Scheduler CFS dla Linuksa

Juliusz Sompolski

http://students.mimuw.edu.pl/~js248396/works/sr/cfs-scheduler.pdf

Seminarium z Systemów Rozproszonych

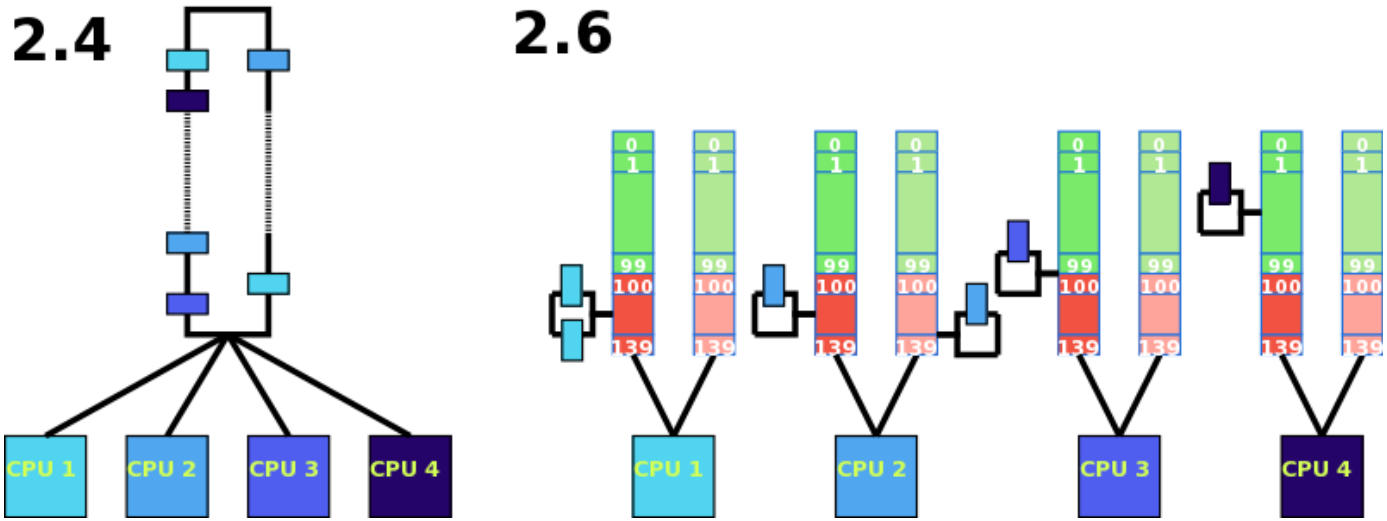
18 lutego 2010r.

Spis treści

Spis treści

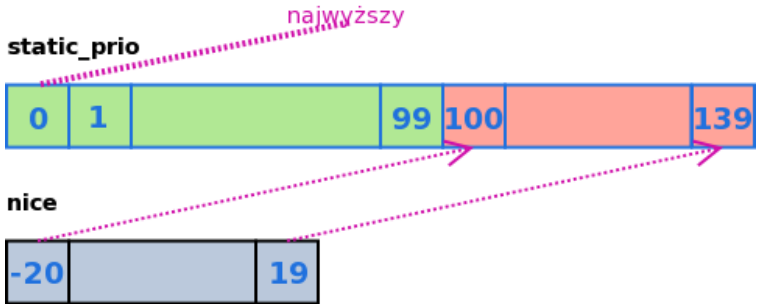
Spis treści

Scheduler O(1) w jądrze 2.6 – przypomnienie z SO



- Scheduler w jądrach serii 2.6 aż do 2.6.22
- Znaczące ulepszenia w stosunku do jądra 2.4
  - osobne kolejki per procesor i równoważenie obciążenia
  - wybór zadania w czasie O(1)
  - zamiana kolejek w czasie O(1), wykorzystując dwie tablice kolejek zadań
- **Działanie oparte na licznych heurystykach**

Priorytety



- Im niższa liczba, tym wyższy priorytet.
- W obrębie jednego priorytetu gotowe procesy czekają na kolejkach prostych.
- Scheduler wybiera gotowy proces o najwyższym priorytecie. Wybór jest w O(1), bo liczba priorytetów do sprawdzenia jest stała.
- Procesy o priorytetach 0..99 – procesy SCHED_FIFO lub SCHED_RT.
- Procesy o priorytetach 100..139 – procesy użytkownika, SCHED_NORMAL.

Spis treści

Długość kwantu w zależności od priorytetu

```
#define MIN_TIMESLICE           max(5 * HZ / 1000, 1)
#define DEF_TIMESLICE           (100 * HZ / 1000)
#define SCALE_PRIO(x, prio) \
        max(x * (MAX_PRIO - prio) / (MAX_USER_PRIO/2), MIN_TIMESLICE)

static unsigned int task_timeslice(task_t *p)
{
        if (p->static_prio < NICE_TO_PRIO(0))
                return SCALE_PRIO(DEF_TIMESLICE*4, p->static_prio);
        else
                return SCALE_PRIO(DEF_TIMESLICE, p->static_prio);
}
```

p->static_prio	100	101	105	110	115	119
task_time_slice(p) (msec)	800	780	700	600	500	420
p->static_prio	120	125	130	135	138	139
task_time_slice(p) (msec)	100	75	50	25	10	5

Problemy

Nierówny podział

- Zadanie o priorytecie 100 dostaje tylko 2.5% więcej czasu procesora niż zadanie o priorytecie 101
- Zadanie o priorytecie 138 dostaje 2 razy więcej czasu procesora niż zadanie o priorytecie 139
- Zadanie o priorytecie 119 dostaje 4 razy więcej czasu procesora niż zadanie o priorytecie 120 !

Zadania o takiej samej różnicy priorytetów mogą mieć bardzo różne proporcje przydziału procesora.

Problemy

Zadania wsadowe

Zadania wsadowe zazwyczaj wykonuje się na niższych priorytetach, aby nie powodowały one spadku responsywności interaktywnych procesów użytkowników. Jednak w momencie niskiego obciążenia, gdy zadania wsadowe mogłyby się wykonywać efektywnie, niski priorytet powoduje, że mają krótkie kwanty i często (co 5 ms przy nice +19) wymieniają się na procesorze, podczas gdy efektywniej byłoby gdyby wykonywały się w dłuższych kwantach.

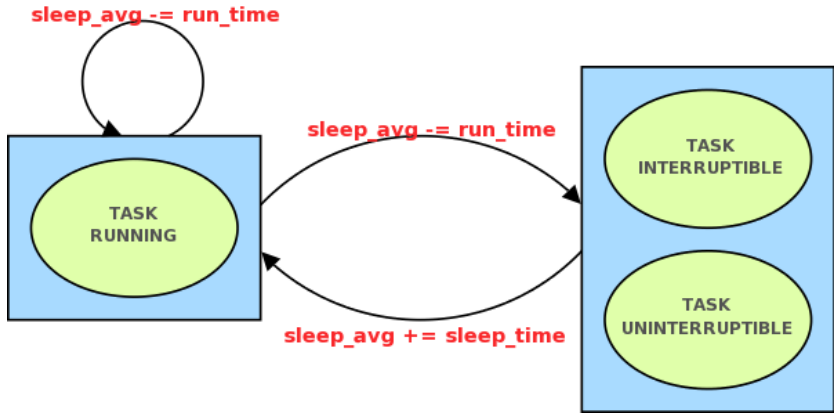
Paradoks – procesy interaktywne, które potrzebują procesora na krótsze kawałki czasu (ale częściej) otrzymują długie kwanty, a procesy które mogłyby efektywnie wykorzystywać procesor otrzymują kwanty krótkie.

Spis treści

Bonus za interaktywność

- Jeśli proces dużo śpi (oczekując na operacje i/o), to uznajemy, że jest interaktywny, należy więc zwiększyć jego priorytet, aby miał lepszą responsywność.
- Bonusy są liniowe w zakresie [-5, +5].

Bonus za interaktywność




```
#define PRIO_BONUS_RATIO          25
#define MAX_BONUS                 (MAX_USER_PRIO * PRIO_BONUS_RATIO / 100)
#define MAX_SLEEP_AVG             (DEF_TIMESLICE * MAX_BONUS) // 1000 ms
#define CURRENT_BONUS(p) \
        (NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS / MAX_SLEEP_AVG)
static int effective_prio(task_t *p)
{ (...)
        bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;
        prio = p->static_prio - bonus;
        (...)
        return prio;
}
```

Problemy

Test: thud.c, Charles Baylis, 2003

Proces (lub kilka) który na zmianę śpi, a po obudzeniu zaczyna intensywnie używać procesora. W tle działający odtwarzacz muzyki.
Efekt: W czasie snu procesy zyskują maksymalny bonus do priorytetu, po czym po obudzeniu monopolizują procesor dopóki priorytety nie zostaną im zredukowane.

Przykład z życia: X-y, które dużo śpią, a gdy zaczyniemy przełączać się między pulpitami czy oknami potrafią „przyciąć" inne procesy.
W efekcie do jądra dopisane zostały dodatkowe heurystyki, które nie pozwalają zadaniom powracającym z długiego snu uzyskać zbyt dużego bonusu, uznając takie zadania za *bezczynne*, a nie *interaktywne*.
Heurystyki te rodzą jednak nowe problemy... o czym za chwilę.

Wstawianie do kolejki aktywnej

Procesy, które uzyskały duży bonus do interaktywności mogą zostać uznane za TASK_INTERACTIVE i być wstawione z powrotem do aktywnej kolejki procesów. Im wyższy priorytet procesu, tym mniejszy bonus wystarcza, aby proces uznany został za TASK_INTERACTIVE:
TASK_INTERACTIVE(p):

bonus do prio	-5	-4	-3	-2	-1	0	1	2	3	4	5
nice -20 (100)	T	T	T	T	T	T	T	T	T	N	N
nice -10 (110)	T	T	T	T	T	T	T	N	N	N	N
nice 0 (120)	T	T	T	T	N	N	N	N	N	N	N
nice 10 (130)	T	T	N	N	N	N	N	N	N	N	N
nice 19 (139)	N	N	N	N	N	N	N	N	N	N	N

Wstawianie do kolejki aktywnej

```
#define STARVATION_LIMIT (MAX_SLEEP_AVG) // 1000 ms
#define EXPIRED_STARVING(rq) \
    ((STARVATION_LIMIT && ((rq)->expired_timestamp && \
      (jiffies - (rq)->expired_timestamp >= \
        STARVATION_LIMIT * ((rq)->nr_running) + 1))) || \
          ((rq)->curr->static_prio > (rq)->best_expired_prio))
void scheduler_tick(void)
{ (...)
  if (!rq->expired_timestamp)
      rq->expired_timestamp = jiffies;
  if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
      enqueue_task(p, rq->expired);
      if (p->static_prio < rq->best_expired_prio)
          rq->best_expired_prio = p->static_prio;
  } else
      enqueue_task(p, rq->active);
(...) }
```

Także zadania powracające ze snu dodawane są do kolejki aktywnej.

Problemy

Test: massive_intr.c, Satoru Takeuchi, 2007

Duża liczba procesów które j.w. na zmiane śpią, a po obudzeniu zaczynają intensywnie używać procesora.
Efekt: Gdy kilku procesom uda się zdobyć duży bonus, dominują procesor, gdyż uznawane są za interaktywne i po zakończeniu kwantu wkładane z powrotem do kolejki aktywnej. Dodatkowo pozostałe nie mogąc dojść do procesora uznawane są za śpiące za dużo i nie dostają bonusu do priorytetu.

W ekstremalnej konfiguracji testowej (200 procesów na jednym procesorze i386 lub 400 procesów na dwurdzeniowym ia64, nice 0, jądro 2.6.20) 4 procesy (na 2 procesorach: 8 procesów) zajmowały 98% czasu procesora.

Spis treści

Podsumowanie

- Scheduler wykonuje wszystkie operacje w czasie O(1). Pod tą stałą kryje się bardzo wiele różnych obliczeń, heurystyk i łat.
- Heurystyki w większości przypadków działają dobrze, lecz w niektórych działają bardzo źle...
- ... aby to naprawić dopisywane są kolejne heurystyki rozważające szczególne przypadki.
- W kodzie kryje się jeszcze dużo więcej haków i szczególnych przypadków.
- Heurystyki komplikują zarówno koncepcję jak i kod oraz są mało uniwersalne i traktują procesy bardzo nierówno.
- Wydawało się, że nie może być schedulera który działa lepiej oraz jest sprawiedliwy dla procesów.

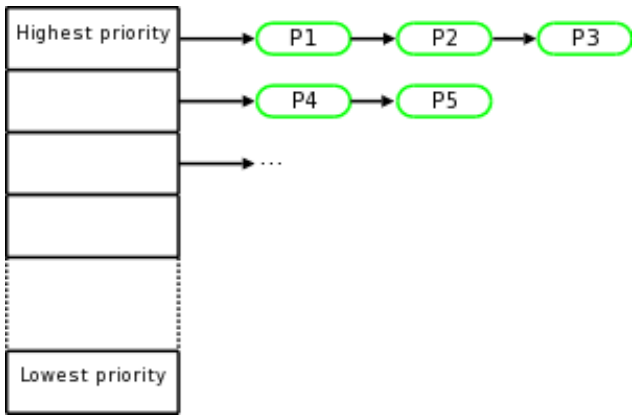
Spis treści

Spis treści

Con Kolivas

- Anestezjolog z Australii.
- Hobbysta, grzebiąc w kernelu nauczył się programować (ok. 2002 roku).
- Interesuje go głównie responsywność i dobre działanie w zastosowaniach „domowych”.
- Autor patchy sygnowanych „-ck”.
- Od 2004 roku pracował nad nowym schedulerem – Rotating Staircase Deadline Scheduler / Staircase Deadline.

