątek
- Dla każdego procesu z runqueue, który może być wykonany na tym procesorze, obliczana jest dobroć. Wygrywa proces z najwyższą dobrocią. Zwycięzca jest na zmiennej next

Goodness()

```
static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{
        int weight;

        /*
         * aktualny proces ma dobroc nizsza od kazdego
         * innego procesu poza idle
         */
        weight = -1;
        if (p->policy & SCHED_YIELD)
                goto out;

        /*
         * Jesli nie jest to proces czasu rzeczywistego:
         */
        if (p->policy == SCHED_OTHER) {
                /*
                 * Pierwsze przyblizenie wartosci dobroci
                 * na podstawie ilosci tikow, ktore mu pozostaly do wykonania
                 *
                 * Nie wykonujemy zadnych innych obliczen jesli "time slice"
                 * dla tego procesu sie skonczyl
                 */
                weight = p->counter;
                if (!weight)
                        goto out;
```

Goodness()

```
#ifdef CONFIG_SMP
            /* Daje pewna przewage procesom dzialajacym ostatnio na tym procesorze           */
            /* (jest to rownowazne karaniu procesow dzialajacych na innych procesorach) ...  */
            if (p->processor == this_cpu)
                weight += PROC_CHANGE_PENALTY;
#endif

            /* .. i niewielka przewage dla procesow korzystajacych z tego samego           */
            /* obszaru pamieci (lub nie korzystajace z pamieci)                            */
            if (p->mm == this_mm || !p->mm)
                weight += 1;
            weight += 20 - p->nice;
            goto out;
        }

        /*
         * Proces czasu rzeczywistego - bierzemy proces
         * o najwyzszej wartosci rt_priority
         * Procesy czasu rzeczywistego maja bezwzgledne pierwszenstwo nad innymi procesami
         */
        weight = 1000 + p->rt_priority;
out:
        return weight;
}
```

Schedule() - algorytm

- Jeśli otrzymana wartość dobroci jest równa 0 wtedy cała lista procesow (nie tylko runqueue!) jest przerabiana - dynamiczne priorytety są zmieniane przy pomocy prostego algorytmu:

```
recalculate:
        {
                struct task_struct *p;
                spin_unlock_irq(&runqueue_lock);
                read_lock(&tasklist_lock);
                for_each_task(p)
                        p->counter = (p->counter >> 1) + p->priority;
                read_unlock(&tasklist_lock);
                spin_lock_irq(&runqueue_lock);
        }
```

Schedule() - algorytm

- Ustawiamy next->has_cpu na 1 i next->processor na this_cpu. runqueue_lock jest zwalniany
- Włączamy proces next (jeśli prev==next to opuszczamy całe przygotowanie sprzętowe (rejestry itd.), zmienianie stron itd., i tylko przydzielamy kernel_lock obecnemu procesowi)

Scheduler 2.4 vs 2.6

- W schedulerze jądra 2.4 czas działania schedule() jest linowo zależny od liczby procesów. Skutkuje to dużym obciążeniem systemu, jeśli jest duża liczba działajacych procesów, a więc de facto procesor nie wykonuje nic produktywnego. W jądrze 2.6 mechanizm działa w czasie stałym. Scheduler jądra 2.4 wylicza kwanty czasu przydzielane procesom jednocześnie, po zakończeniu epoki, a scheduler jadra 2.6 - dla każdego procesu z osobna, w momencie kiedy zużyje on swój dotychczas przyznany czas.

Scheduler 2.4 vs 2.6

- W schedulerze 2.4 występuje globalna kolejka run_queue - wszystkie procesory muszą czekać, aż pozostałe skończą wybór.
- W schedulerze 2.4 średni przyznawany kwant czasu wynosi ok. 210 ms, natomiast w 2.6 - 100 ms. To znacząca różnica.
- Jądra 2.4 nie można wywłaszczyć - brak wsparcia dla aplikacji czasu rzeczywistego. Jądro 2.6 można wywłaszczyć, ale nie w dowolnym momencie.

Scheduler 2.4 vs 2.6

- Podbijanie priorytetów wątków wykonujących operacje wejścia - wyjścia w schedulerze 2.4 nie jest idealne:
  - nieinterakcyjne wątki mają podwyższony priorytet nawet jeśli nie jest to potrzebne (np. baza danych zapisująca lub pobierająca dużo danych)
  - Procesy wykonujące dużo obliczeń nie mają obniżanego priorytetu - pomogłoby to procesom interakcyjnym w otrzymaniu dostępu do procesora. Problem ten został rozwiązany w schedulerze 2.6
- Kod schedulera 2.4 jest około 3 razy krótszy niż kod schedulera 2.6, ale ma mniej przewidywalne zachowanie.

Scheduler O(1)

Scheduler O(1)

Problemy ze schedulerem O(1)

- Dość szybko okazało się, że istnieją przypadki, w których scheduler O(1) słabo sobie radzi
- Próbowano rozwiązywać te sytuacje, pisząc patche na scheduler
- Jednak każda łatka tworzyła nowe, wcześniej nieznane przypadki krytyczne
- Okazało się, że aby rozwiązać wszystkie te problemy, należy zmienić sposób szeregowania procesów

Przykładowy przypadek krytyczny - massive_intr.c

- Podczas uruchamiania dużej ilości procesów interaktywnych, niewielka część z nich zajmuje prawie cały czas procesora, a cała reszta ledwo działa
- Te procesy, które okupują procesor zawsze mają maksymalny możliwy effective_prio, podczas gdy reszta ma effective_prio o jeden mniejszy

Przykładowy przypadek krytyczny - massive_intr.c

Idea jest następująca:

1. Jeśli mamy umiarkowaną ilość maksymalnie interaktywnych procesów, mogą być one wielokrotnie ponownie wstawiane do kolejki active bez zmniejszania swojego priorytetu.
2. W tym przypadku pozostałe procesy rzadko się wykonują i nie mogą dostać maksymalnego effective_prio w następnej rundzie, ponieważ scheduler uważa, że zbyt długo spały
3. Idź do punktu 1

Przykładowy przypadek krytyczny - massive_intr.c

Idea jest następująca:

1. Jeśli mamy umiarkowaną ilość maksymalnie interaktywnych procesów, mogą być one wielokrotnie ponownie wstawiane do kolejki active bez zmniejszania swojego priorytetu.
2. W tym przypadku pozostałe procesy rzadko się wykonują i nie mogą dostać maksymalnego effective_prio w następnej rundzie, ponieważ scheduler uważa, że zbyt długo spały
3. Idź do punktu 1, ZONK!

