Szeregowanie procesów w Linuksie

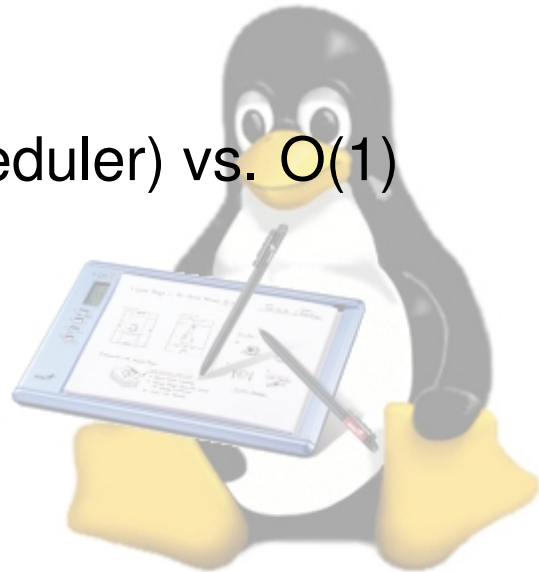
trendy rozwojowe

Agata Hejmej Grzegorz Maj Grzegorz Ziemiański

07.12.2007

Plan prezentacji

- wprowadzenie
- Scheduler 2.4 vs. Scheduler O(1)
- zmiany, zmiany...
- RSDL/SD vs. CFS (Completely Fair Scheduler) vs. O(1)
- testowanie schedulerów



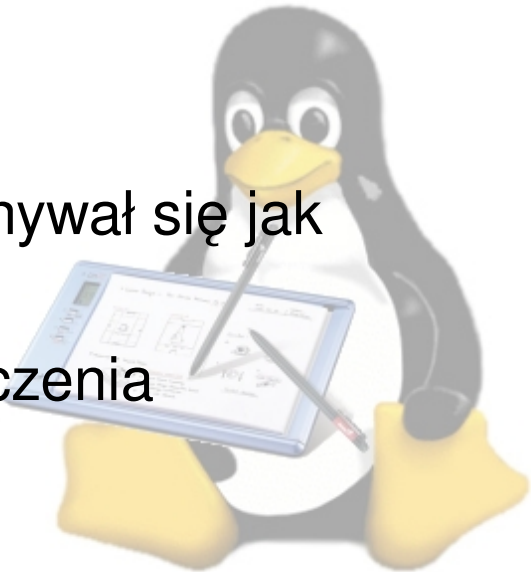
Wprowadzenie

SCHEDULER

to algorytm decydujący, który proces ma w danej chwili korzystać z procesora

Jaki powinien być?

- powinien być szybki
- szeregować procesy tak, żeby sam wykonywał się jak najrzadziej
- nie dopuszczać do zagłodzenia i zakleszczenia
- maksymalizować efekt współbieżności



Wprowadzenie

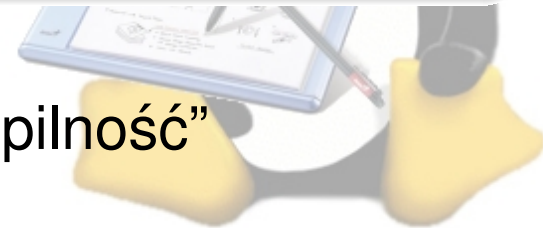
Co jest wspólne dla wszystkich omawianych schedulerów?

- podział na procesy czasu rzeczywistego (obsługiwane w pierwszej kolejności) i zwykłe

PROCESY CZASU RZECZYWISTEGO

(ang. realtime, RT) procesy, które muszą uzyskać dostęp do procesora w określonym przedziale czasu; np. urządzenia pomiarowe, obsługa taśm produkcyjnych. Mogą być uruchamiane tylko przez superużytkownika.

- przydzielanie procesom kwantów czasu
- procesy mają priorytety, określające ich “pilność” wykonania
- są procesy interakcyjne i zorientowane na CPU

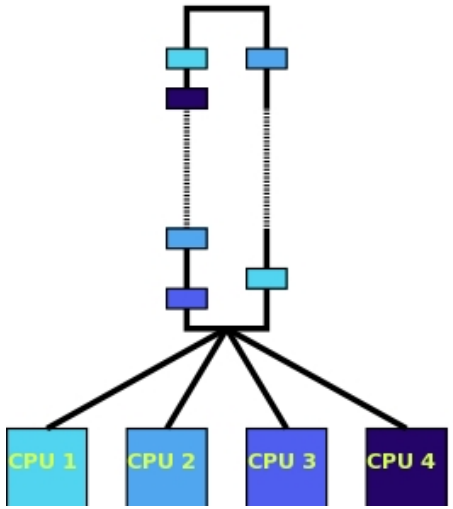


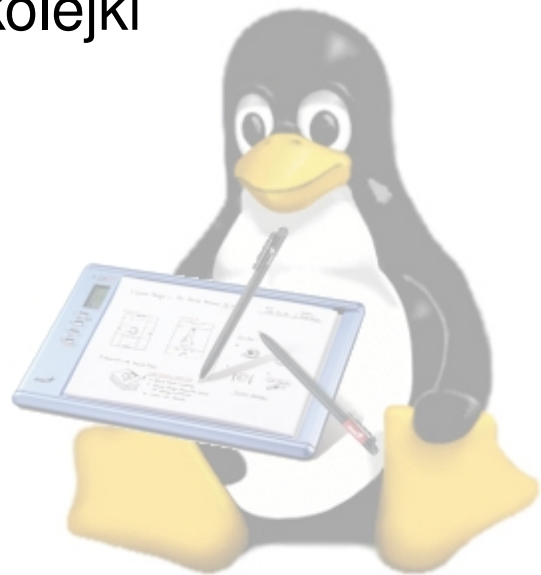
Scheduler 2.4

Ogólne zasady działania

Jest jedna kolejka procesów gotowych, w związku z tym:

- są na niej zarówno procesy, które zużyły swój kwant czasu jak i te, które nie wykorzystały jeszcze całego
- wszystkie procesory korzystają z jednej kolejki





Scheduler 2.4

Ogólne zasady działania

Zmiana epoki, czyli przydzielanie nowych kwantów

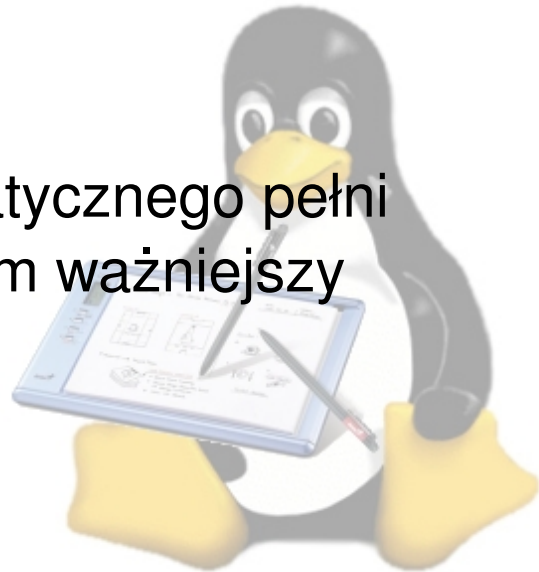
KWANT CZASU

jest to czas, na który proces ma możliwość dostępu do procesora. Wielkość przydzielanych kwantów zależy od priorytetu statycznego procesu (wartość **nice**). Ilość pozostałego do wykorzystania czasu jest pamiętana w polu **counter** i jest zmniejszana w trakcie działania procesu przez funkcję update_process_times() przy każdym przerwaniu zegara systemowego.