Rotating Staircase Deadline Scheduler



- Bazuje na starym schedulerze, z dwoma tablicami kolejek procesów.
- Kolejki procesów tworzą „schody" priorytetów.
- Początkowo procesy dostają kwant czasu (RR_INTERVAL, domyślnie 6 ms) na swoim statycznym priorytecie.
- Kiedy skończy im się kwant dostają nowy na niższym priorytecie – „schodzą w dół schodów".
- Śpiące procesy zachowują swoje kwanty i jeśli się obudzą w tej samej epoce, mogą wykorzystać je do końca.

Rotating Staircase Deadline Scheduler

- Każdy priorytet ma też swój limit czasu wykonania. Gdy się wyczerpie, wszystkie procesy z tego „schodka" schodzą schodek niżej, niezależnie od tego, ile pozostało im kwantów – *minor rotation*.
- Procesy które wykorzystały już wszystkie przysługujące im priorytety, kolejkowane są w tablicy nieaktywnej na swoim statycznym priorytecie. Gdy tablica aktywna jest pusta, następuje *major rotation*.
- Limity czasu spędzonego na każdym priorytecie gwarantują, że epoka skończy się w ograniczonym czasie i nie będzie dochodzić do zagłodzeń.
- Procesy są traktowane sprawiedliwie, nie ma żadnych heurystyk faworyzujących procesy interaktywne, poza tym, że powracając ze snu najprawdopodobniej zostanie im więcej kwantu czasu.

W 2007 roku *Staircase Deadline* scheduler Cona Kolivasa był bliski zostania oficjalnym schedulerem w jądrze Linuxa...

Linus Torvalds, marzec 2007

I agree, partly because it's obviously been getting rave reviews so far, but mainly because it looks like you can think about behaviour a lot better, something that was always very hard with the interactivity boosters with process state history.
I'm not at all opposed to this, but we do need:
- to not do it at this stage in the stable kernel
- to let it sit in -mm for at least a short while
- and generally more people testing more loads.

So as long as the generic concerns above are under control, I'll happily try something like this if it can be merged early in a merge window.

Spis treści

Ingo Molnar

- Pracownik RedHata.
- Programista na pełny etat zajmujący się rozwojem jądra Linuksa.
- Oficjalny maintainer schedulera Linuksa.
- Autor starego schedulera z 2.6.

Completely Fair Scheduler

Ingo Molnar, kwiecień 2007

„I wrote the first line of code of the CFS patch this week, 8am Wednesday morning, and released it to lkml 62 hours later, 22pm on Friday."
„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!
The CFS patch uses a completely different approach and implementation from RSDL/SD. My goal was to make CFS's interactivity quality exceed that of RSDL/SD, which is a high standard to meet :-)"
`18 files changed, 1454 insertions(+), 1133 deletions(-)`

Nowy scheduler, który tak gwałtownie wszedł do gry, wywołał sporą burzę na linuksowych listach dyskusyjnych. Ostatecznie to CFS został wdrożony do jądra 2.6.23, a Con Kolivas zrezygnował z pracy nad jądrem Linuksa.

Spis treści

Spis treści

Reorganizacja kodu

Wraz z CFS dokonano reorganizacji kodu schedulera, tak aby wydzielić w kodzie porcje odpowiadające za politykę szeregowania.

- Stworzenie struktury `struct sched_class`, do której wskaźnik znajduje się w `struct task_struct`.
- Klasy schedulera uporządkowane są w listę, `pick_next_task` próbuje wybrać zadanie z pierwszej możliwej klasy.

```
static inline struct task_struct *pick_next_task(struct rq *rq) {
    const struct sched_class *class;
    struct task_struct *p;
    for (class = sched_class_highest; ; class = class->next) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
    }
}
```

Najważniejsze metody struct sched_class

- `enqueue_task()`, `dequeue_task()` – dodawanie procesu który staje się *runnable* i usuwanie procesu, który przestaje być *runnable*.
- `yield_task()` – zrzeczenie się procesora przez zadanie.
- `check_preempt_curr()` – sprawdza, czy nowe zadanie powinno wywłaszczyć obecnie wykonywane.
- `pick_next_task()` – wybiera nowe zadanie do wykonania.
- `put_prev_task()` – odkłada poprzednie zadanie do kolejki.
- `task_tick()` – logika wykonywana przy przerwaniu zegara.
- `task_new()` – wtyczka pozwalająca wykonać dodatkowe operacje przy tworzeniu procesu.
- `pre_schedule()`, `post_schedule()` – wtyczki pozwalające wykonać dodatkowe operacje przed/po wyborze zadania.
- Klasa nie musi implementować wszystkich procedur.

struct sched_class

```
struct sched_class {
    const struct sched_class *next;
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
    void (*yield_task) (struct rq *rq);
    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
    struct task_struct * (*pick_next_task) (struct rq *rq);
    void (*put_prev_task) (struct rq *rq, struct task_struct *p);
#ifdef CONFIG_SMP
    int  (*select_task_rq)(struct task_struct *p, int sd_flag, int flags);
    unsigned long (*load_balance) (struct rq *this_rq, int this_cpu,
            struct rq *busiest, unsigned long max_load_move,
            struct sched_domain *sd, enum cpu_idle_type idle,
            int *all_pinned, int *this_best_prio);
    int (*move_one_task) (struct rq *this_rq, int this_cpu,
                  struct rq *busiest, struct sched_domain *sd,
                  enum cpu_idle_type idle);
    void (*pre_schedule) (struct rq *this_rq, struct task_struct *task);
    void (*post_schedule) (struct rq *this_rq);
    void (*task_wake_up) (struct rq *this_rq, struct task_struct *task);
    void (*set_cpus_allowed)(struct task_struct *p,
                 const struct cpumask *newmask);
    void (*rq_online)(struct rq *rq);
    void (*rq_offline)(struct rq *rq);
#endif
    void (*set_curr_task) (struct rq *rq);
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    void (*task_new) (struct rq *rq, struct task_struct *p);
    void (*switched_from) (struct rq *this_rq, struct task_struct *task,
                   int running);
    void (*switched_to) (struct rq *this_rq, struct task_struct *task,
                 int running);
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                 int oldprio, int running);
    unsigned int (*get_rr_interval) (struct task_struct *task);
#ifdef CONFIG_FAIR_GROUP_SCHED
    void (*moved_group) (struct task_struct *p);
#endif
};
```

Klasy schedulera

W nowym schedulerze umieszczono następujące klasy schedulowania (od najwyższej do najniższej):

- `rt_sched_class` (`sched_rt.c`)
  - Obsługuje procesy `SCHED_RR` i `SCHED_FIFO`.
  - Zasada działania dokładnie taka jak w starym schedulerze. Kilka uproszczeń – np. wystarcza jedna tablica kolejek, gdyż zadania RT i tak były odkładane z powrotem do tablicy aktywnej.
- `fair_sched_class` (`sched_fair.c`)
  - Obsługuje procesy `SCHED_NORMAL`, `SCHED_BATCH` i `SCHED_IDLE`
  - Szeroko o niej za chwilę.
- `idle_sched_class` (`sched_idletask.c`)
  - Proces bezczynności.

Modular scheduler design vs plugsched

Con Kolivas zarzucał CFS, że wykorzystał on jego pomysły modularyzacji schedulera zawarte we frameworku `plugsched` (umożliwiającego łatwą wymianę schedulerów), mimo, że wcześniej były krytykowane:

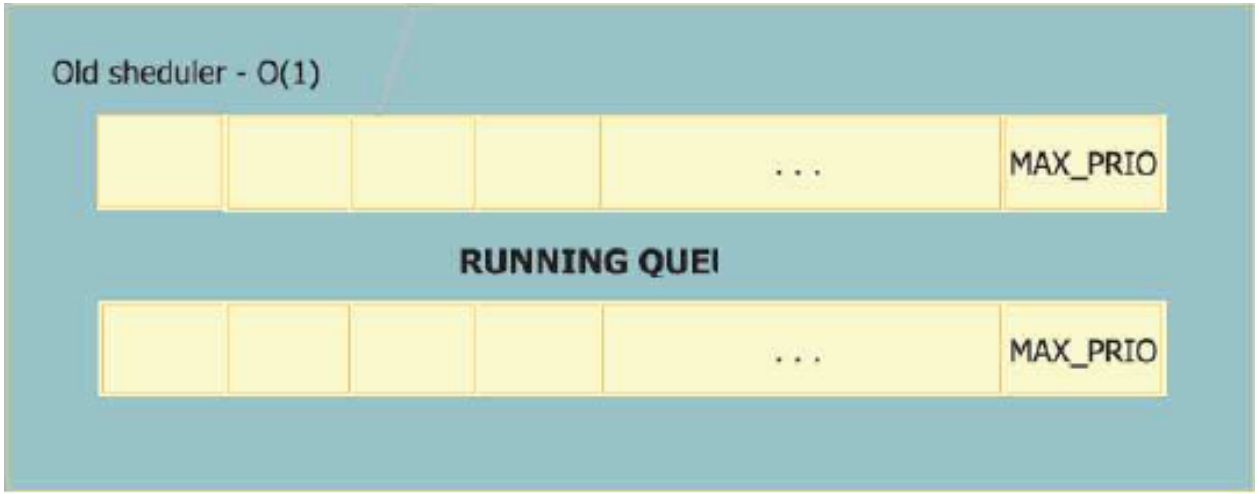
Linus Torvalds o wymiennych schedulerach

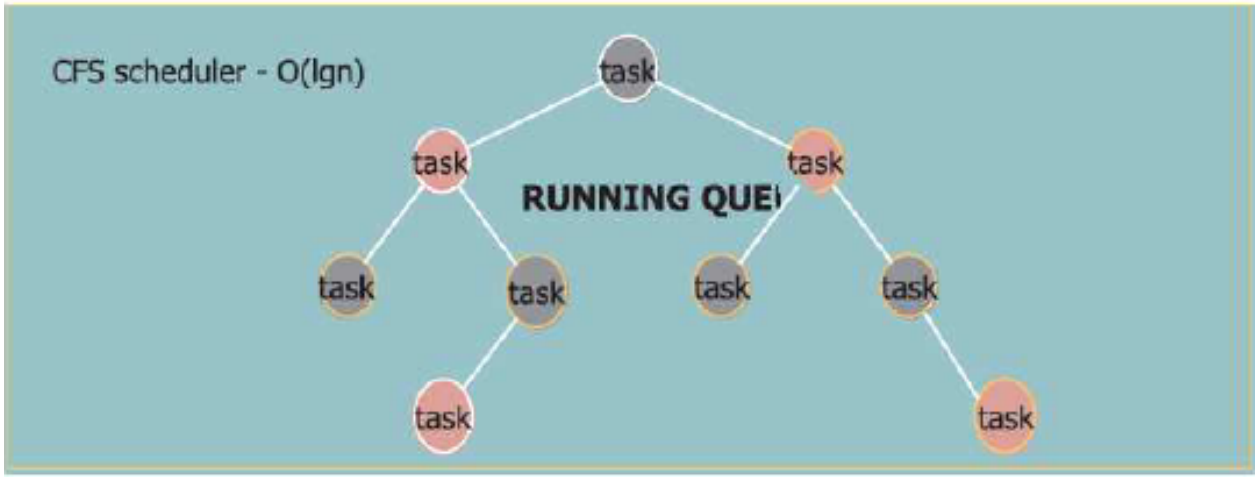
The arguments that 'servers' have a different profile than 'desktop' is pure and utter garbage, and is perpetuated by people who don't know what they are talking about. (...) Yes, there are differences in tuning, but those have nothing to do with the basic algorithm. They have to do with goals and trade-offs, and most of the time we should aim for those things to auto-tune.

Ingo odpierał zarzuty Cona twierdząc, że zaproponowana przez niego reorganizacja kodu schedulera nie miała na celu umożliwienia wymienności schedulerów, a jedynie poprawienie jakości kodu.

Spis treści

Completely Fair Scheduler





Główne założenia

- Całkowicie nowa koncepcja schedulera.
- Koniec z kolejkami procesów. Zamiast nich – jedno drzewo czerwono-czarne procesów, posortowanych po „wirtualnym czasie" wykonania i wybór zadania, które wykonywało się najkrócej.
- Wirtualny czas płynący z prędkością zależną od priorytetu.
- Możliwe także grupowanie zadań i sprawiedliwy podział czasu pomiędzy zdefiniowane „encje", grupy procesów.
- Decyzja o wyborze procesu w $O(1)$. Wstawianie procesu w $O(\log n)$ zamiast w $O(1)$.