Przykładowy przypadek krytyczny - massive_intr.c

Program massive_intr.c działa następująco:

1. Wczytuje parametry @nproc - ilość procesów oraz @runtime - czas działania procesów
2. Proces główny odpala @nproc procesów potomnych i czeka aż się zakończą
3. Każdy proces potomny działa w następującej pętli: “pracuj przez 8 msec i śpij przez 1 msec”
4. Po @runtime sekundach, każdy proces potomny wypisuje ilość obrotów pętli, które wykonał

Oczekiwany wynik to prawie równa ilość wykonanych obrotów pętli

Przykładowy przypadek krytyczny - massive_intr.c

- Przeprowadziłem test w następującym środowisku:
  Jądro linuksa 2.6.22-14, dwurdzeniowy procesor 32-bitowy, @nproc=200, @runtime=60
- Wyniki:
  Cztery procesy wykonały łącznie 96,2%(!) wszystkich wykonanych obrotów pętli

Na powyższym przykładzie widać, że działanie schedulera O(1) znacznie mija się z oczekiwaniami

Scheduler CFS

Scheduler CFS

Powstanie CFS

- autor: Ingo Molnar(zatrudniony w RedHat, autor schedulera O(1))
- pomysł: Kon Colivas (praktykujący lekarz z Australii)
- pierwsza implementacja: Rotating Staircase Deadline Scheduler (już z 2004 r.) i Staircase Deadline
- 100K patch w 62 godziny

“I’d like to give credit to Con Kolivas for the general approach here: he has proven via RSDL/SD that ’fair scheduling’ is possible and that it results in better desktop scheduling. Kudos Con!”

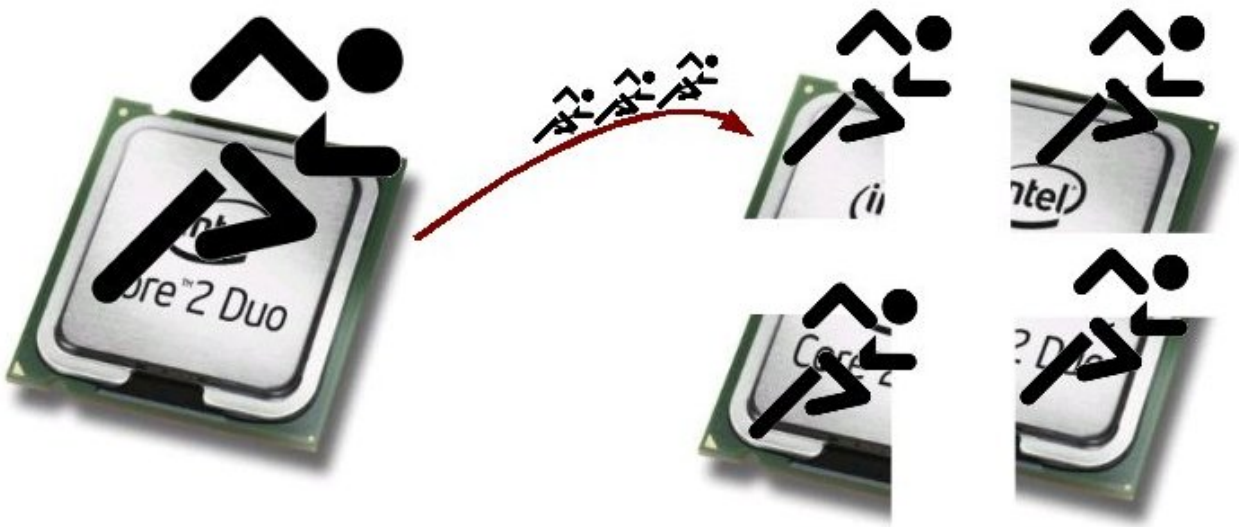
- - Ingo Molnar

Historyjka

- Linus Torvalds i Ingo Molnar skrytykowali pomysły Cona Kolivasa w 2004 r.
- Kon skarżył się, że jego wiadomości były kasowane lub odrzucane przez filtr antyspamowy
- Gdy scheduler RSDL był już gotowy, aby wejść do głównej linii jądra, Ingo Molnar pisze CFS w 3 dni
- Linus podejmuje decyzję o włączeniu CFS (z niewiadomych powodów faworyzując Ingo Molnara)
- 24 lipca 2007 Con Kolivas ogłasza decyzję o porzuceniu rozwijania jądra
- Kon zaprzecza jakoby przyczyną jego odejścia było odrzucenie RSDL przez Linusa