Scheduler 2.4

Ogólne zasady działania

- procesy RT mają priorytety od 1 do 99 (rt_priority), im **wyższy** tym lepszy. Dla zwykłych procesów rt_priority wynosi 0
- dla procesów zwykłych rolę priorytetu statycznego pełni pole nice od -19 do 20, im **niższy** nice tym ważniejszy proces (mniej “nice” dla innych)



Scheduler 2.4

Ogólne zasady działania

Kiedy proces opuszcza procesor?

- kiedy się zakończy
- pojawi się proces ważniejszy
- sam się zrzeknie procesora
- uśnie w oczekiwaniu na zasób
- zależnie od strategii: kiedy skończy mu się kwant czasu

Scheduler 2.4

Strategie szeregowania

Dla procesów czasu rzeczywistego:

SCHED_FIFO

nie ma kwantów czasu; wykonuje się dopóki nie zostanie wywłaszczony. Wtedy zostaje umieszczony na początku kolejki procesów gotowych, tak żeby wykonywał się znowu, gdy tylko skończą ważniejsze

SCHED_RR

proces dostaje procesor maksymalnie na kwant czasu, a po jego zużyciu zostaje wywłaszczony. Funkcja schedule() od razu przydziela mu nowy i umieszcza na końcu kolejki procesów gotowych (sprawiedliwość między procesami o takim samym priorytecie)

Scheduler 2.4

Strategie szeregowania

Dla procesów zwykłych:

SCHED_OTHER

procesy mogą wykonywać się przez kwant czasu, a nowy dostaną dopiero gdy wszystkie inne zużyją swój

Scheduler 2.4

Prawa procesów :)

Czy wiesz, że.. ?

sched_yield()

procesy mogą dobrowolnie zrezygnować z procesora, wywołując funkcję **sched_yield()**.
Zostają wtedy umieszczone na końcu kolejki procesów gotowych.
Procesy zwykłe zostają dodatkowo oznaczone jako SCHED_YIELD, co powoduje, że następny wybrany proces może mieć mniejszy priorytet.
Dla RT sched_yield() w schedulerze 2.4 nie działa poprawnie. Jego wywołanie nie da żadnego efektu.

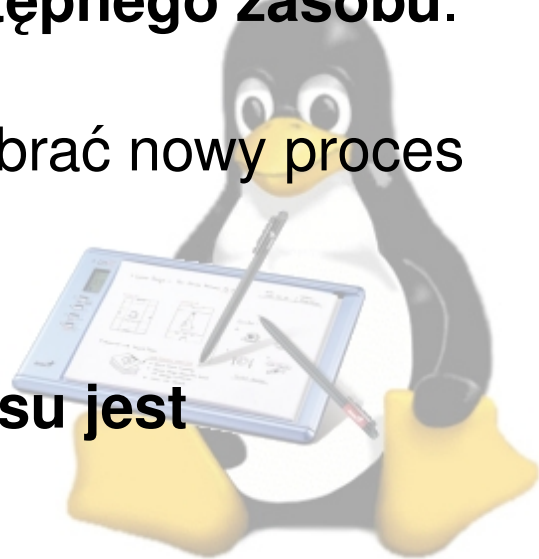
Scheduler 2.4

SCHEDULE()

Główną funkcją szeregującą procesy jest **schedule()**.

schedule() jest wywoływany gdy:

- **wykonujący się proces zażąda niedostępnego zasobu**. Wtedy jego stan zmieniany jest na TASK_(UN)INTERRUPTIBLE i trzeba wybrać nowy proces do wykonania
- **wykonujący się proces zakończy się**
- pole **need_resched działającego procesu jest ustawione na 1**



Scheduler 2.4

SCHEDULE()

Skąd się bierze need_resched.. ?

need_resched

jest ustawione, gdy:

- **proces wykorzystał kwant czasu** (sprawdzane w update_process_times())
- **proces zrzekł się procesora** (sched_yield())
- **pojawił się gotowy proces o wyższym priorytecie** (sprawdzane w reschedule_idle())
- **wywołanie funkcji sched_setscheduler(..) uczyniło któryś proces procesem klasy RT**

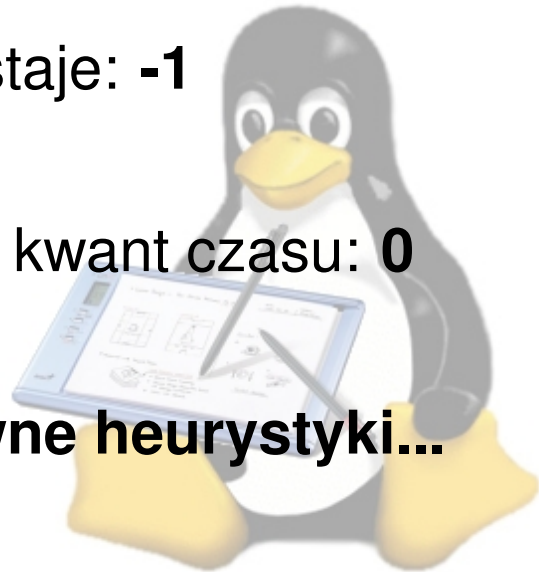
Scheduler 2.4

GOODNESS()

Funkcja schedule() wybiera procesy według ich ważności (priorytetu dynamicznego) ustalonej przez funkcję **goodness()**.