Budowa CFS

CFS przechowuje potrzebne mu do szeregowania procesów dane w dodatkowych strukturach dodanych do `struct rq` i `struct task_struct`.

```
struct rq {
+   struct cfs_rq cfs;
}

struct task_struct {
+   const struct sched_class *sched_class;
+   struct sched_entity se;
}
```

struct sched_entity

- Do drzewa wstawiane są „encje” szeregowania, posortowane po wirtualnym czasie `vruntime` – jest to czas wykonywania, przeskalowany o czynnik zależny od priorytetu.
- „Encje” mogą odpowiadać procesom (pole se w `struct task_struct`, ale mogą też odpowiadać grupom procesów.

```
struct sched_entity {
    struct load_weight load;
    struct rb_node run_node;
    u64 vruntime;
#ifdef CONFIG_FAIR_GROUP_SCHED
    struct sched_entity *parent;
    /* rq on which this entity is (to be) queued: */
    struct cfs_rq *cfs_rq;
    /* rq "owned" by this entity/group: */
    struct cfs_rq *my_q;
#endif
    (...)
};
```

struct sched_entity – grupy procesów

- Grupy procesów zorganizowane są w sposób hierarchiczny.
- `struct sched_entity *parent` – wskaźnik do encji nadrzędnej do której należy dana encja.
- `struct cfs_rq *my_q` – kolejka encji należących do danej encji. NULL gdy encja jest procesem.
- `struct cfs_rq *cfs_rq` – kolejka do której dana encja będzie wstawiana (czyli `parent->my_q` jeśli istnieje encja nadrzędna lub `rq->cfs_rq` dla encji najwyższego poziomu).
- Do `vruntime` dodawany jest czas wykonania jej podencji.
- `load` dla procesu jest jego wagą skalującą czas `vruntime`

Wagi procesów

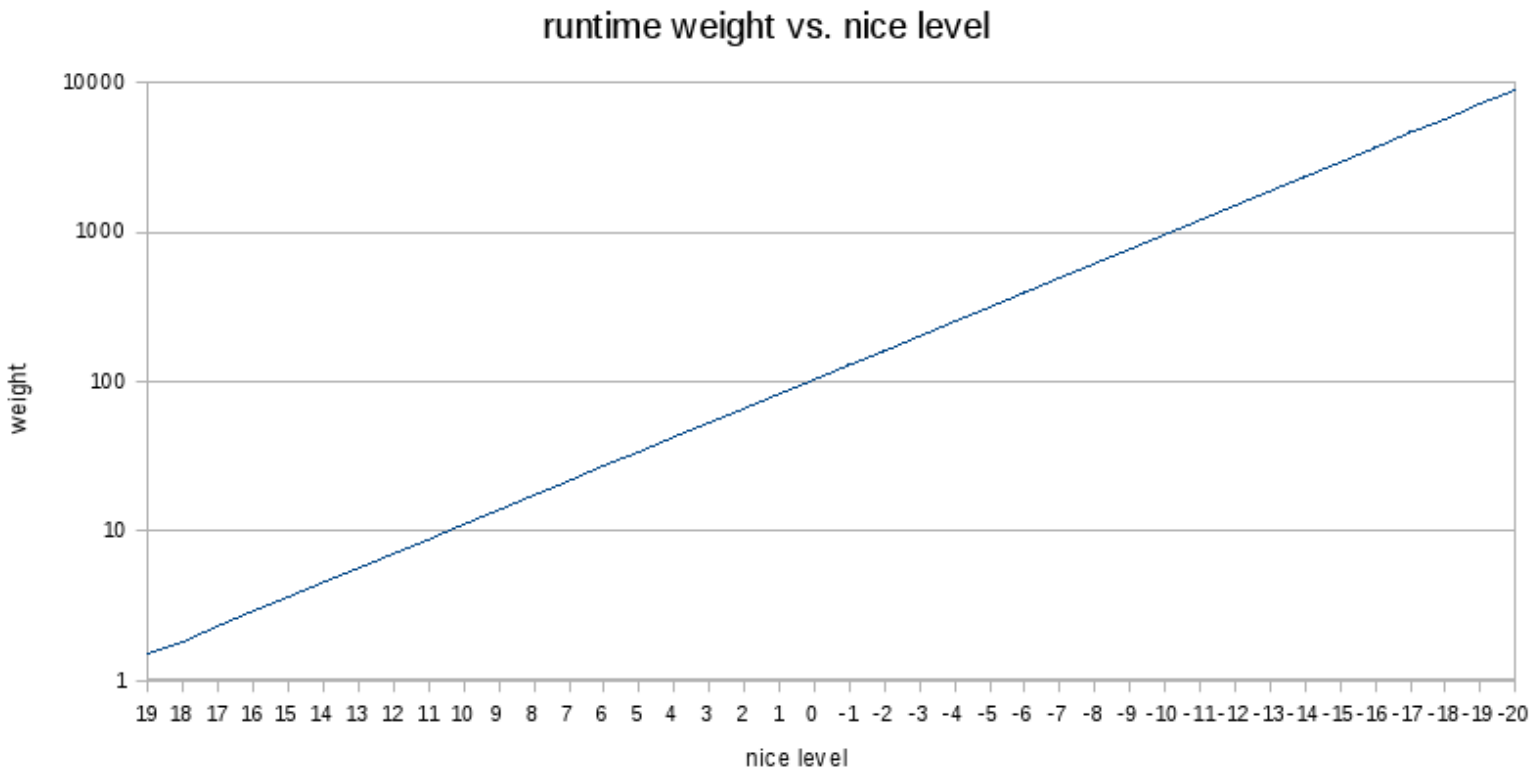
- Procesy o różnych priorytetach otrzymują różne wagi. Wirtualny czas, według którego są szeregowane jest skalowany tymi wagami – w ten sposób scheduler uwzględnia różnice w *nice* procesów.
- Wagi priorytetów przydzielone są jako postęp geometryczny:

  `prio_to_weight[n]` $\approx$ `prio_to_weight[n+1] * 1.25`

  Dzięki temu procesy o danej różnicy priorytetów zawsze otrzymywać będą czas procesora w stałej proporcji.
- (`inv_weight` = 2^{32} / `weight` przechowywane aby przyśpieszyć obliczenia przez zamianę dzielenia na mnożenie).

```
struct load_weight {
    unsigned long weight, inv_weight;
};
```

Wagi procesów



Proces o *nice* -20 dostanie ok. 6000 razy więcej czasu procesora niż proces o *nice* 19 (dla porównania: 160 razy w starym schedulerze).

struct cfs_rq

- Struktura przechowywująca drzewo encji.
- `min_vruntime` przechowuje najmniejszy wirtualny czas encji w drzewie; istotny przy wstawianiu nowego procesu.
- `task_weight` jest to suma wag procesów w drzewie, używana przy równoważeniu obciążenia między procesorami.

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
    u64 min_vruntime;
    struct rb_root tasks_timeline;
    struct rb_node *rb_leftmost;
#ifdef CONFIG_SMP
    /* the part of load.weight contributed by tasks */
    unsigned long task_weight;
#endif
    (...)
};
```

Implementacja fair_sched_class

```
static const struct sched_class fair_sched_class = {
    .next                   = &idle_sched_class,
    .enqueue_task           = enqueue_task_fair,
    .dequeue_task           = dequeue_task_fair,
    .yield_task             = yield_task_fair,
    .check_preempt_curr     = check_preempt_wakeup,
    .pick_next_task         = pick_next_task_fair,
    .put_prev_task          = put_prev_task_fair,
#ifdef CONFIG_SMP
    .select_task_rq         = select_task_rq_fair,
    .load_balance           = load_balance_fair,
    .move_one_task          = move_one_task_fair,
#endif
    .set_curr_task          = set_curr_task_fair,
    .task_tick              = task_tick_fair,
    .task_new               = task_new_fair,
    .prio_changed           = prio_changed_fair,
    .switched_to            = switched_to_fair,
    .get_rr_interval        = get_rr_interval_fair,
#ifdef CONFIG_FAIR_GROUP_SCHED
    .moved_group            = moved_group_fair,
#endif
};
```

Kwanty czasu

Epoka i kwanty

- `sched_latency_ns` – czas, w którym powinna wykonać się jedna epoka, tzn. powinny zostać wykonane wszystkie zadania z kolejki. Domyślnie 20 ms.
- `sched_min_granularity_ns` – minimalny czas, który średnio powinien przypadać na zadanie w epoce. Domyślnie 4 ms.

$$\texttt{epoch} = \max(\texttt{sched\_latency\_ns}, \texttt{sched\_min\_granularity\_ns} \cdot \texttt{nr\_running})$$

$$\texttt{ideal\_slice[se]} = \texttt{epoch} \cdot \frac{weight}{\sum weight}$$

Zmienne długości kwantów, ale stały okres obrotu epoki, gwarantuje małe opóźnienia. Pod dużym obciążeniem kwanty nie są skracane poniżej minimalnej wartości, kosztem responsywności.

Kwanty czasu

```
static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    u64 slice = __sched_period(cfs_rq->nr_running + !se->on_rq);
    /* slice = epoch */
    for_each_sched_entity(se) { /* for (; se; se = se->parent) */
        struct load_weight *load;
        struct load_weight lw;
        cfs_rq = cfs_rq_of(se);
        load = &cfs_rq->load;
        (...)
        slice = calc_delta_mine(slice, se->load.weight, load);
        /* slice = slice * se->load.weight / load; */
    }
    return slice;
}
```

Obliczanie kwantu czasu porusza się w górę hierarchii encji szeregowania, obliczając ułamek czasu należnego encji w nadgrupie, nadgrupy w jej nadgrupie etc.

Wybór zadania do wykonania

```
static struct task_struct *pick_next_task_fair(struct rq *rq)
{
    struct task_struct *p;
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;
    (...)
    do {
        se = pick_next_entity(cfs_rq);
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);
    p = task_of(se);
    hrtick_start_fair(rq, p);
    return p;
}
```

- `pick_next_entity` wybiera encje o najmniejszym `vruntime` z drzewa (z grubsza). Encje wybierane są w dół hierarchii grupowania, aż dotrzemy do procesu.
- `hrtick_start_fair` nastawia timer, który wywłaszczy zadanie po obliczonym dla niego `ideal_slice`.

Wstawianie zadania

```
static void enqueue_task_fair(struct rq *rq, struct task_struct *p, int wakeup)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se;
    for_each_sched_entity(se) {
        if (se->on_rq)
            break;
        cfs_rq = cfs_rq_of(se);
        enqueue_entity(cfs_rq, se, wakeup);
        wakeup = 1;
    }
    hrtick_update(rq);
}
```

- Wstawia zadanie i w razie potrzeby jego encje nadrzędne.
- `hrtick_update` poprawia timer.

Gdzie zostanie wstawione zadanie?

- Nowy proces wstawiony zostanie na koniec aktualnej epoki.
- Jeśli budzący się proces spał krócej niż jedną epokę, zachowa swój czas `vruntime` i będzie miał bonus w postaci możliwości „nadrobienia" przespanego czasu.
- Proces budzący się z dłuższego snu także otrzymuje możliwość „nadrobienia" części przespanego czasu.

```
vruntime = cfs_rq->min_vruntime - sysctl_sched_latency
```

```
static void task_new_fair(struct rq *rq, struct task_struct *p)
{ (...)
    if (curr) se->vruntime = curr->vruntime;
    place_entity(cfs_rq, se, 1);
(...) }

static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int wakeup)
{ (...)
    if (wakeup) { (...) place_entity(cfs_rq, se, 0); (...) }
(...) }
```

Gdzie zostanie wstawione zadanie?

```
static void
place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
{
    u64 vruntime = cfs_rq->min_vruntime;
    /* nowe zadanie na koniec epoki */
    if (initial && sched_feat(START_DEBIT))
        vruntime += sched_vslice(cfs_rq, se);
    /* jesli powraca ze snu, to dostaje bonus */
    if (!initial && sched_feat(FAIR_SLEEPERS)) {
        unsigned long thresh = sysctl_sched_latency;
        (...)
        vruntime -= thresh;
    }
    vruntime = max_vruntime(se->vruntime, vruntime);
    se->vruntime = vruntime;
}
```

Usuwanie zadania

```
static void dequeue_task_fair(struct rq *rq, struct task_struct *p, int sleep)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se;

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        dequeue_entity(cfs_rq, se, sleep);
        /* Don't dequeue parent if it has other entities besides us */
        if (cfs_rq->load.weight)
            break;
        sleep = 1;
    }

    hrtick_update(rq);
}
```

Usunięcie zadania oraz jego nadencji (jeśli nie mają innych zadań) z drzewa.

Tyknięcie zegara

```
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &curr->se;

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
    }
}
```

W czasie tyknięcia zegara przechodzimy po hierarchii encji, sprawdzając czy którejś z nich nie skończył się kwant czasu.

Tyknięcie zegara

```
static void
entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr, int queued)
{
    update_curr(cfs_rq);
#ifdef CONFIG_SCHED_HRTICK
    /* queued ticks are scheduled to match the slice, so don't bother
     * validating it and just reschedule. */
    if (queued) {
        resched_task(rq_of(cfs_rq)->curr);
        return;
    }
    (...)
#endif
    if (cfs_rq->nr_running > 1 || !sched_feat(WAKEUP_PREEMPT))
        check_preempt_tick(cfs_rq, curr);
}
```

Jeżeli używamy timerów wysokiej rozdzielczości, to nie trzeba sprawdzać czy kwant się skończył, gdyż są one nastawiane na właściwy czas przez `hrtick_start_fair` i `hrtick_update`.

Tyknięcie zegara