Idea

- Stosowane dotychczas heurystyki zbyt często się nie sprawdzają
- Dobrym pomysłem wydaje się “krojenie” procesora na mniejsze jednostki (niestety wolniejsze)
- Oczywiście jest to nierealne



p->wait_runtime

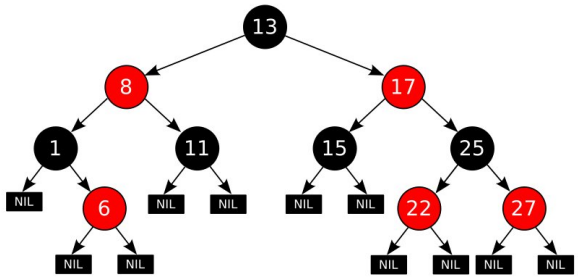
- Jeżeli mamy n procesów, to w okresie czasu t każdy powinien być przez t/n czasu na procesorze
- Dla każdego procesu staramy się liczyć czas, po którym odzyska bilans (jeżeli będzie przez tyle czasu sam “na procesorze”) - to jest właśnie p->wait_runtime
- W idealnym modelu p->wait_runtime zawsze równe 0
- Strategia: Uruchom proces z największym p->wait_runtime

rq->fair_clock

- Tak naprawdę procesy sortowane wg wartości rq->fair_clock - p->wait_runtime
- fair_clock jest zwiększany odwrotnie proporcjonalnie do obciążenia kolejki procesów (struktury runqueue): Jezeli obciążenie = 1 (zadanie), to będzie zwiększany o rzeczywiście upływający czas, jednak ze wzrastającą ilością zadań, tempo jego zwiększania będzie spadać
- Dodatkowo procesy mają różne wagi (zależne np. od parametru nice i ilości przespanego czasu)

Drzewo

- Do wyboru najmniejszej wartości służy drzewo czerwono-czarne (stąd złożoność O(logn))



Fragment nowo wprowadzonej struktury sched_entity (w takiej strukturze przechowywane są dane o procesie):

```
struct sched_entity {
        long wait_runtime;
        unsigned long delta_fair_run;
        unsigned long delta_fair_sleep;
        unsigned long delta_exec;
        s64 fair_key;
        (...)
```

Dokładność czasowa

- CFS nie opiera się ani na jiffies ani na HZ
- CFS działa z dokładnością do nanosekund
- Zbyt duża dokładność powoduje zbyt częste przełączanie procesów i spadek wydajności pamięci cache, stąd rzeczywista szczegółowość jest ustawiana na większe wartości (można ją kontrolować przez plik /proc/sys/kernel/sched_granularity_ns)
- Duża dokładność jest lepsza dla systemów typu desktop, mała natomiast lepiej sprawdza się w przypadku serwerów

Klasy schedulerów

- CFS wprowadza pojęcie klasy schedulerów, dzięki czemu łatwiejsze jest wprowadzenie nowej polityki do schedulera
- Klasy tworzą listę uporządkowaną według priorytetów
- Domyślnie lista wygląda następująco:

Czas na kod

Przejrzymy następujące funkcje:

- schedule() - wybieraj następny proces do wykonania
  - pick_next_task() - tak naprawdę to ją wywołuje schedule w celu wybrania procesu
- scheduler_tick() - wywoływana przy każdym przerwaniu pochodzącym od zegara
  - task_tick_fair() - funkcja zegara dla klasy “fair”
  - entity_tick() - funkcja uaktualniająca pojedynczy proces w klasie “fair”

Funkcja schedule()

Wybór następnego procesu w funkcji schedule() to tylko wywołania funkcji odpowiedniej klasy schedulera:

```
prev->sched_class->put_prev_task(rq, prev);
next = pick_next_task(rq, prev);
```

Jeśli nowo wybrany proces (next) jest różny od poprzednio wykonywanego (prev), to nastąpi przełączenie kontekstu:

```
if (likely(prev != next)) {
        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;

        context_switch(rq, prev, next);
```

pick_next_task()

Co robi pick_next_task w CFS?

```
/* Przeglądamy klasy schedulerów zaczynając od najbardziej
uprzywilejowanej */
class = sched_class_highest;
for ( ; ; ) {
        p = class->pick_next_task(rq);
        if (p)
                return p;

        /*
        * Z pętli wyjdziemy najpóźniej dochodząc do klasy idle
        * (ta klasa nigdy nie zwraca NULL)
        */
        class = class->next;
}
```

Funkcja scheduler_tick()

Podobnie jak schedule() polega głównie na wywołaniu funkcji z odpowiedniej klasy (w tym wypadku aktualnie wykonywanego procesu):

```
/* uaktualnianie statystyk dla cpu */
update_cpu_load(rq);
if (curr != rq->idle)
        /* wywołanie funkcji z odpowiedniej klasy */
        curr->sched_class->task_tick(rq, curr);
```

Funkcja task_tick_fair()

- task_tick_fair() - wybiera procesy do uaktualnienia:

```
struct sched_entity *se = &curr->se;
/*
* W domyślnej konfiguracji tylko bierzący proces,
* możliwe ustawienie aktualizowania całej grupy procesów
*/
for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se);
}
```

- entity_tick() - uaktualnia pojedynczy proces:

```
dequeue_entity(cfs_rq, curr, 0);
enqueue_entity(cfs_rq, curr, 0);

/* tutaj następuje pominięty fragment kodu przełączający
procesy w razie potrzeby */
```

/proc/sched_debug

Po włączeniu opcji CONFIG_SCHED_DEBUG możemy obserwować scheduler czytając plik /proc/sched_debug:

Fragment przykładowego odczytu:

```
runnable tasks:
            task   PID         tree-key      delta       waiting  switches  prio
-----------------------------------------------------------------------------------
    massive_intr  3397    157052887358    9212740     -10650375        12   120
    massive_intr  3398    157054437886   10763268     -12409926        12   120
    massive_intr  3399    157050121667    6447049      -8301523        11   120
    massive_intr  3400    157054011304   10336686     -10963103         9   120
    massive_intr  3401    157050121473    6446855      -8090781         8   120
    massive_intr  3402    157053907404   10232786     -11060141         9   120
```

Porównywanie schedulerów

Porównywanie schedulerów

Porównywanie schedulerów

No dobrze. Mamy kilka schedulerów. Nasuwają się pytania:

- Jak je porównać?
- Jak interpretować wyniki?
- Który jest lepszy?

Co mierzyć?