Zasady priorytetowania:

- jeśli proces ma flagę SCHED_YIELD dostaje: **-1**
- jest klasy RT: **1000 + jego rt_priority**
- jest zwykłym procesem i skończył się mu kwant czasu: **0**



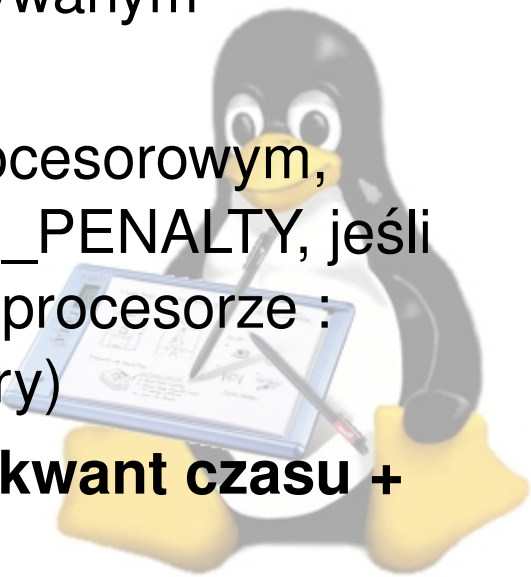
Wybierając procesy zwykłe stosujemy pewne heurystyki...

Scheduler 2.4

GOODNESS()

Zasady dla procesów zwykłych z niezerowym kwantem:

- do priorytetu **wliczamy pozostały kwant czasu** - faworyzowanie procesów, które dotychczas krócej działały
- jeśli współdzieli pamięć z ostatnio wykonywanym procesem: **bonus +1**
- jeśli scheduler działa w systemie wieloprocesorowym, przydzielany jest bonus CHANGE_PROC_PENALTY, jeśli proces ostatnio wykonywał się na danym procesorze : **bonus +15 lub +20** (zależy od architektury)
- ostatecznym priorytetem jest: **pozostały kwant czasu + (20 - nice) + bonusy**

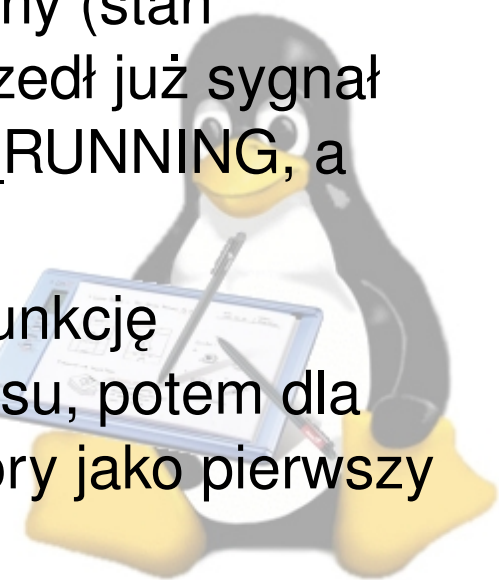


Scheduler 2.4

SCHEDULE() - opis działania

Działanie funkcji schedule() :

- sprawdza czy aktualnemu procesowi nie trzeba odnowić kwantu czasu (SCHED_RR)
- sprawdza czy aktualny proces został uśpiony (stan TASK_(UN)INTERRUPTIBLE) - jeśli przyszedł już sygnał budzący, to przywraca jego stan na TASK_RUNNING, a jeśli nie, usuwa go z runqueue
- wybiera proces do wykonania - wywołuje funkcję goodness(), najpierw dla aktualnego procesu, potem dla całego runqueue. Wybrany jest proces, który jako pierwszy uzyska największą wartość.



Scheduler 2.4

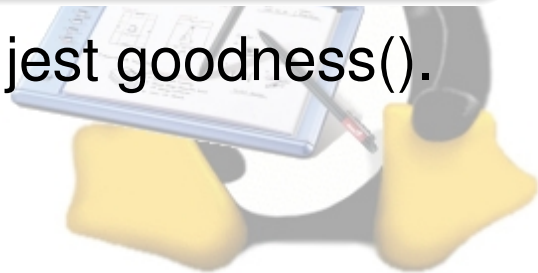
SCHEDULE() - opis działania

- jeśli największy goodness to 0 (kwant czasu wykorzystany), następuje

zmiana epoki

wszystkim procesom wylicza się nowe kwanty, niezależnie od tego czy są w kolejce runqueue, czy np. śpią

Po zmianie epoki ponownie wywoływana jest goodness().

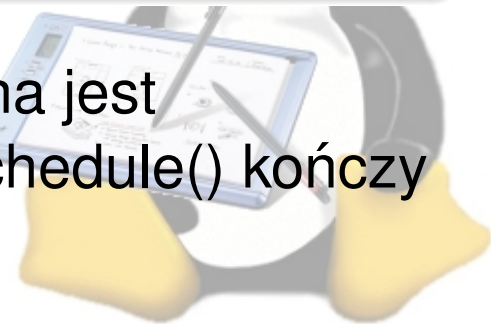


Scheduler 2.4

SCHEDULE() - opis działania

NOWY KWANT CZASU

wielkość nowego kwantu czasu zależy od wartości nice procesu i wynosi od 1 do 10.
Jeśli proces nie wykorzystał całego kwantu w starej epoce (np.śpi), dostanie bonusowo połowę wartości starego kwantu.
Jest to heurystyka faworyzująca procesy interakcyjne



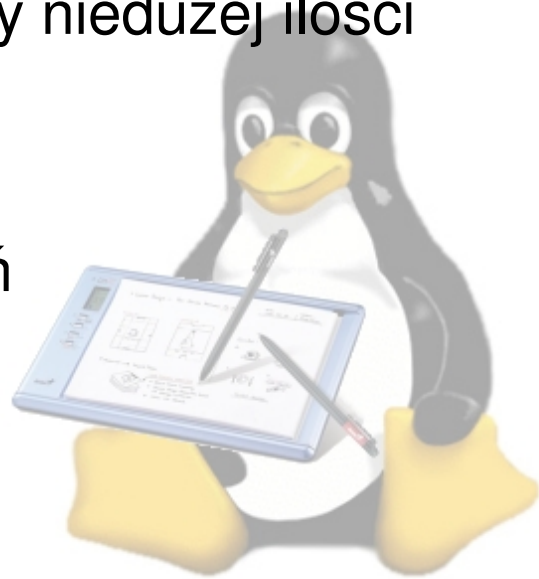
- po wybraniu kolejnego procesu wykonywana jest (ewentualna) zmiana kontekstu i funkcja schedule() kończy działanie

Scheduler 2.4

Podsumowanie - zalety

Czy on był taki zły.. ?