```
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    unsigned long ideal_runtime, delta_exec;

    ideal_runtime = sched_slice(cfs_rq, curr);
    delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    if (delta_exec > ideal_runtime) {
        resched_task(rq_of(cfs_rq)->curr);
        (...)
        return;
    }
    (...)
}
```

W przeciwnym razie zadanie jest wywłaszczane jeżeli pracowało dłużej niż jego `ideal_runtime`.

Wywłaszczanie przez obudzone zadania

```
static void check_preempt_wakeup(struct rq *rq, struct task_struct *p, int wake_flags)
{
    struct task_struct *curr = rq->curr;
    struct sched_entity *se = &curr->se, *pse = &p->se;
    if (!sched_feat(WAKEUP_PREEMPT))
        return;
    find_matching_se(&se, &pse);
    (...)
    if (wakeup_preempt_entity(se, pse) == 1) {
        resched_task(curr);
        (...)
    }
}
static int
wakeup_preempt_entity(struct sched_entity *curr, struct sched_entity *se)
{
    s64 gran, vdiff = curr->vruntime - se->vruntime;
    gran = wakeup_gran(curr, se); // sysctl_sched_wakeup_granularity;
    if (vdiff > gran)
        return 1;
    return 0;
}
```

Nowo wstawione zadanie musi mieć `vruntime` co najmniej o `sysctl_sched_wakeup_granularity` mniejszy, aby wywłaszczyć aktualnie wykonywane zadanie. Zapobiega to wywłaszczaniu zadań tuż po rozpoczęciu przez nie kwantu czasu.

Możliwości dostrajania

Poprzez `/proc/sys/kernel/sched_*`:

- `sched_latency_ns` – domyślny czas, w ciągu którego wszystkie zadania powinny otrzymać procesor (20 ms).
- `sched_min_granularity_ns` – minimalna długość kwantu czasu średnio przypadająca na zadanie (4 ms).
- `sched_wakeup_granularity_ns` – o ile nowo obudzone zadanie musi mieć lepszy czas, aby wywłaszczyć aktualne (5 ms).
- `sched_compat_yield` – nie pozwala procesom na dobrowolne zrzekanie się procesora przez `yield()` (wyłączone).
- `sched_child_runs_first` – sprawia, że po `fork()` procesor najpierw przyznawany jest procesowi potomnemu (włączone).
- `sched_features` – inne opcje dostrajania.

Możliwości dostrajania

Do `/proc/sys/kernel/sched_features` można pisać:

```
echo FEATURE_NAME    > sched_features
echo NO_FEATURE_NAME > sched_features
```

- `NEW_FAIR_SLEEPERS` – pozwala obudzonym zadaniom nadrobić część przespanego czasu.
- `NORMALIZED_SLEEPER` – dodatkowo normalizuje bonus zależnie od priorytetu.
- `WAKEUP_PREEMPT` – gdy jest wyłączone, budzone procesy nie wywłaszczają obecnie wykonywanych zadań, które mogą dokończyć swój kwant.
- `START_DEBIT` – proces potomny rozpoczyna z `vruntime` zwiększonym o kwant (gdy `sched_child_runs_first` – proces rodzicielski)
- `ASYM_GRAN` – dodatkowo normalizuje `sched_wakeup_granularity` zależnie od priorytetu.
- `HRTICK` – włącza timery wysokiej rozdzielczości

Grupowanie zadań – CONFIG_USER_SCHED

Gdy w konfiguracji kernela włączone jest `CONFIG_USER_SCHED`, czas procesora dzielony jest sprawiedliwie pomiędzy użytkowników.

```
# cd /sys/kernel/uids
# cat 512/cpu_share # Display user 512's CPU share
1024
# echo 2048 > 512/cpu_share # Modify user 512's CPU share
# cat 512/cpu_share # Display user 512's CPU share
2048
```

Grupowanie zadań – CONFIG_CGROUP_SCHED

Gdy w konfiguracji kernela włączone jest `CONFIG_CGROUP_SCHED`, czas procesora dzielony jest sprawiedliwie pomiędzy grupy cgroups.

```
# mkdir /dev/cpuctl
# mount -t cgroup -ocpu none /dev/cpuctl
# cd /dev/cpuctl

# mkdir multimedia # create "multimedia" group of tasks
# mkdir browser # create "browser" group of tasks

# echo 2048 > multimedia/cpu.shares
# echo 1024 > browser/cpu.shares

# firefox & # Launch firefox and move it to "browser" group
# echo <firefox_pid> > browser/tasks
```

Grupowanie zadań – przyszłość

- Kernel obsługuje już w pełni hierarchiczne grupy procesów – `struct sched_entity` uporządkowane są w drzewo.
- Obecnie interfejs konfiguracji grupowania pozwala jednak tylko na tworzenie hierarchii płaskich, nie ma możliwości skonfigurowania bardziej rozbudowanego grupowania
  - albo grupy dla użytkowników
  - albo grupy jako cgroups

Obecnie trwają prace nad tym jak właściwie taka bardziej zaawansowana, umożliwiająca zagnieżdżanie konfiguracja miałaby wyglądać.

Podgląd działania schedulera

- Działanie schedulera można podejrzeć czytając z pliku `/proc/sched_debug`. Wyświetla on wartości parametrów schedulera oraz aktualną zawartość drzewa gotowych procesów.
- Statystyki związane z szeregowaniem konkretnego procesu można podejrzeć poprzez `/proc/<pid>/sched`.

Podgląd działania schedulera

```
jullass@hermes:/proc/$ cat sched_debug
Sched Debug Version: v0.07, 2.6.26-2-686 #1
now at 107837415.170126 msecs
  .sysctl_sched_latency                    : 20.000000
  .sysctl_sched_min_granularity            : 4.000000
  .sysctl_sched_wakeup_granularity         : 10.000000
  .sysctl_sched_child_runs_first           : 0.000001
  .sysctl_sched_features                   : 895

cpu#0, 1594.872 MHz
  .nr_running                    : 5
  .load                          : 5120
  .nr_switches                   : 128388762
  .nr_load_updates               : 19703795
  .nr_uninterruptible            : 0
  .jiffies                       : 26884353
  .next_balance                  : 26.884376
  .curr->pid                     : 5034
  .clock                         : 107837414.402970
  .cpu_load[0]                   : 2048
  .cpu_load[1]                   : 2048
  .cpu_load[2]                   : 2047
  .cpu_load[3]                   : 1963
  .cpu_load[4]                   : 1737

cfs_rq[0]:
  .exec_clock                    : 0.000000
  .MIN_vruntime                  : 19449343.869908
  .min_vruntime                  : 19449367.494149
  .max_vruntime                  : 19449367.494149
  .spread                        : 23.624241
  .spread0                       : 0.000000
  .nr_running                    : 5
  .load                          : 5120
  .nr_spread_over                : 0

runnable tasks:
            task   PID         tree-key  switches  prio     exec-runtime         sum-exec        sum-sleep
----------------------------------------------------------------------------------------------------------
         yakuake  3432  19449348.325627    363943   120               0               0             0.000000             0.000000
           opera  3593  19449367.494149  38588558   120               0               0             0.000000             0.000000
            bash  4157  19449343.869908       119   120               0               0             0.000000             0.000000
        pdflatex  5030  19449359.522826       471   120               0               0             0.000000             0.000000
R            cat  5034  19449328.325627         1   120               0               0             0.000000             0.000000
```

Podgląd działania schedulera