1. Wydajność - jak szybko procesy się wykonują
2. Koszt działania - jakie są koszty działania samego planisty
3. Skalowalność - jak scheduler radzi sobie z różną ilością procesów na różnej ilości procesorów
4. Sprawiedliwość - czy scheduler równo traktuje wszystkie procesy (uwzględniając priorytety) i nie głodzi niektórych procesów
5. Interaktywność - czy procesy interaktywne są uprzywilejowane

Powyższe cechy mogą być częściowo sprzeczne. Na przykład scheduler O(1) ma niski koszt działania, ale kiepsko u niego ze sprawiedliwością (patrz: massive_intr)

Czym mierzyć?

Microbenchmarki

- Testują konkretne cechy schedulera (wydajność, koszt działania, skalowalność)
- Nie wykluczają przypadków krytycznych, w których scheduler zachowuje się bardzo źle
- W rzeczywistości bardzo rzadko występują sytuacje takie jak w tych testach
- Są to jedyne benchmarki, których wyniki są publikowane na listach dyskusyjnych dotyczących schedulerów

Czym mierzyć?

Testy interaktywności

- Jest to zestaw różnych testów symulujących procesy interaktywne
- Testują raczej całe jądro niż sam scheduler
- Niestety, mój linux okazał się zbyt mało interaktywny i po kilkunastu minutach testu nie reagował absolutnie na nic
- Za każdym razem, gdy odpalałem test interbench, po pół godzinie musiałem pomóc schedulerowi, wykonując twardy reset

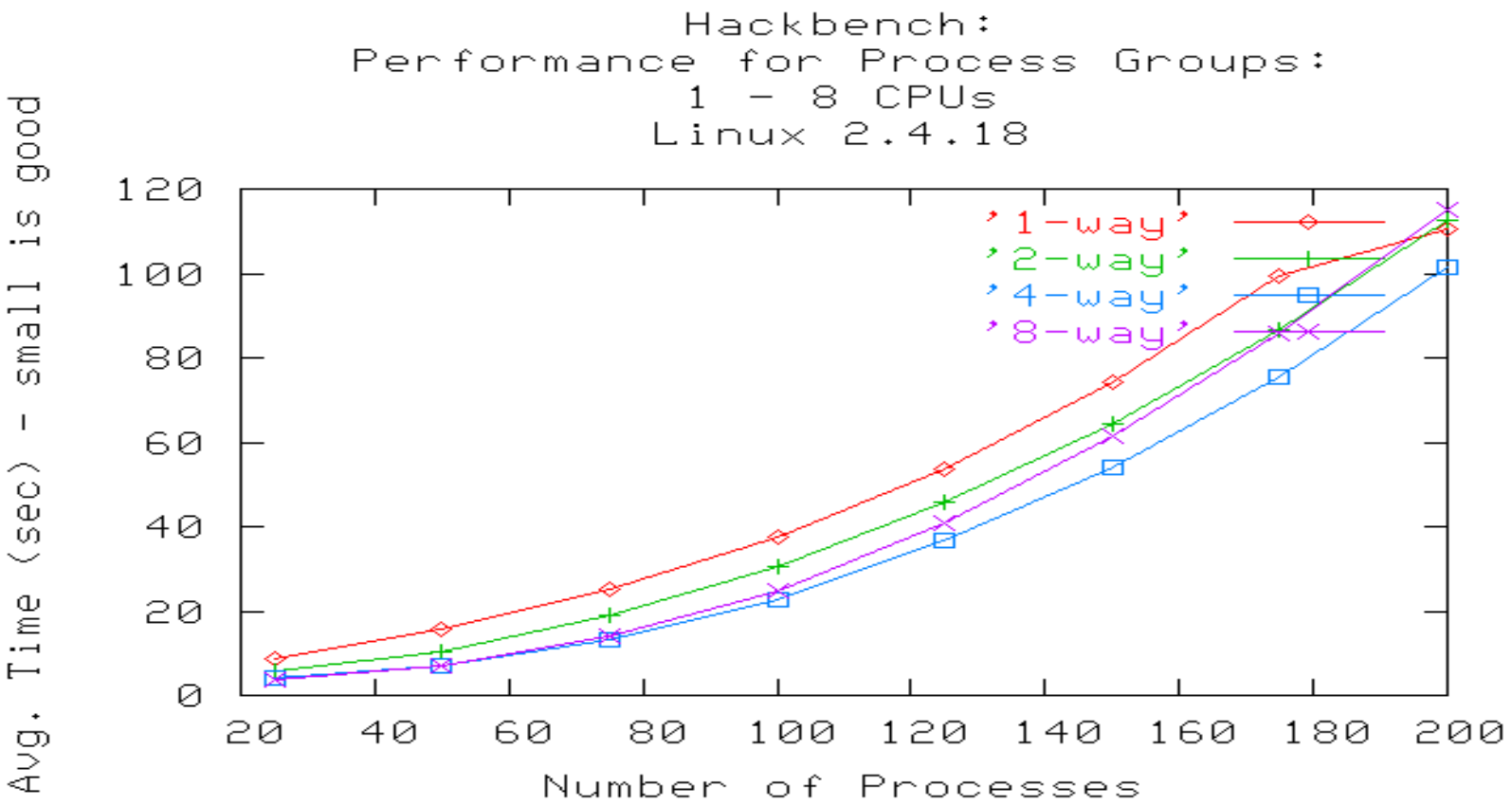
Jak mierzyć?