- na jednoprocesorowych maszynach i przy niedużej ilości procesów zachowywał się dobrze
- nie powodował zakleszczeń
- nie doprowadzał do brutalnych zagłodzeń

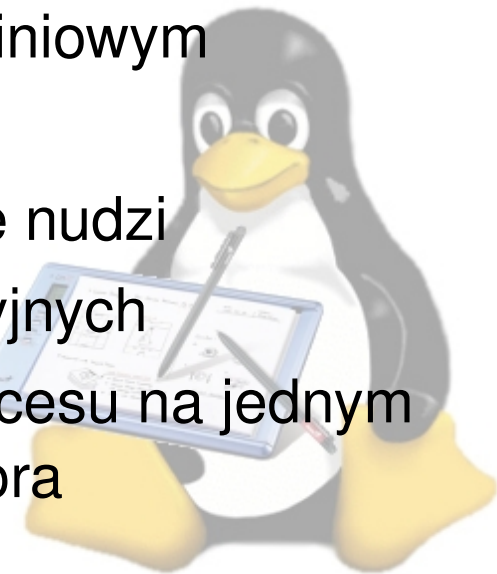


Scheduler 2.4

Podsumowanie - wady

ALE:

- funkcja schedule() zakłada spinlock na runqueue - tylko jeden procesor może naraz korzystać
- wybieranie następnego procesu w czasie liniowym
- przestój na zmianę epoki
- jeden procesor zmienia epokę, a reszta się nudzi
- niewielkie wsparcie dla procesów interakcyjnych
- nieskuteczne heurystyki utrzymywania procesu na jednym procesorze - często tracony cache procesora

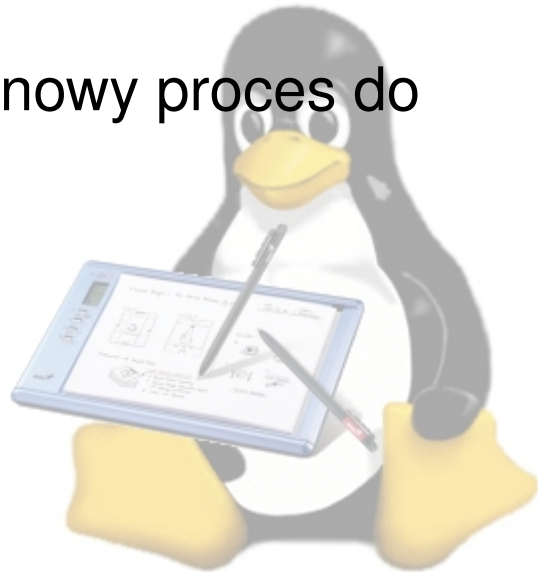


Scheduler O(1)

Wstęp

Dlatego w jądrze 2.6 wprowadzono nowy scheduler.

Nazywany jest **schedulerem O(1)**, ponieważ nowy proces do wykonania wybiera w czasie stałym.
Został napisany przez Ingo Molnara.



Scheduler O(1)

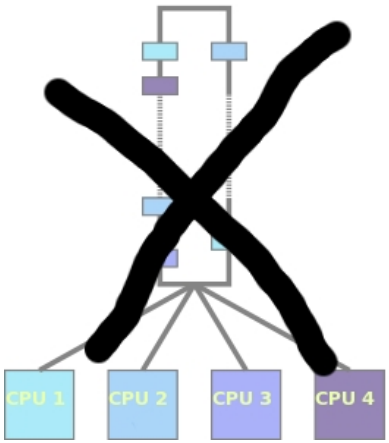
Co nowego?

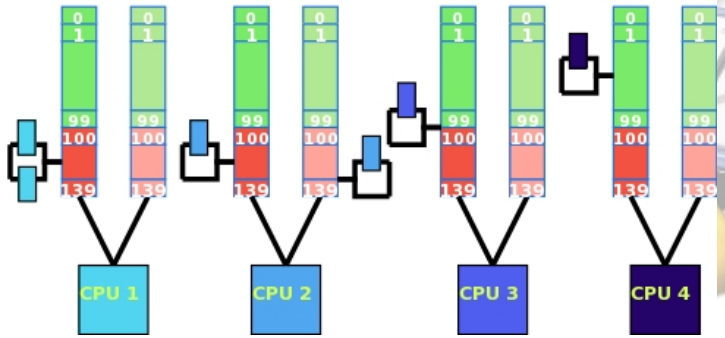
- Scheduler O(1) nie wprowadził rewolucyjnych zmian w sposobie szeregowania procesów
- szeregowanie nadal oparte jest na różnego rodzaju heurystykach (trochę innych)
- wprowadził za to zupełnie nowe struktury danych do przechowywania procesów gotowych, które dają mu wyraźną przewagę nad poprzednikiem

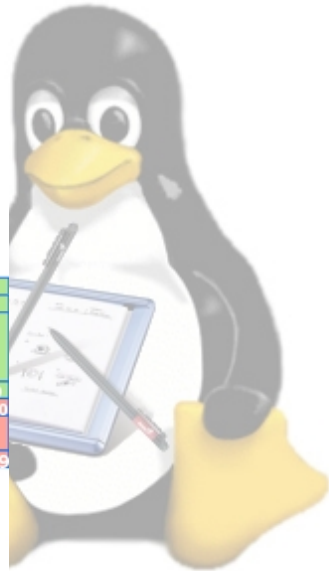
Scheduler O(1)

Nowe strukturki

- teraz każdy procesor ma swoją kolejkę procesów gotowych
- na kolejkę składają się dwie tablice - active i expired - do zmiany epoki wystarczy ich podmiana
- pola tablic (140) stanowią dowiązania do list procesów o priorytecie równym indeksowi





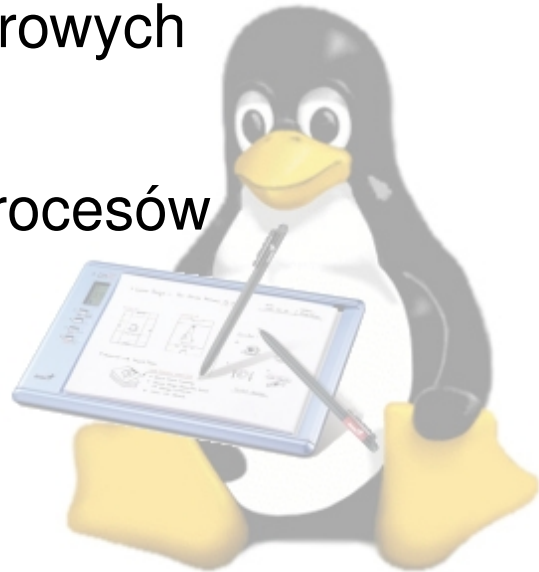


Scheduler O(1)

Najważniejsze cechy

Od razu widać poprawę

- szeregowanie w czasie stałym
- skalowalność w systemach wieloprocesorowych
- dobry związek procesu z procesorem
- nowe heurystyki poprawiające obsługę procesów interaktywnych
- nie boi się dużego obciążenia