```
jullass@hermes:/proc/29666$ cat sched
htop (29666, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :      109169258.813659
se.vruntime                                  :       19738536.594745
se.sum_exec_runtime                          :         199720.128567
se.avg_overlap                               :              0.032160
nr_switches                                  :                 37122
nr_voluntary_switches                        :                 10818
nr_involuntary_switches                      :                 26304
se.load.weight                               :                  1024
policy                                       :                     0
prio                                         :                   120
clock-delta                                  :                   212
```

Spis treści

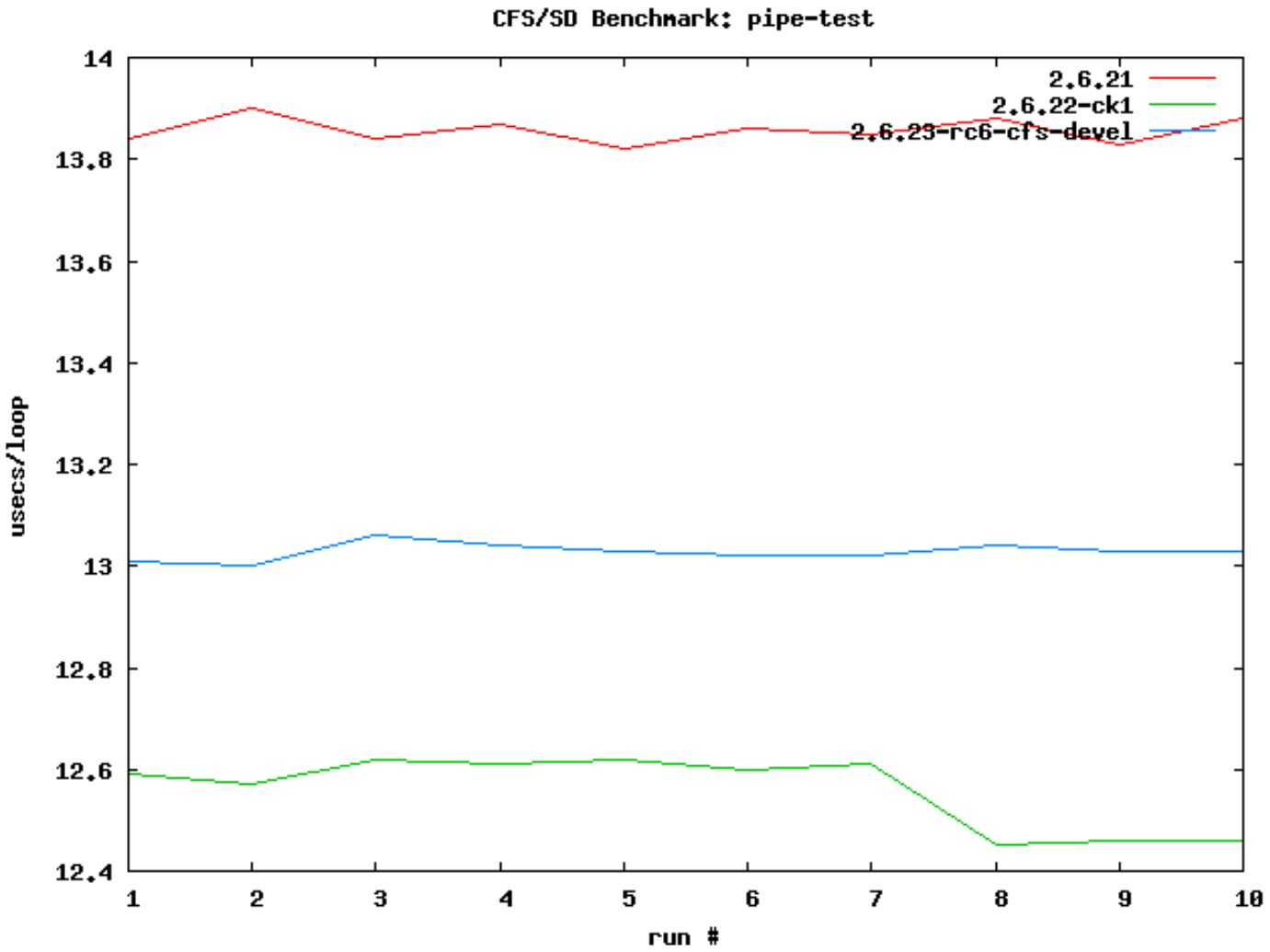
Benchmarki

Benchmarki opublikowane przez Roba Husseya, porównujące stary scheduler $O(1)$, Staircase Deadline Cona Kolivasa i Completely Fair Scheduler.

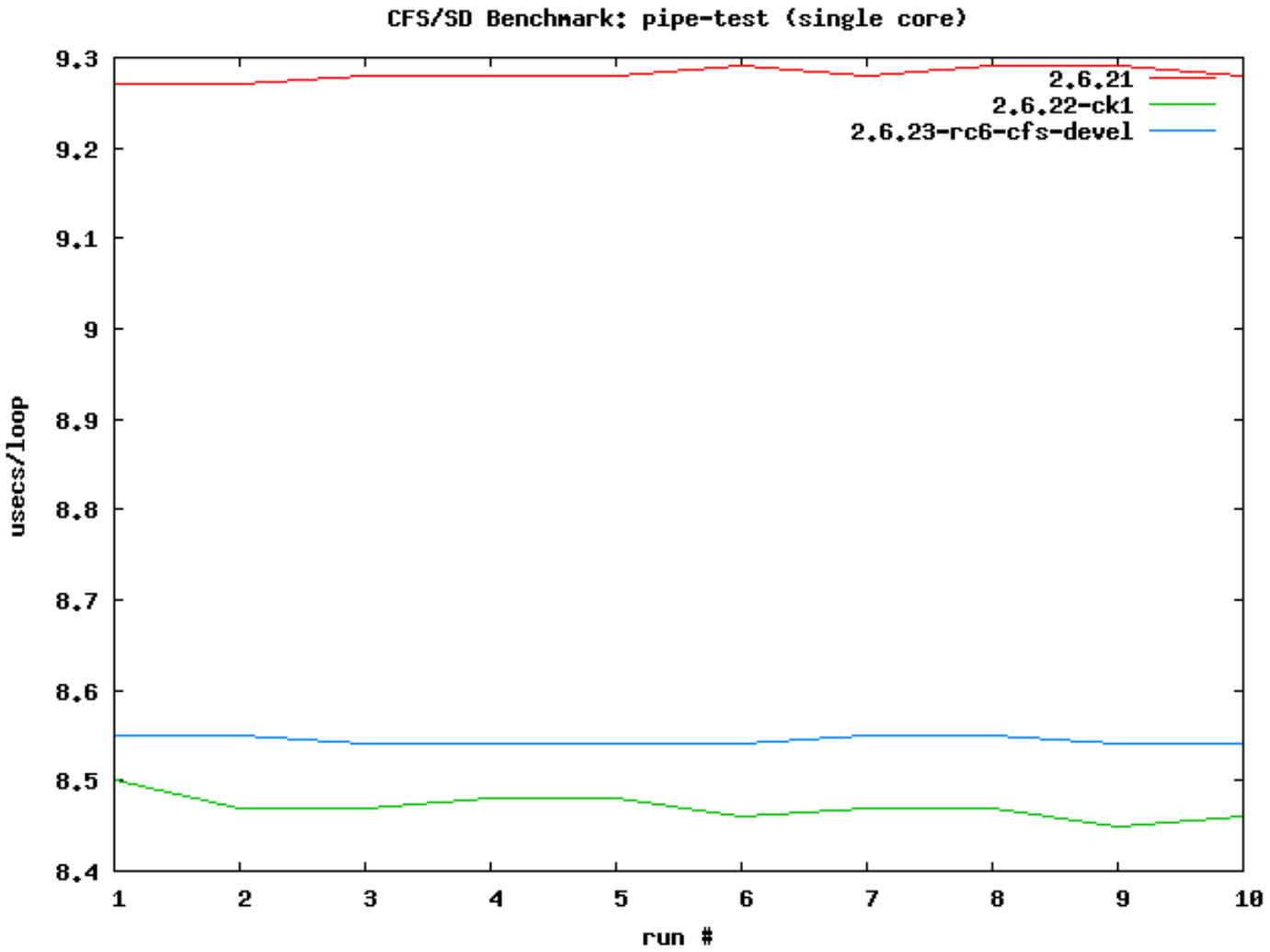
- `pipe-test` – dwa procesy wymieniające się komunikatami poprzez pipe.
- `lat-ctx` – procesy połączone pipe'ami w pierścień, przekazujące po tym pierścieniu komunikaty. Po odebraniu komunikatu pracują (intensywnie używają procesora) a następnie wysyłają komunikat dalej.
- `hackbench` – Tworzy grupy wysyłające i grupy odbierające. Każdy proces grupy wysyłającej wysyła wiadomości po pipe/socket do każdego procesu z grupy odbierającej.

Benchmarki zostały wykonane na jednym oraz na dwóch rdzeniach.

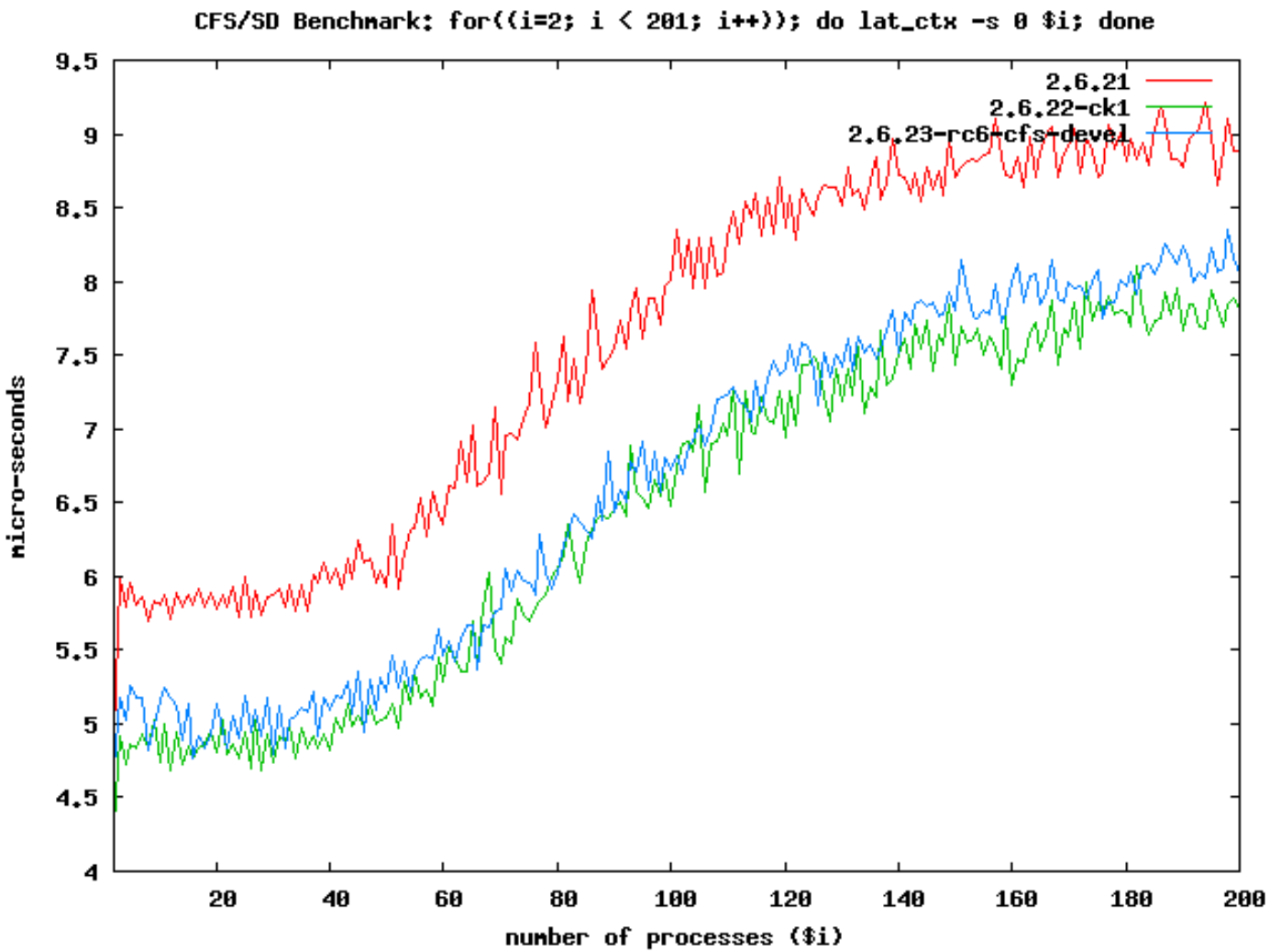
pipe-test


CFS/SD Benchmark: pipe-test
2.6.21
2.6.22-ck1
2.6.23-rc6-cfs-devel
usecs/loop
run #
14
13.8
13.6
13.4
13.2
13
12.8
12.6
12.4
1
2
3
4
5
6
7
8
9
10

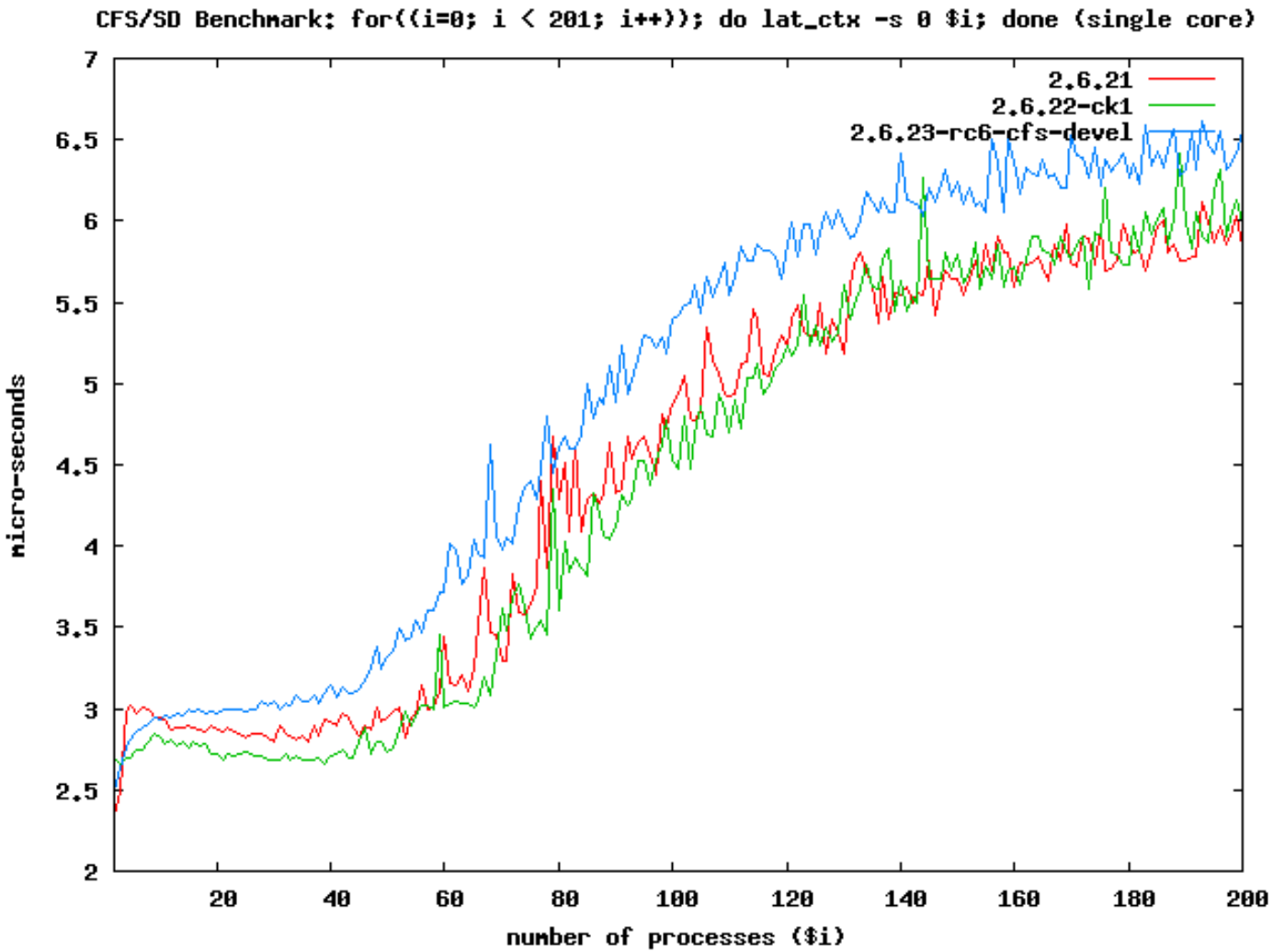
pipe-test – jeden rdzeń



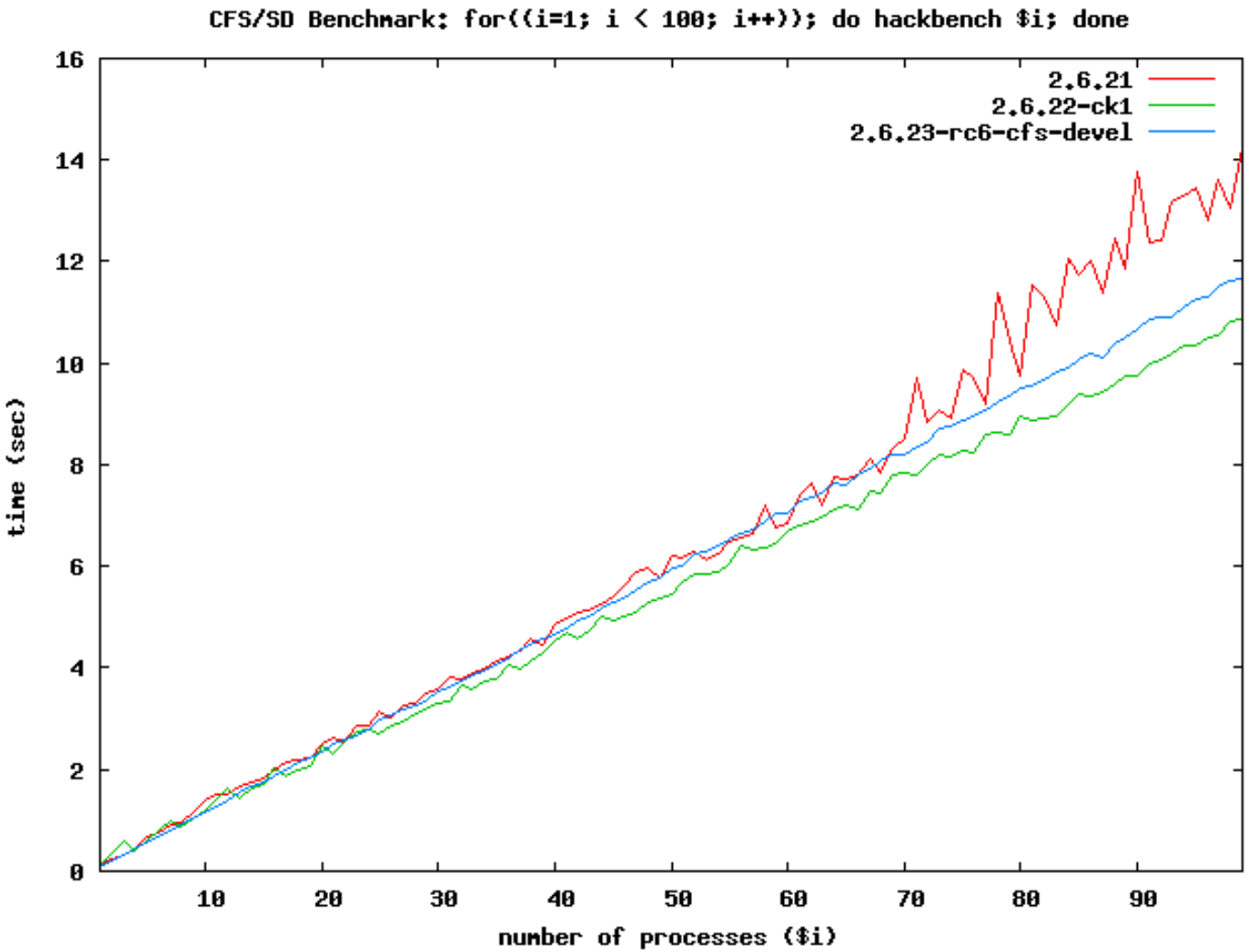
lat-ctx


CFS/SD Benchmark: for((i=2; i < 201; i++)); do lat_ctx -s 0 $i; done
2.6.21
2.6.22-ck1
2.6.23-rc6-cfs-devel
micro-seconds
number of processes ($i)
9.5
9
8.5
8
7.5
7
6.5
6
5.5
5
4.5
4
20
40
60
80
100
120
140
160
180
200

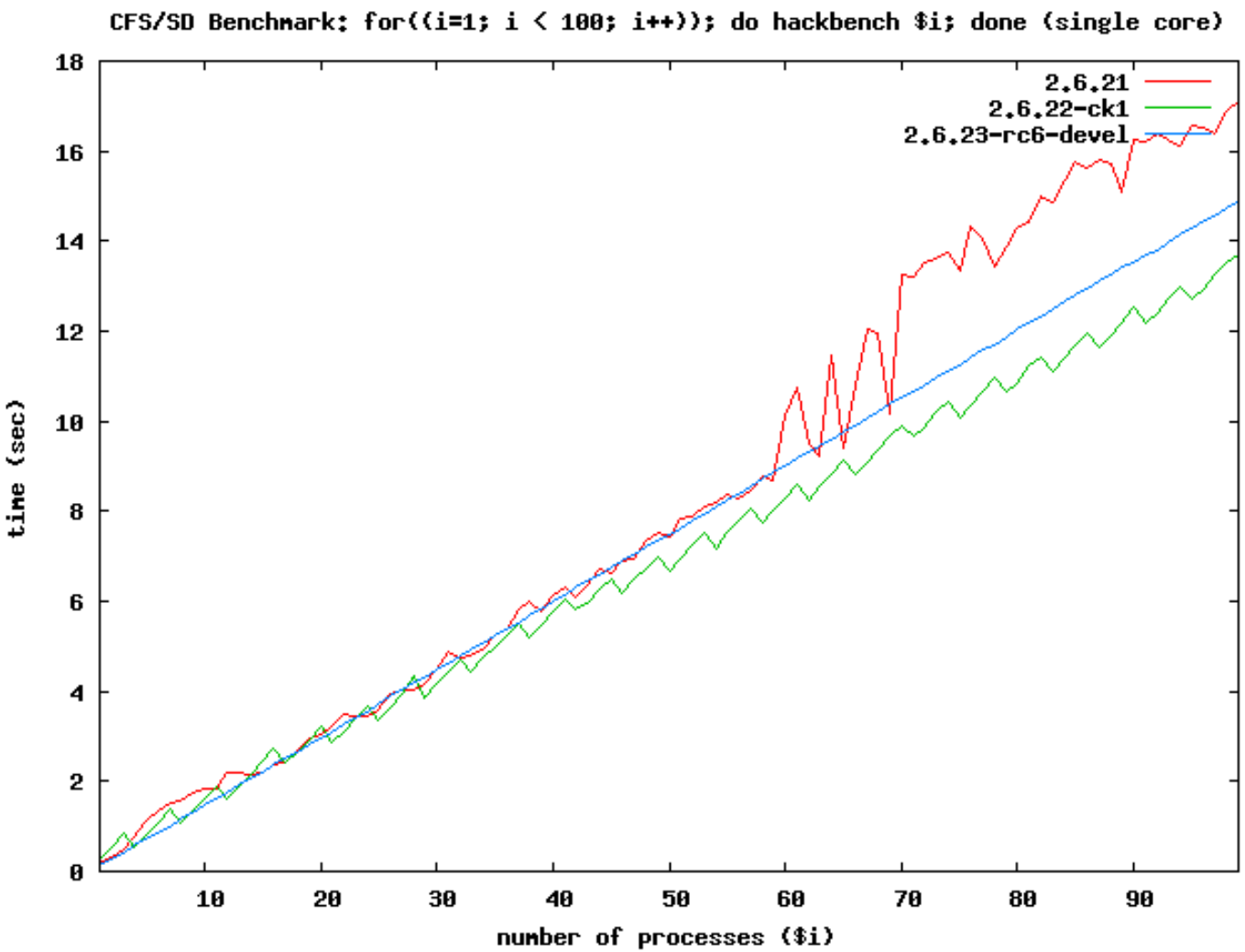
lat-ctx – jeden rdzeń



hackbench


CFS/SD Benchmark: for((i=1; i < 100; i++)); do hackbench $i; done
2.6.21
2.6.22-ck1
2.6.23-rc6-cfs-devel
time (sec)
number of processes ($i)
0
2
4
6
8
10
12
14
16
10
20
30
40
50
60
70
80
90

hackbench – jeden rdzeń



Wniosek

Completely Fair Scheduler został włączony do jądra Linuksa, mimo że Staircase Deadline Cona Kolivasa na testach wychodził tak samo dobrze lub nieco lepiej.

Spis treści

Spis treści

Brain Fuck Scheduler

- Powrót Cona Kolivasa.
- Pierwszy patch – sierpień 2009.
- Nazwa schedulera nie ma nic wspólnego z językiem programowania, lecz z tym, że jest on bardzo prosty w budowie. Zmniejsza o ok. 9000 linii wielkość kodu schedulera.
- Z założenia **nie** ma się dobrze skalować na komputery o większej niż kilka liczbie procesorów/rdzeni i napisany jest tak, aby nigdy nie zostać włączonym do głównego kodu Linuksa.

Brain Fuck Scheduler – budowa

- Powrót do konstrukcji z kernela 2.4 – tylko jedna globalnie blokowana kolejka na wszystkie procesory (argument: równoważenie obciążenia jest drogie, trudne i niedoskonałe, a w komputerach domowych zazwyczaj i tak są co najwyżej 2 lub 4 procesory).
- Zadania RT bez zmian, natomiast zadania użytkownika tylko na trzech kolejkach:
  - `SCHED_ISO` – kolejka dla zadań użytkownika symulujących działanie zadań RT.
  - `SCHED_NORMAL` – jedna lista dla wszystkich normalnych zadań.
  - `SCHED_IDLE` – osobna kolejka dla zadań szeregowanych tylko, gdy procesor jest bezczynny.

Brain Fuck Scheduler – budowa

- Zadanie wstawiane do kolejki otrzymuje kwant czasu równy `rr_interval` oraz deadline:

  ```
  jiffies + (prio_ratio * rr_interval)
  ```

  gdzie `prio_ratio`, podobnie jak wagi w CFS, zależy geometrycznie od priorytetu.
- Jeżeli ustalony deadline jest wcześniejszy od deadline'u procesu wykonywanego na którymś z procesorów, nowy proces od razu go wywłaszcza.
- System wybiera zadanie przeglądając w czasie $O(n)$ (!) całą listę zadań. Jeśli natrafi na zadanie z przeterminowanym deadline, od razu je wykonuje. W przeciwnym razie zaczyna wykonywać ten z najbliższym deadlinem.
- Proste dodatkowe mechanizmy faworyzujące wykonywanie zadań na tym samym procesorze na którym wykonywane były dotychczas.

Brain Fuck Scheduler – konfiguracja

Tylko dwa parametry konfiguracyjne:

- `/proc/sys/kernel/rr_interval` – wielkość kwantu czasu, domyślnie 6ms.
- `/proc/sys/kernel/iso_cpu` – procent czasu procesora który maksymalnie mogą zająć procesy `SCHED_ISO`, domyślnie 70%.

Odpalanie procesów `SCHED_ISO`:

```
schedtool -I -e program
```

Odpalanie procesów `SCHED_IDLE`:

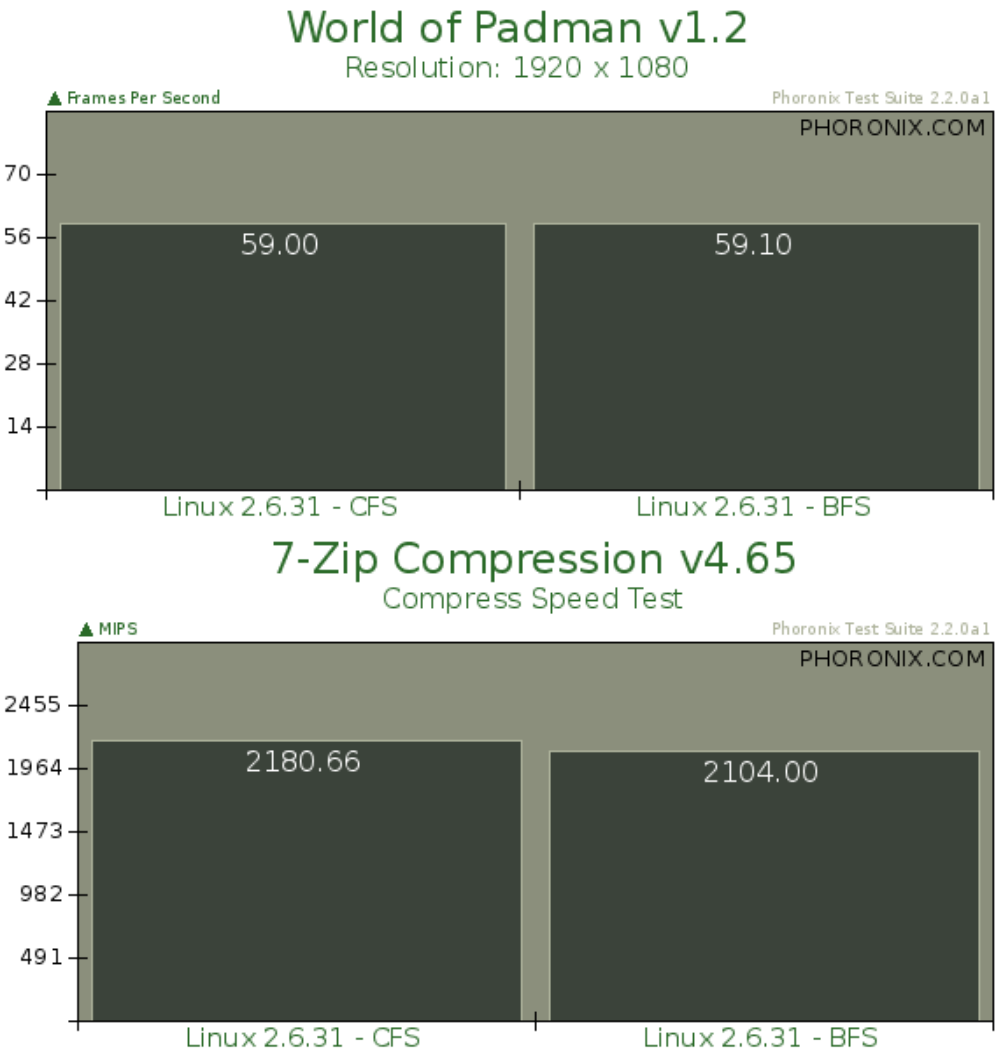
```
schedtool -D -e program
```

Benchmarki

Dwa benchmarki porównujące nowy BFS do CFSa:

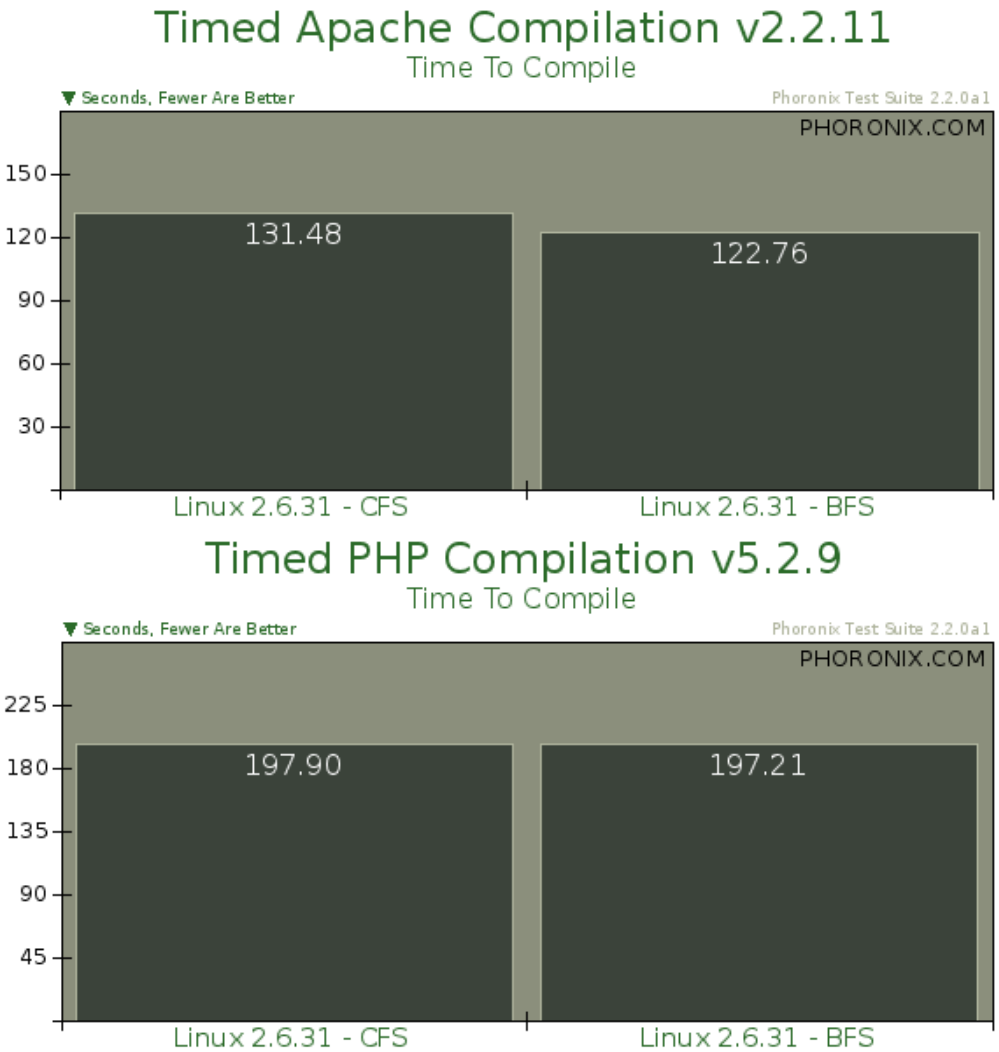
- Testy portalu `phoronix.com`, pokazujące przewagę schedulera BFS.
  - dual-core Intel Atom 330 CPU, hyperthreading, 2.10GHz = 4 wątki wykonania.
- Testy dobrane przez Ingo Molnara, pokazujące miażdżącą przewagę CFS.
  - Dual quad core, hyperthreading = 16 wątków wykonania (taki Con Kolivas prognozował górny limit dla skalowalności jego schedulera, więc Ingo taki wziął).

Phoronix.com – gry, kompresja



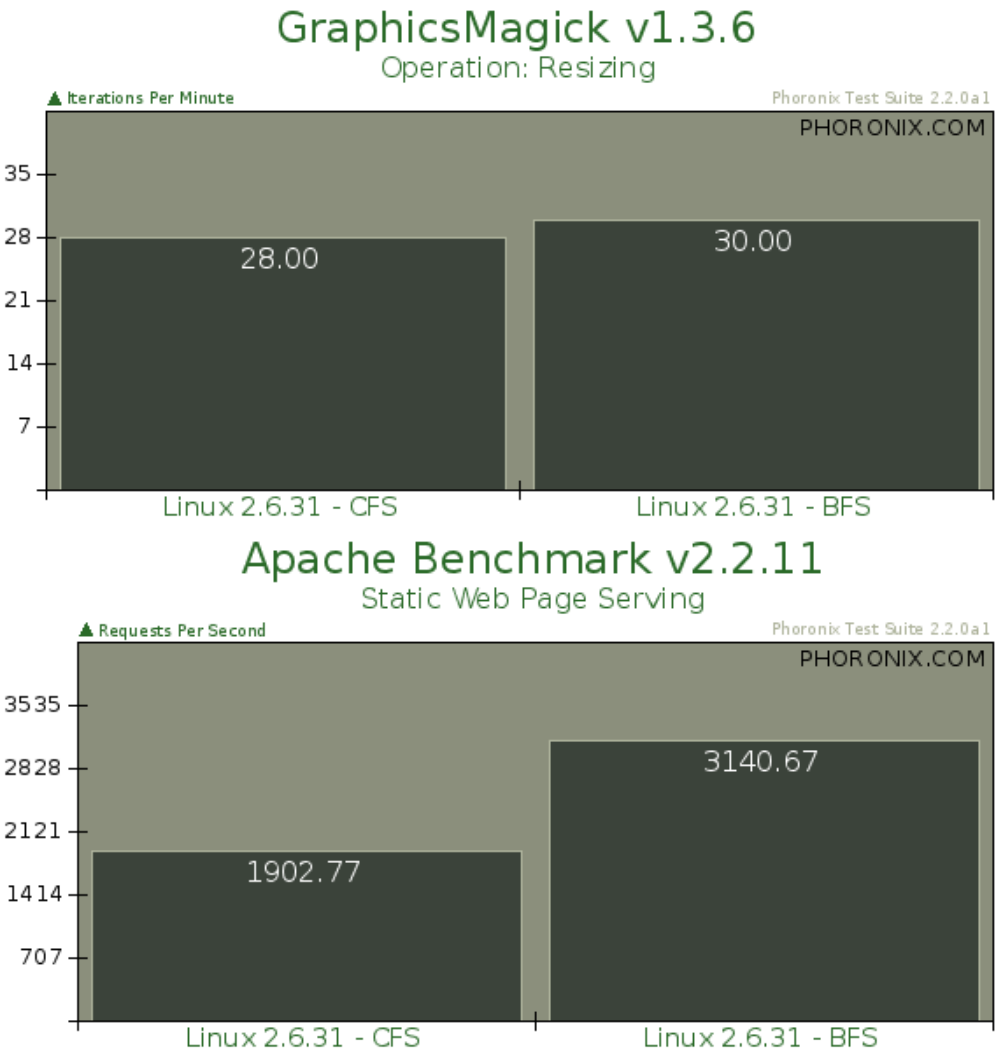
- W grze oba schedulery osiągały podobną liczbę klatek na sekundę.
- CFS osiągał nieznacznie lepszą wydajność przy kompresji plików.

Phoronix.com – kompilacja



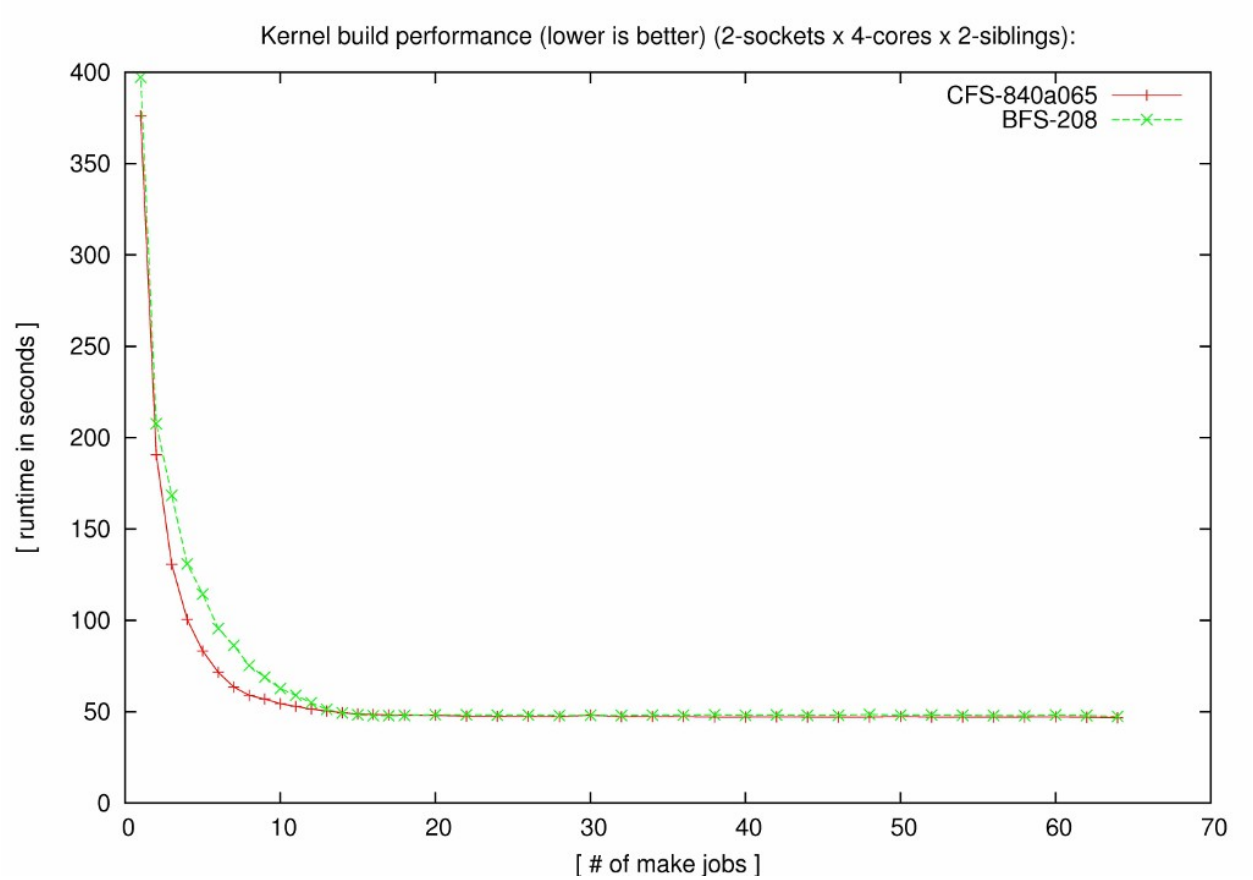
Kompilacja odbywała się na 8 jobach (dwa razy więcej niż wątków wykonania). Widać nieznaczną przewagę BFS.

Phoronix.com – grafika, Apache

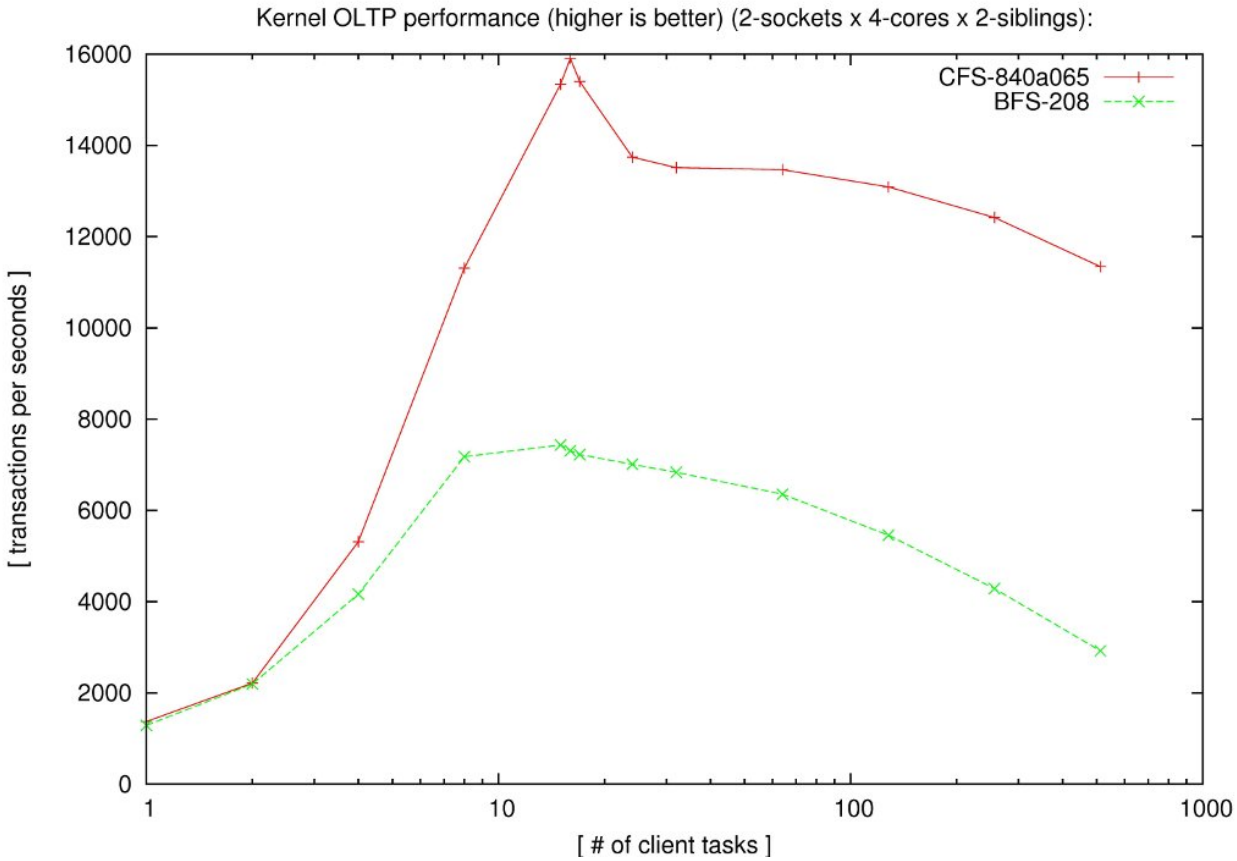