1. Duża ilość testów - wyniki testów zależą od chwilowego obciążenia procesora. Uruchamiając testy po kilka razy uzyskujemy wyniki bliższe rzeczywistości
2. Zróżnicowana architektura - architektura systemu może mieć wielki wpływ na zachowanie schedulera. Widać to m.in. przy porównaniu schedulera z jądra 2.4 do schedulera O(1)
3. Różna ilość procesów uruchomionych w tle - tworzymy w ten sposób sytuacje, które są bardziej zbliżone do rzeczywistych, gdyż scheduler musi sobie radzić pod różnymi obciążeniami

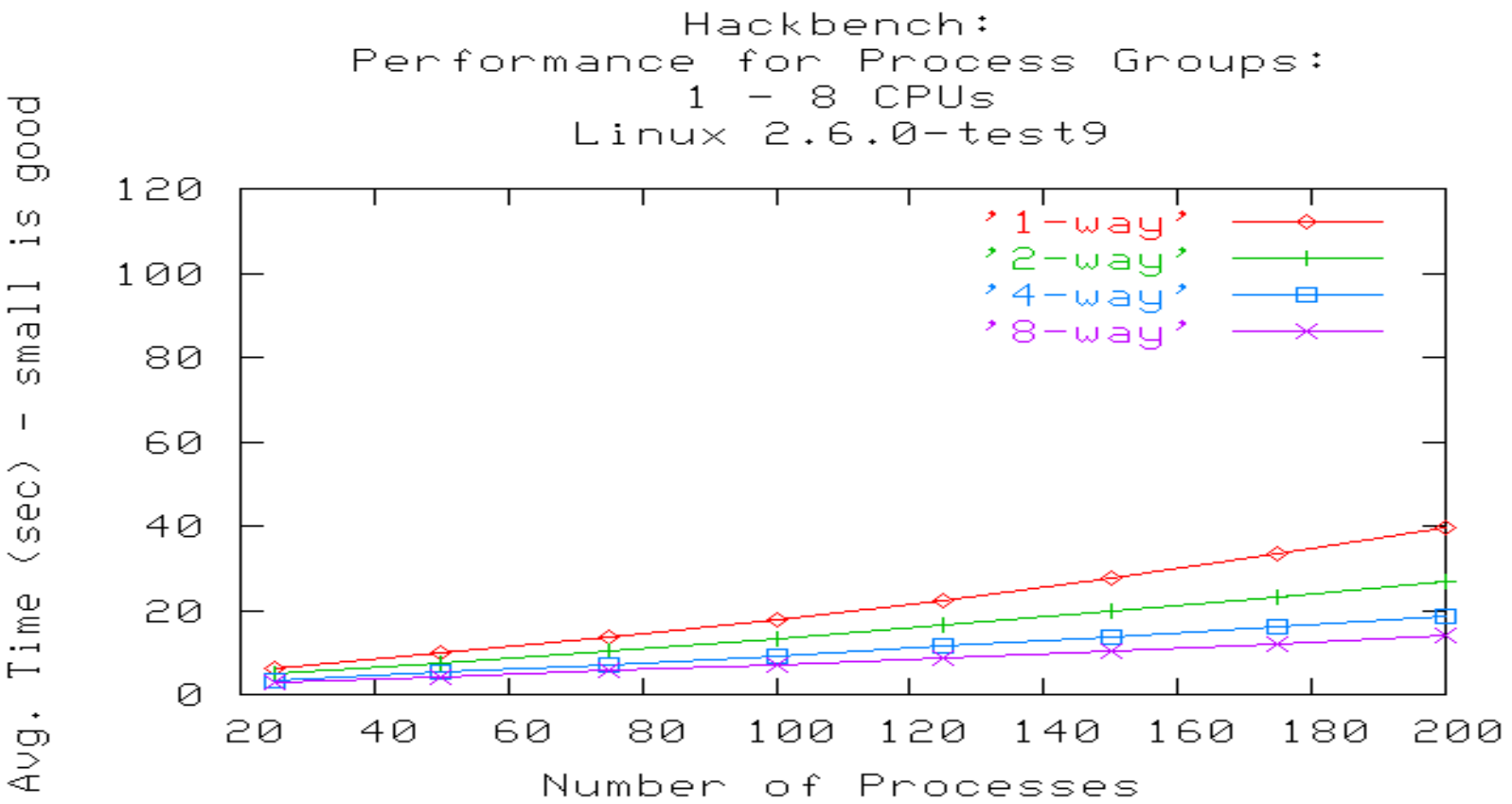
hackbench

- Mierzy skalowalność, wydajność i koszt działania schedulera
- Test jako parametr wczytuje liczbę n, a następnie uruchamia n pisarzy i n słuchaczy. Każdy pisarz nadaje 100 komunikatów do każdego ze słuchaczy, czyli np. dla n=20 każdy pisarz łącznie nadaje 2000 komunikatów, a każdy słuchacz odbiera również 2000 komunikatów
- Jako wynik otrzymujemy czas, jaki zajął test