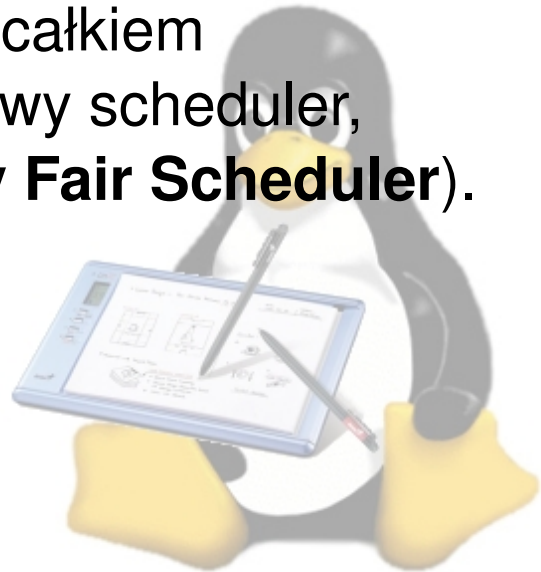


Zmiany, zmiany...

Niektórym to jednak nie wystarcza...

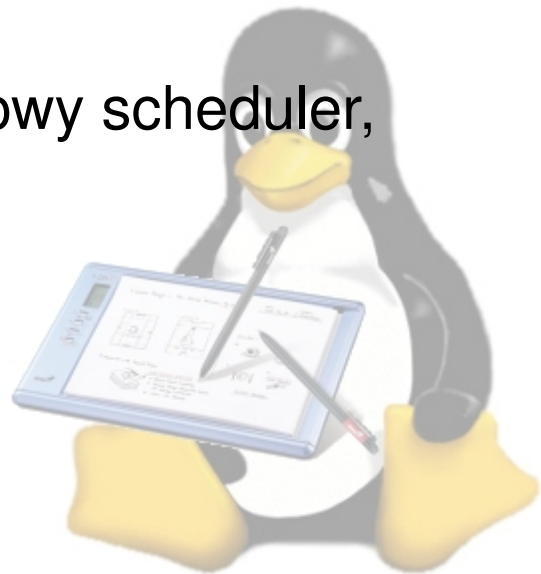
Chociaż Scheduler O(1) został wprowadzony całkiem niedawno, w wersji jądra 2.6.23 pojawił się nowy scheduler, zwany “Całkiem Sprawiedliwym” (**Completely Fair Scheduler**).

Dlaczego ?

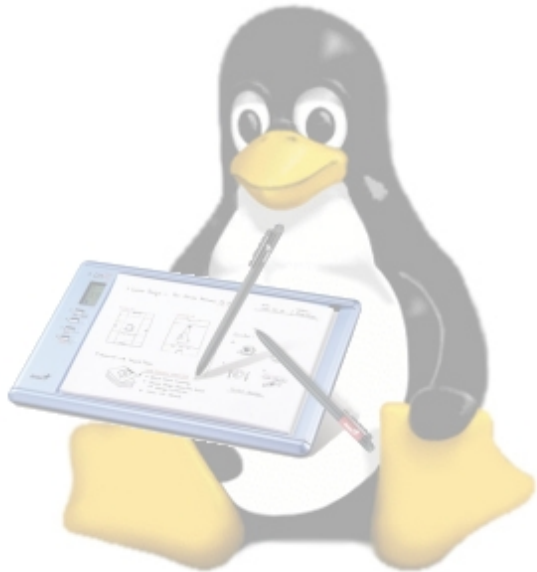


Jak to się zaczęło...

- Con Kolivas - zbuntowany użytkownik;
- SD (Staircase Deadline) - łatki na jądro;
- RSDL (Rotating Staircase DeadLine) - nowy scheduler, rozwinięcie SD.



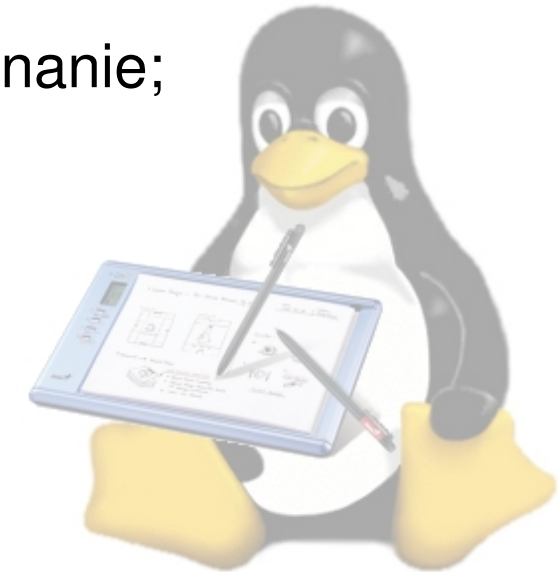
Rotating Staircase DeadLine



RSDL

Cechy RSDLa - według Cona Kolivasa

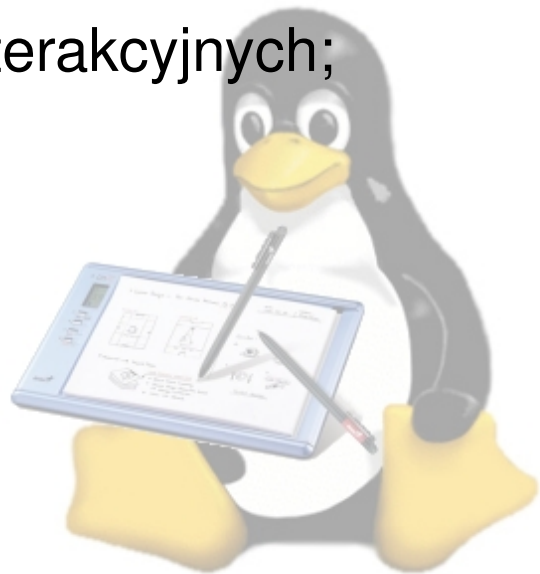
- całkowita sprawiedliwość;
- brak zagłodzeń;
- ograniczone czasy oczekiwania na wykonanie;
- czas działania O(1);
- wysoka skalowalność;
- interaktywność;