- BFS nieco lepiej radził sobie przy obróbce grafiki (w programie, który zrównoleglał operacje przy wykorzystaniu biblioteki OpenMP).
- Największa różnica na korzyść BFS była jednak w wydajności serwowania stron internetowych przez serwer Apache.

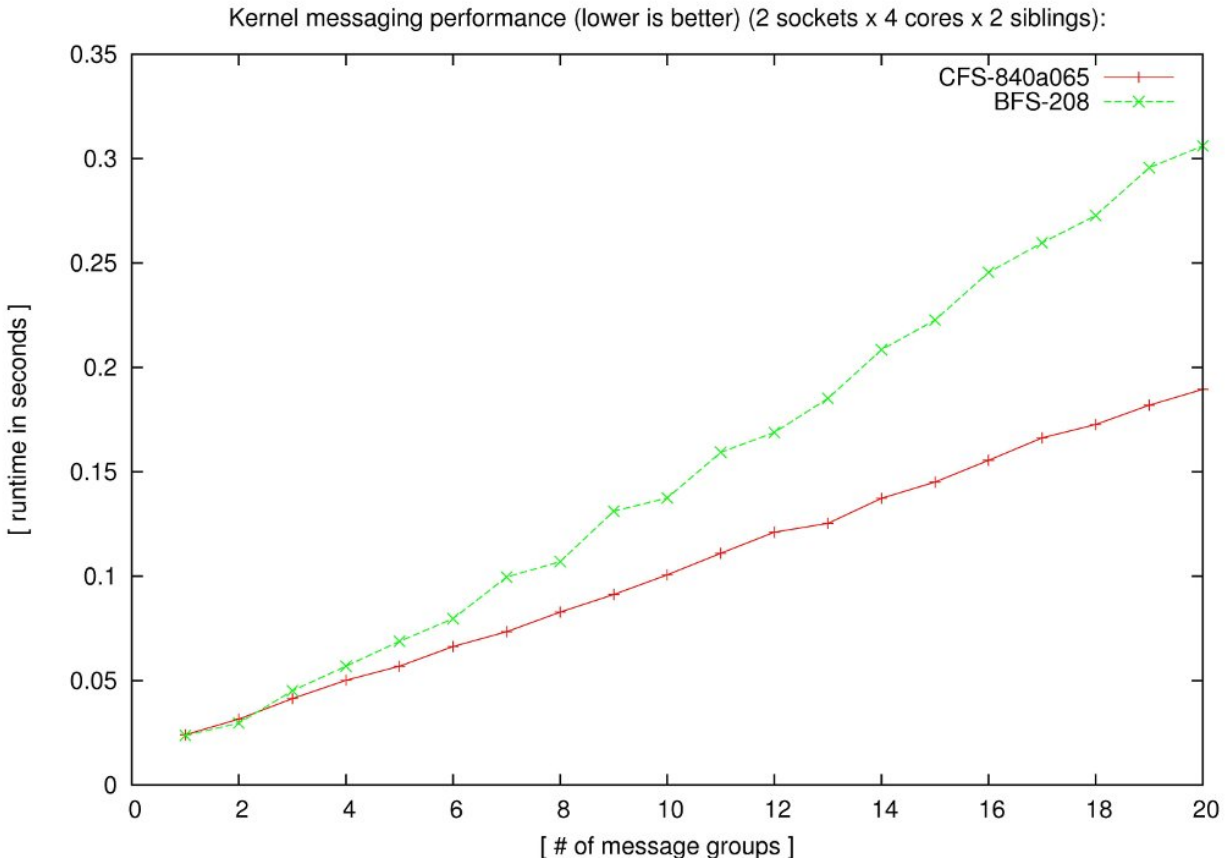
Benchmark Ingo Molnara – kompilacja kernela



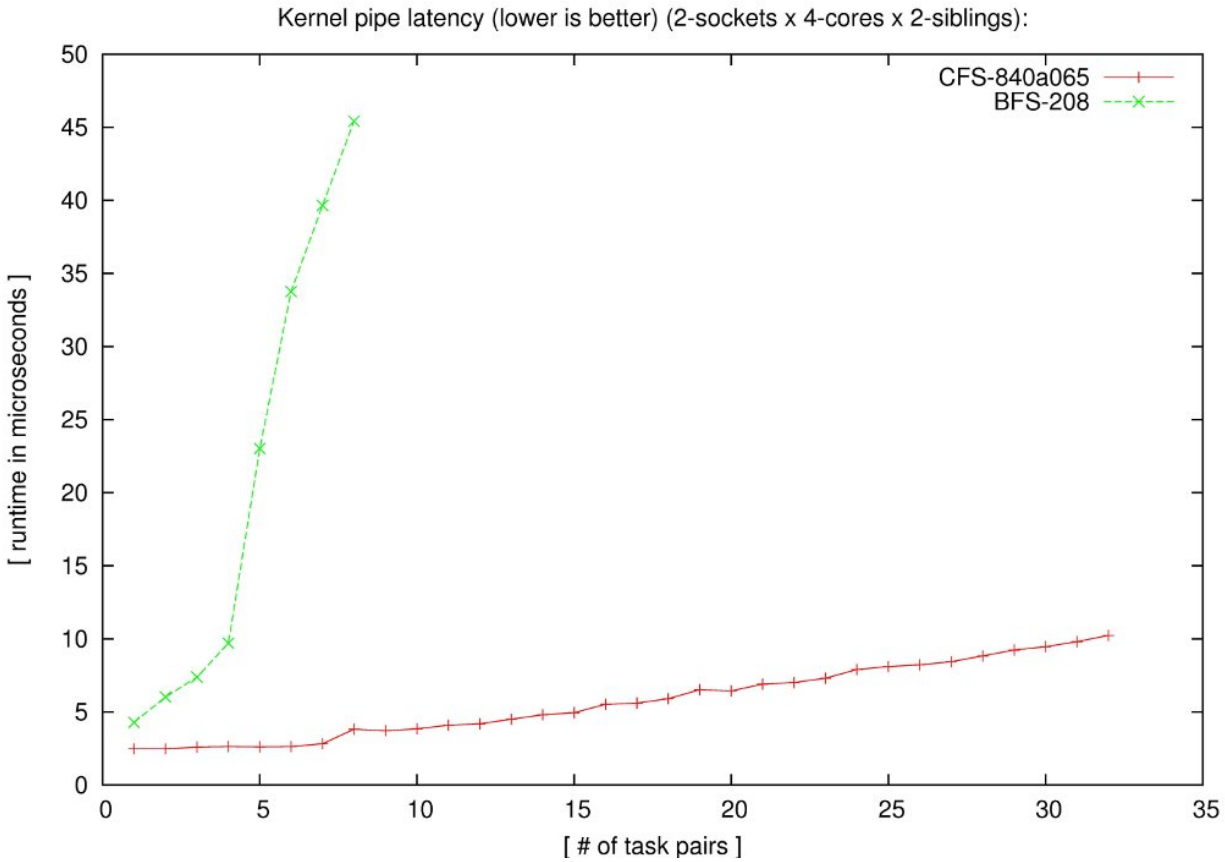
Benchmark Ingo Molnara – wydajność bazy danych (postgreSQL)



Benchmark Ingo Molnara – hackbench



Benchmark Ingo Molnara – pipe-test



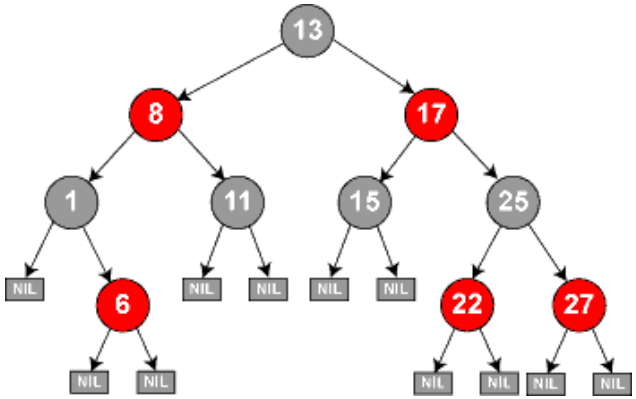
Benchmarki Ingo Molnara – reakcja Cona Kolivasa

Con Kolivas, 7 Sep 2009

Hard to keep a project under wraps and get an audience at the same time, it is. I do realise it was inevitable LKML would invade my personal space no matter how much I didn't want it to, but it would be rude of me to not respond. (...)
/me sees Ingo run off to find the right combination of hardware and benchmark to prove his point.
[snip lots of bullshit meaningless benchmarks showing how great cfs is and/or how bad bfs is, along with telling people they should use these artificial benchmarks to determine how good it is, demonstrating yet again why benchmarks fail the desktop]
Do you know what a normal desktop PC looks like? No, a more realistic question based on what you chose to benchmark to prove your point would be: Do you know what normal people actually do on them?
Feel free to treat the question as rhetorical.
Regards, -ck

Spis treści

API drzew czerwono-czarnych



Używane w takich rejonach kernela jak:

- Schedulery IO – anticipatory, deadline, CFQ
- High Resolution Timers
- System plików ext3
- Virtual Memory Areas
- Hierarchical Token Bucket scheduler

API drzew czerwono-czarnych

- #include<linux/rbtree.h>
- /lib/rbtree.c
- Zorganizowane w sposób podobny jak API list. struct rb_node należy umieścić jako pole struktury którą chcemy trzymać na drzewie.
- Struktura struct rb_root służąca jako korzeń drzewa.