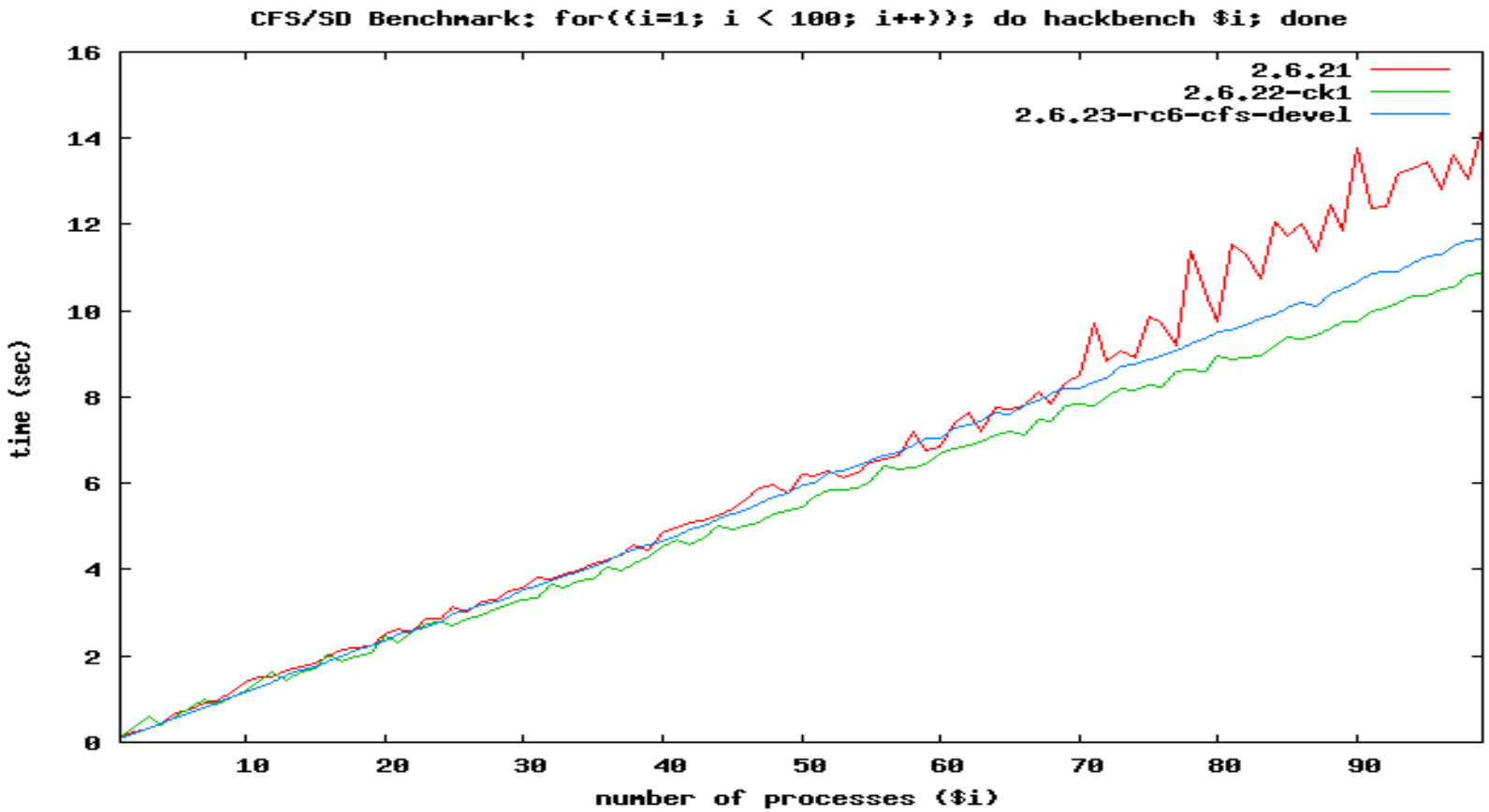
Wyniki testów - hackbench



Wyniki testów - hackbench



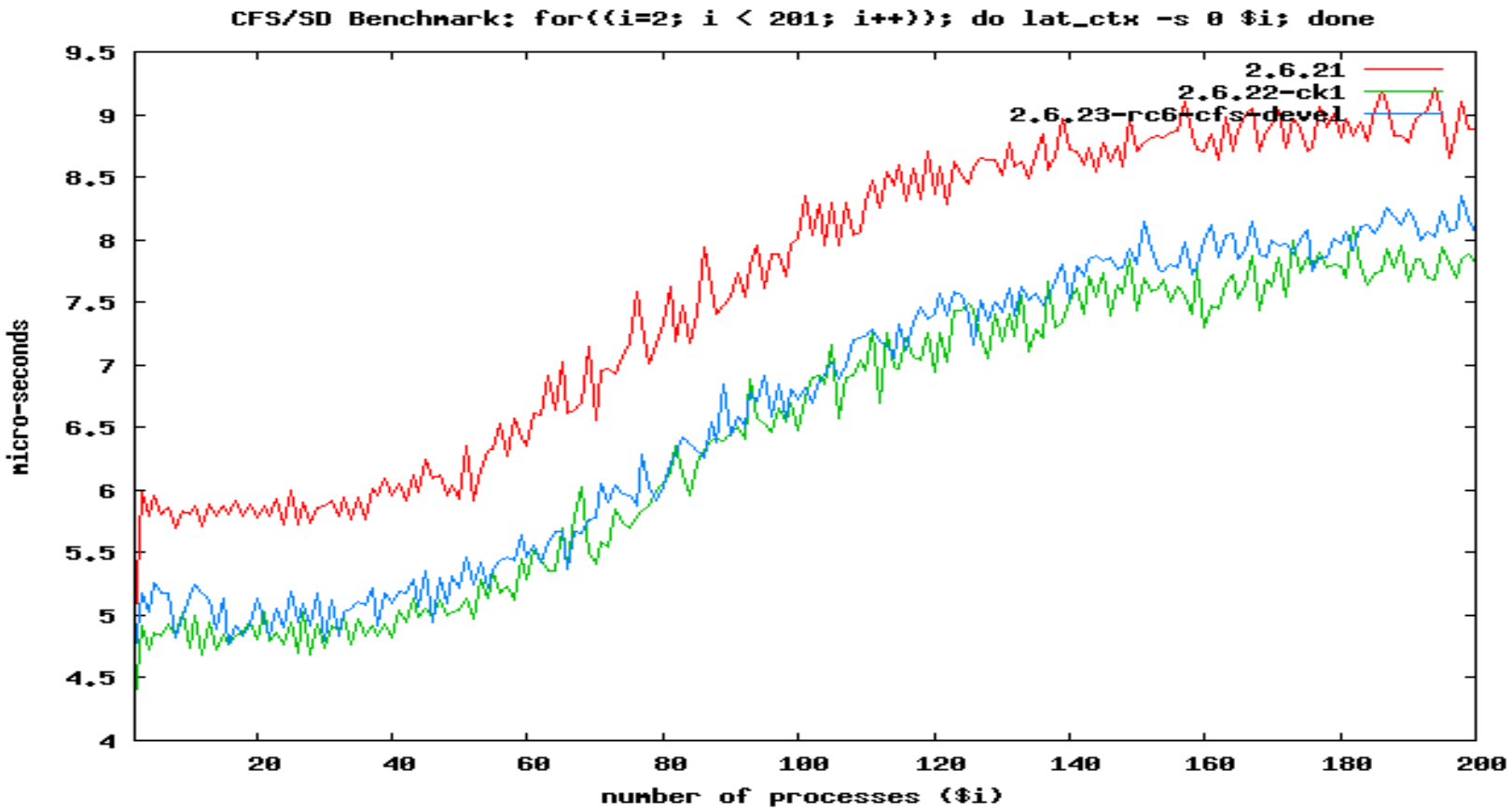
Wyniki testów - hackbench



lat_ctx

- Mierzy koszt zmiany kontekstu
- Tworzy się pierścień procesów połączonych przez pipe'y, w którym krąży żeton. Każdy proces po odbiorze żetonu wykonuje pewien kod, a następnie przekazuje żeton dalej
- W wyniku dla każdego testowanego rozmiaru pierścienia otrzymujemy łączny koszt przełączania kontekstu