RSDL

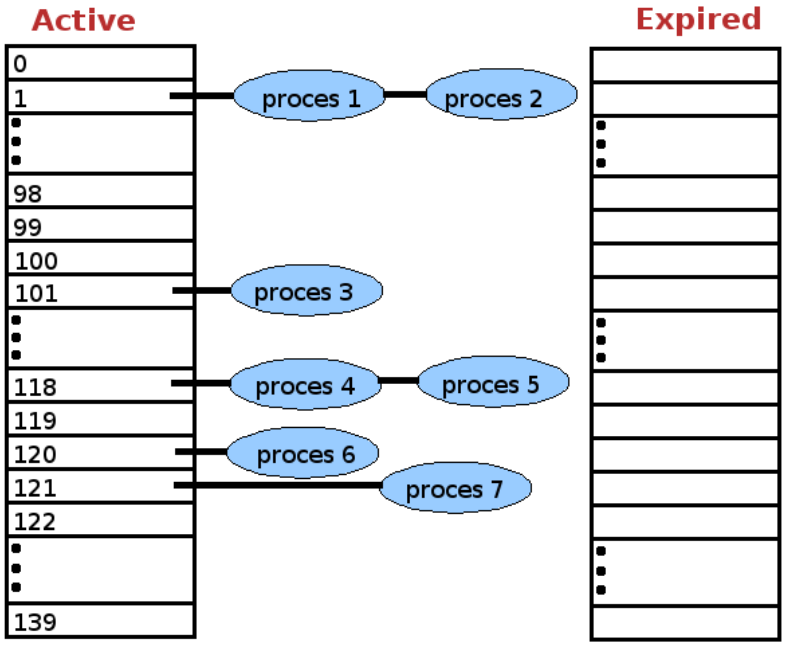
Cechy RSDLa - według Cona Kolivasa cd.

- prosta budowa;
- brak heurystyk dotyczących procesów interakcyjnych;
- brak pomiarów czasu działania/spania;
- tylko proste obliczenia.



RSDL

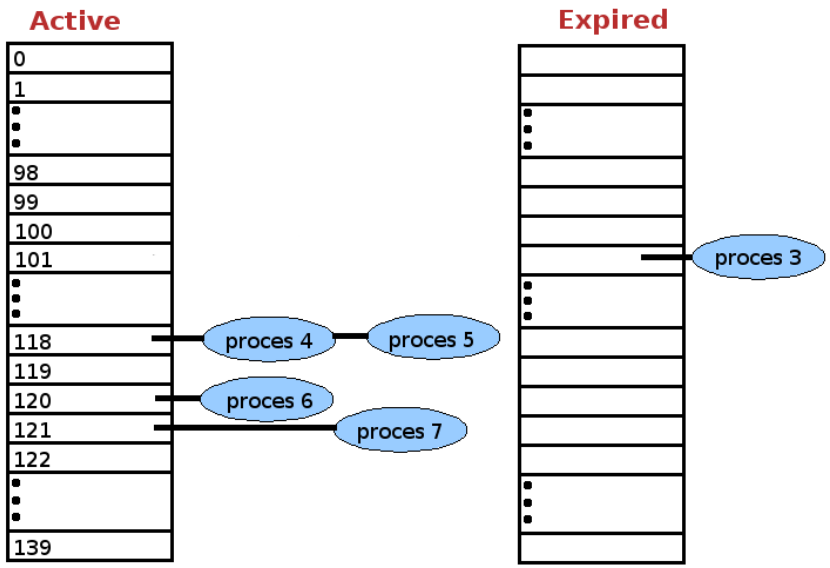
Schemat działania

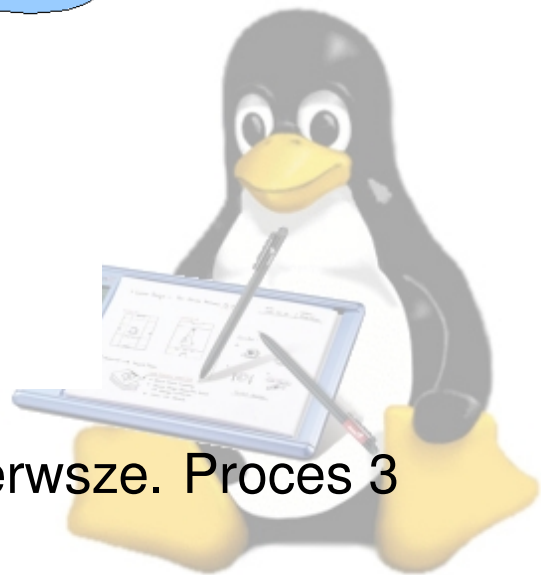


Rysunek: Procesy gotowe do pracy są umieszczone w kolejce Active.

RSDL

Schemat działania cd.cd.

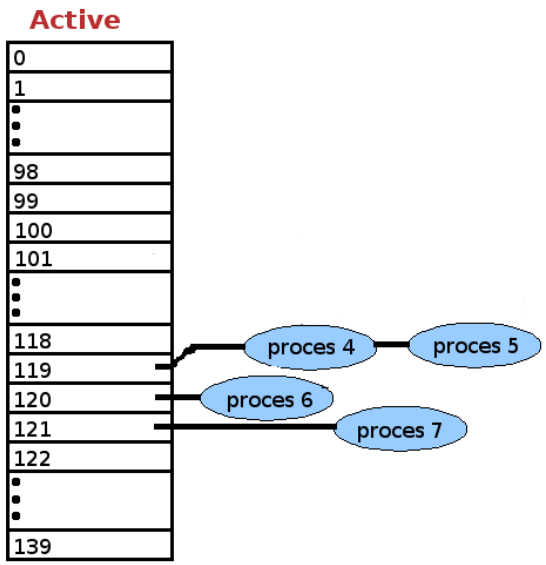




Rysunek: Procesy 1 i 2 (RT) wykonały się jako pierwsze. Proces 3 wykorzystał swój time_slice.

RSDL

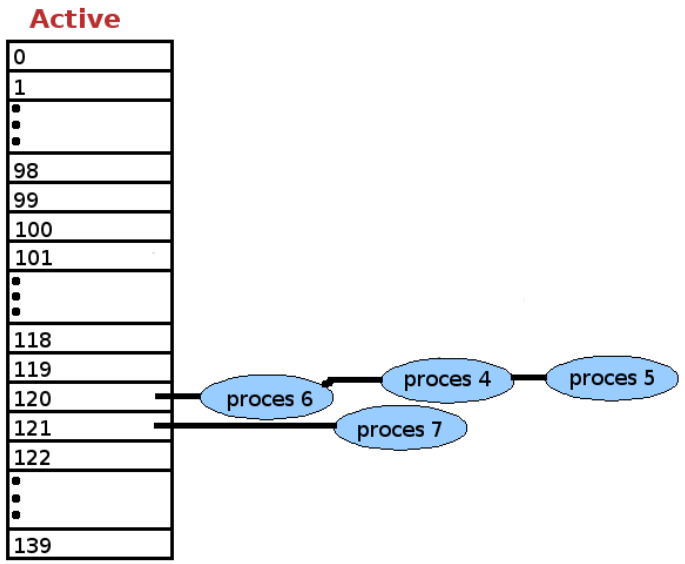
Schemat działania cd.cd.cd.