```
struct rb_node
{
    unsigned long  rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
struct rb_root
{
    struct rb_node *rb_node;
};

#define rb_entry(ptr, type, member) container_of(ptr, type, member)
#define container_of(ptr, type, member) ({                      \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

Funkcje API

API dostarcza funkcje:

```
struct rb_node *rb_first(struct rb_root *tree);
struct rb_node *rb_last(struct rb_root *tree);
struct rb_node *rb_next(struct rb_node *node);
struct rb_node *rb_prev(struct rb_node *node);
void rb_erase(struct rb_node *victim, struct rb_root *tree);
void rb_replace_node(struct rb_node *old,
                     struct rb_node *new,
                     struct rb_root *tree);

void rb_link_node(struct rb_node * node, struct rb_node * parent,
                  struct rb_node ** rb_link);
void rb_insert_color(struct rb_node *node, struct rb_root *root);
```

Funkcja wyszukująca

Należy napisać własną funkcję wyszukującą w drzewie.

```
struct my_stuff *my_rb_search(struct rb_root *root, int value)
{
    struct rb_node *node = root->rb_node;  /* top of the tree */

    while (node)
    {
        struct my_stuff *stuff = rb_entry(node, struct my_stuff, node);

        if (stuff->coolness > value)
            node = node->rb_left;
        else if (stuff->coolness < value)
            node = node->rb_right;
        else
            return stuff;  /* Found it */
    }
    return NULL;
}
```

Funkcja wstawiająca

Oraz własną funkcję wstawiającą do drzewa.

```
void my_rb_insert(struct rb_root *root, struct my_stuff *new)
{
    struct rb_node **link = &root->rb_node, *parent;
    int value = new->coolness;

    /* Go to the bottom of the tree */
    while (*link)
    {
        parent = *link;
        struct my_stuff *stuff = rb_entry(parent, struct my_stuff, parent);
        if (stuff->coolness > value)
            link = &(*link)->rb_left;
        else
            link = &(*link)->rb_right;
    }

    /* Put the new node there */
    rb_link_node(new, parent, link);
    rb_insert_color(new, root);
}
```

Spis treści

Bibliografia I

- Przede wszystkim: źródła linuxa
- Kernel: Documentation/scheduler/sched-design-CFS.txt
- http://students.mimuw.edu.pl/SO/Wyklady-html/03_schedule/3_schedule.html
- http://students.mimuw.edu.pl/SO/Wyklady-html/04_pamiec/4_cw-schedLx.html
- http://rainbow.mimuw.edu.pl/SO/Projekt07-08/temat3-g3/index.html
- http://lkml.indiana.edu/hypermail/linux/kernel/0303.1/1455.html
- http://lkml.org/lkml/2007/3/26/319
- http://lkml.org/lkml/2009/9/6/231

Bibliografia II

- http://www.linuxjournal.com/article/7178?no_loop=1
- http://kerneltrap.org/taxonomy/term/607
- http://kerneltrap.org/Linux/Additional_CFS_Benchmarks
- http://www.ibm.com/developerworks/linux/library/l-cfs/
- http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/
- http://www.scribd.com/doc/24111564/Project-Linux-Scheduler-2-6-32
- http://www.linux-foundation.jp/jp_uploads/seminar20080709/lfjp2008.pdf
- http://gustavus.edu/+max/os-book/updates/CFS.html

Bibliografia III

- http://kerneltrap.org/node/8059
- http://lwn.net/Articles/230574/
- http://lwn.net/Articles/351058/
- http://www.fizyka.umk.pl/~jkob/prace-mag/cfs-tuning.pdf
- http://www.phoronix.com/scan.php?page=article&item=bfs_scheduler_benchmarks&num=1
- http://ck.kolivas.org/patches/bfs/
- http://ck.kolivas.org/patches/staircase-deadline/
- http://ck.wikia.com/wiki/RSDL
- http://www.linuxpromagazine.com/Online/News/Con-Kolivas-Introduces-New-BFS-Scheduler

Spis treści

Pytania