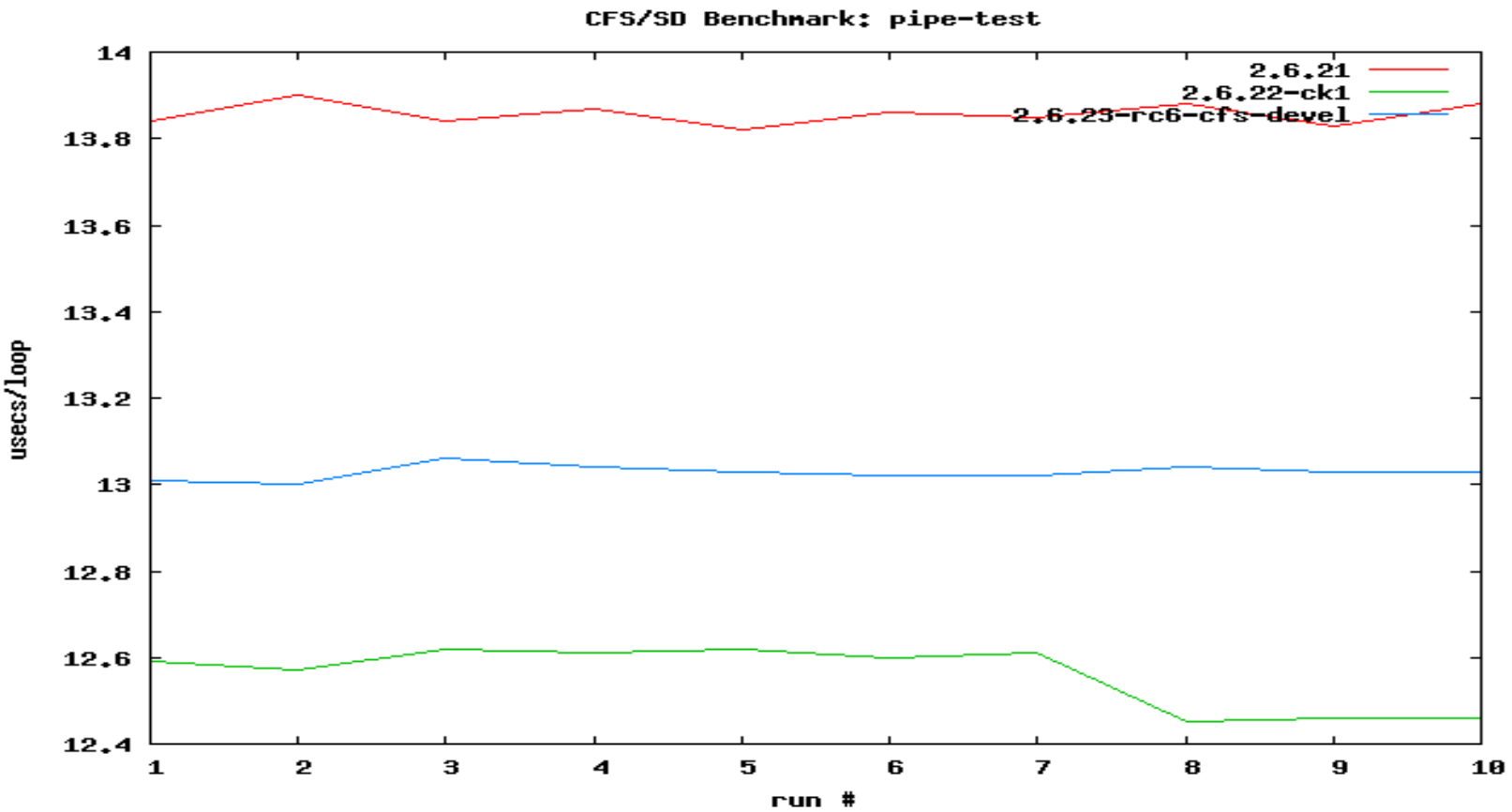
Wyniki testów - lat_ctx



pipe-test

- Mierzy koszt zmiany kontekstu
- Tworzą się dwa procesy połączone w pierścień za pomocą pipe'a. Następnie milion razy powtarza się scenariusz: proces pierwszy wkłada do pipe'a żeton z numerem rundy, drugi proces go odbiera i przekazuje z powrotem do pierwszego
- Jako wynik wypisywany jest średni czas obiegu żetonu

Wyniki testów - pipe-test



Trendy rozwojowe

Trendy rozwojowe

Rozwój CFS

- CFS jest dość nowym planistą i cały czas trwa jego optymalizacja
- W najbliższym czasie żaden inny scheduler nie wypchnie go raczej z głównej linii jądra
- Inne schedulery nie pozostają bez wpływu na jego rozwój
- Dzięki odchudzeniu, a także zawarciu w nim kilku idei wziętych z RSDL i RFS, CFS w najnowszej wersji jest średnio o 16% szybszy niż w wersji z jądra 2.6.23-rc6, a także odrobinę szybszy niż planista O(1) z jądra 2.6.22