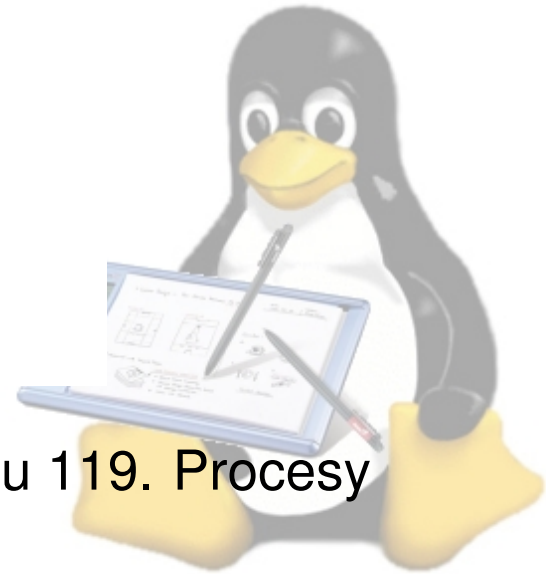


Rysunek: Procesy 4 i 5 wykorzystały quotę przewidzianą dla priorytetu 118 i spadły na priorytet 119 (przydzielona została nowa quota dla priorytetu 118. Taka operacja to 'minor rotation'.

RSDL

Schemat działania cd.cd.cd.

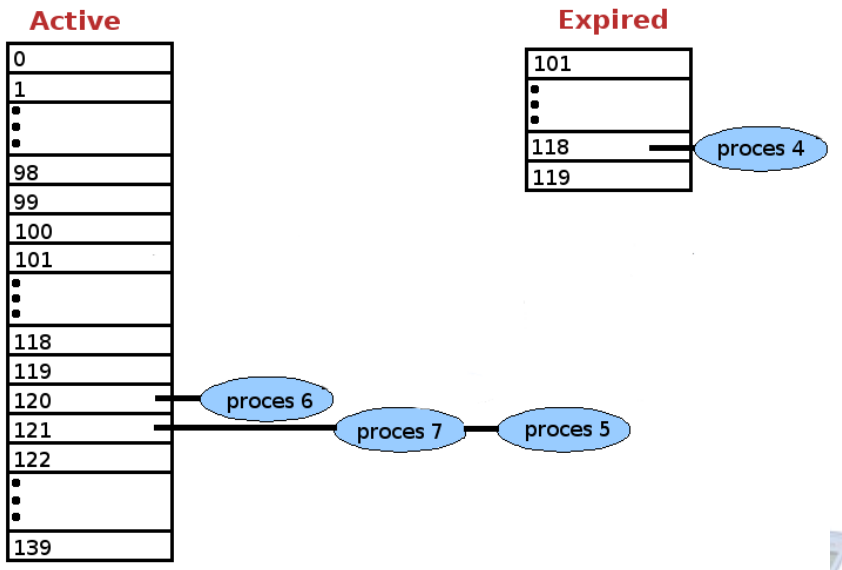


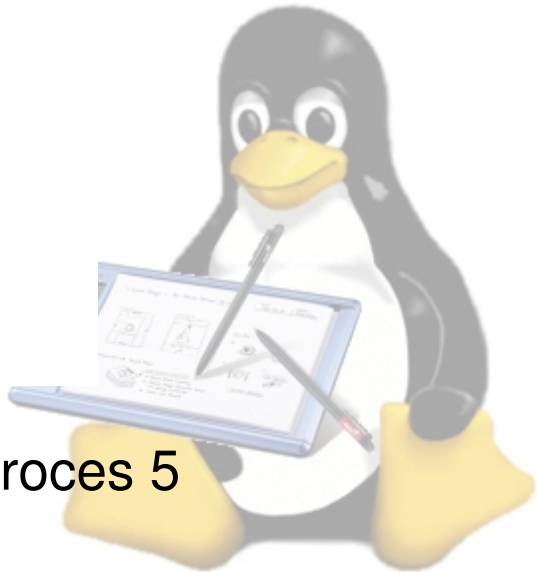


Rysunek: Podobnie, wykorzystano quotę priorytetu 119. Procesy spadając dołączają się do kolejnej listy.

RSDL

Schemat działania cd.cd.cd.

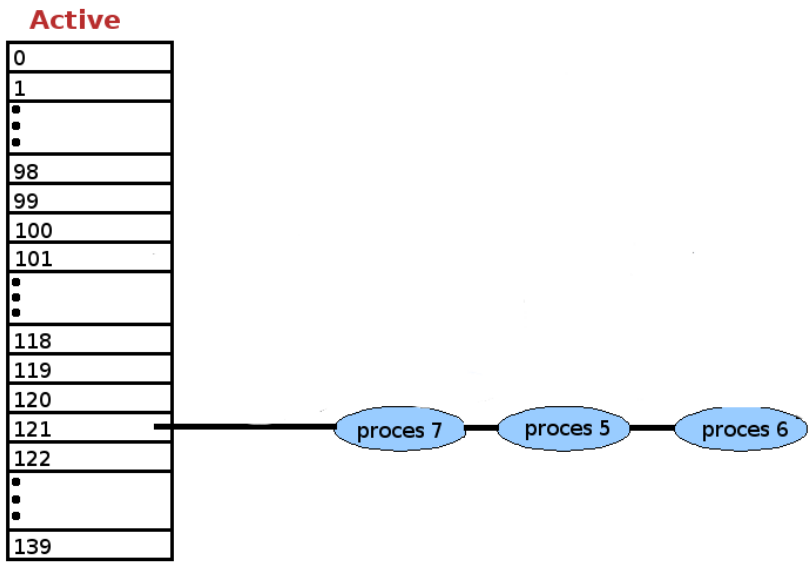


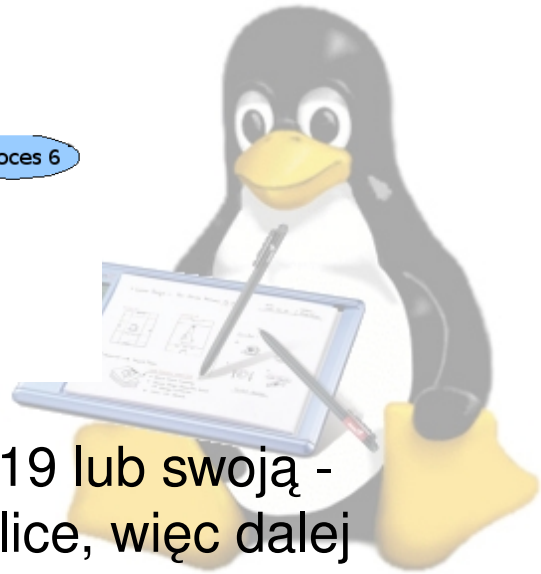


Rysunek: Proces 4 wykorzystał swój time_slice. Proces 5 wykorzystał swoją quotę dla priorytetu 119.