Koncepcja Fair Scheduling

- Zaproponowana dwa lata temu przez Cona Kolivasa
- Rezygnacja z heurystyk obliczających interaktywność procesów
- Początkowo idea ta była bardzo chłodno przyjmowana
- Kolivas rozwinął własny scheduler oparty na swojej idei
- Pierwotnie nazywał się on SD (Staircase Deadline) i był wydawany jako patch na scheduler O(1)
- Później zmienił nazwę na RSDL (Rotating Staircase Deadline Scheduler) i był już odrębnym schedulerem

Koncepcja Fair Scheduling

- RSDL udowodnił poprawność koncepcji Fair Scheduling
- Miał on zostać włączony do głównej linii jądra
- Ostatecznie zamiast niego w jądrze pojawił się CFS
- CFS silnie bazuje na ideach Kolivasa
- Gdy Kolivas ogłaszał decyzję o zakończeniu prac nad rozwojem linuksa, powiedział, że jego zdaniem poświęca się za mało uwagi na wydajność linuksa na komputerach osobistych, a obciążenie linuksa rozwojem jądra pod serwery powoduje, że system ten działa słabo na komputerach osobistych

Rotating Staircase Deadline Scheduler (RSDL)

- Stosunkowo mało skomplikowany
- Zapewnia dobrą interaktywność, całkowitą sprawiedliwość i ograniczone opóźnienia
- Nawet nie próbuje określać stopnia interaktywności procesów

Rotating Staircase Deadline Scheduler (RSDL)

- Dla każdego priorytetu istnieje kolejka procesów
- Każdy proces ma przydział czasu, po wykorzystaniu którego obniżany jest mu priorytet
- Dopisuje się on wtedy do kolejki o priorytet niższej i przydzielany jest mu nowy przydział czasu
- Także każdy priorytet ma swój przydział czasu
- Kiedy najwyższy priorytet go zużyje, wszystkie procesy o tym priorytecie są przesuwane do poziomu poniżej niezależnie od tego czy wykorzystały swój indywidualny przydział czasu, czy nie. Nazywamy to “małą rotacją”.

Rotating Staircase Deadline Scheduler (RSDL)

- Gdy proces zużyje swój czas, jest przesuwany do tablicy procesów zakończonych, gdzie jest mu przypisywany jego pierwotny priorytet
- Gdy nie ma już procesów w tablicy procesów aktywnych (wszystkie wykorzystały swój przydział, bądź w wyniku małych rotacji “spadły” z tablicy procesów aktywnych), dokonuje się “duża rotacja”: zamieniane są tablice procesów aktywnych i zakończonych i cała sekwencja zdarzeń zaczyna się od nowa

Rotating Staircase Deadline Scheduler (RSDL)

Procesy często śpiące są w naturalny sposób uprzywilejowane

- Jeśli śpią, nie wykorzystają swojego przydziału dla wyższych wartości priorytetów
- Gdy proces śpi podczas “dużej rotacji”, jego przydział czasu jest ustalany na tyle, ile wynosi przydział czasu dla jego priorytetu
- Umożliwia mu to uruchamianie się z wysokim priorytetem nawet, gdy inne procesy z wysokim priorytetem, które wykowyały się, gdy ten spał, spadły na niższy priorytet w wyniku “małych rotacji”

Really Fair Scheduler (RFS)

- Roman Zippel krytykował CFS za zbyt wyszukane sztuczki i operacje, które wykonywał
- Ingo Molnar częściowo się do nich ustosunkowywał
- Zippel postanowił samodzielnie napisać uproszczoną wersję CFS, którą nazwał Really Fair Scheduler
- W odpowiedzi Molnar na podstawie CFS napisał Really Simple Really Fair Scheduler
- Zippel uznał RSRFS za zbytnio uproszczony

Plugsched - scheduler jako wtyczka

- Idea: Scheduler jako moduł wybierany podczas uruchamiania systemu
- Patch na jądro napisany przez Cona Kolivasa i inne osoby zajmujące się rozwojem schedulerów
- Możliwość łatwiejszego testowania schedulerów, a także doboru planisty do własnych potrzeb

Plugsched - scheduler jako wtyczka

- Linus Torvalds i Ingo Molnar są przeciw włączaniu go do głównej linii jądra
- Uważają oni, że byłoby dużym błędem danie użytkownikowi możliwości manipulowania mechanizmami, o których nie ma on pojęcia
- Dyskusja nad włączeniem systemu wtyczek dla planisty czasu procesora do jądra Linuksa powstała po rezygnacji z włączenia do głównej gałęzi jądra planisty RSDL na rzecz CFS.

Bibliografia

- http://www.faqs.org/docs/kernel_2_4/lki-2.html
- http://www.linuxjournal.com/article/7178
- Linux 2.4 : sched.c
- http://kerneltrap.org/CFS
- http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt
- Linux 2.6.23.9: sched.c i sched_fair.c
- http://kerneltrap.org/node/8208
- http://kerneltrap.org/mailarchive/linux-kernel/2007/9/17/261647
- http://kerneltrap.org/Linux/Benchmarking_CFS
- http://kerneltrap.org/Linux/Additional_CFS_Benchmarks
- http://kerneltrap.org/Linux/Measuring_Process_Scheduler_Performance
- http://lkml.org/lkml/2007/9/13/385
- http://spider.rz.fh-augsburg.de/informatik/vorlesungen/echtzeit/script/system/linux_2.6times.html
- http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt
- http://people.redhat.com/mingo/cfs-scheduler/tools/
- http://www.bitmover.com/lmbench/lat_ctx.8.html
- http://lwn.net/Articles/224865/
- http://kerneltrap.org/hackbench
- http://www.apcstart.com/6759/interview_with_con_kolivas_part_1_computing_is_boring
- http://kerneltrap.org/RFS
- http://www.linuxnews.pl/index.php?s=scheduler

- http://www.linuxtoday.com/developer/2007101102826OSKNDV