RSDL

Schemat działania cd.cd.cd.

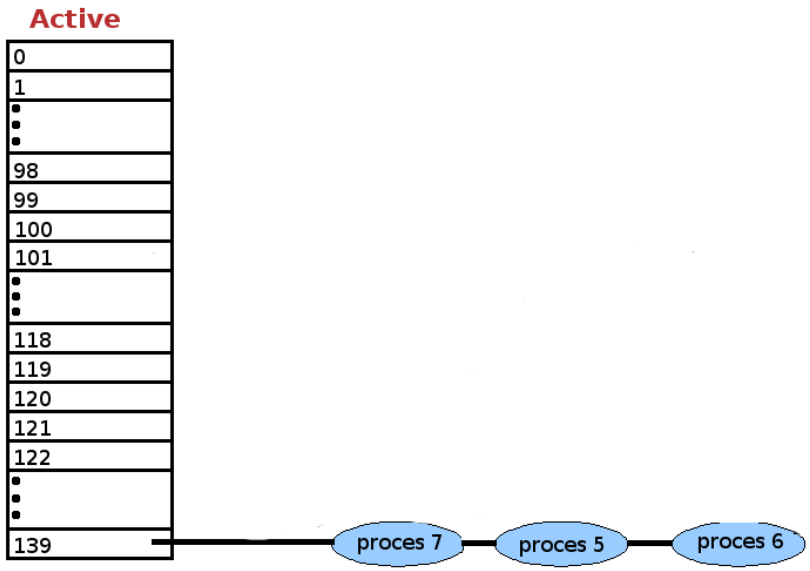


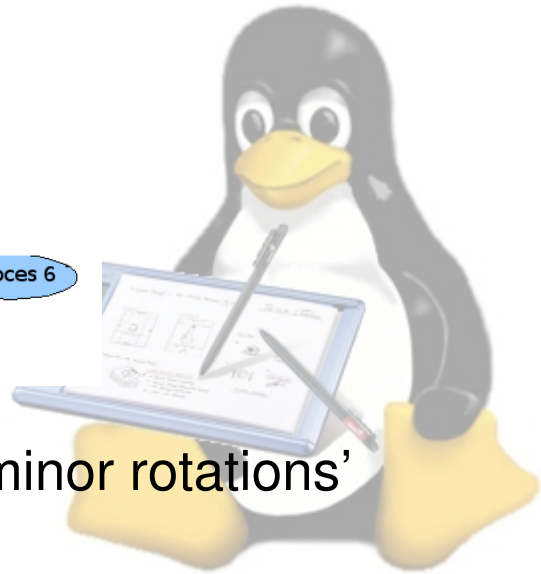


Rysunek: Proces 6 wykorzystał quotę priorytetu 119 lub swoją - przeznaczoną na ten priorytet. Ma jeszcze time_slice, więc dalej znajduje się w kolejce Active.

RSDL

Schemat działania cd.cd.cd.

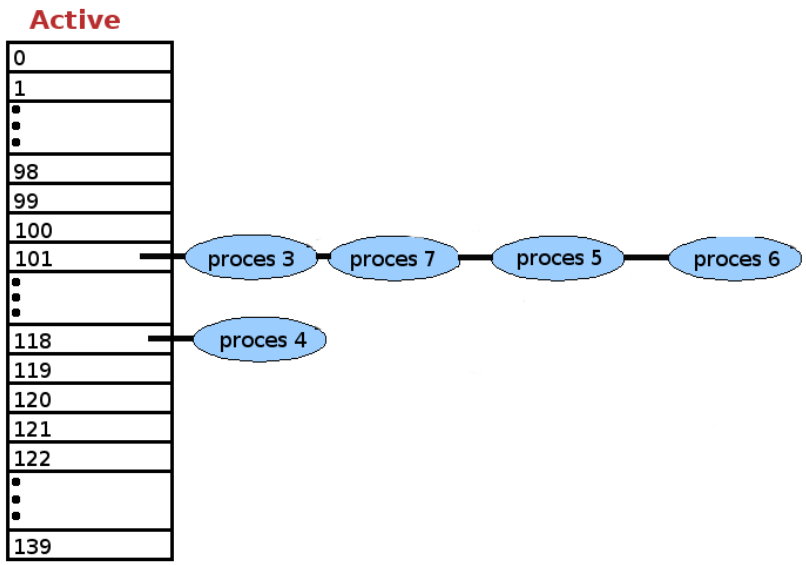


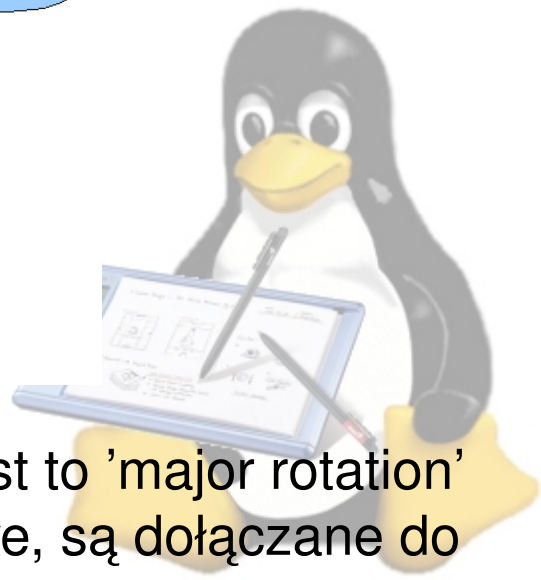


Rysunek: Wszystkie procesy wykonując kolejne ’minor rotations’ spadły na sam dół.

RSDL

Schemat działania cd.cd.cd.





Rysunek: Zamieniamy kolejki Active i Expired. Jest to 'major rotation' Procesy, które znajdowały się na dole kolejki Active, są dołączane do najwyższej (niebędącej RT) niepustej listy procesów aktywnych.

RSDL

runqueue

struct rq

struct prio_array *active, *expired;
/* Tablice procesów aktywnych i tych, które wykorzystały swoje kwanty czasu */
long prio_quota[PRIO_RANGE];
/* Ilość przerwań zegarowych, jaka musi minąć
na każdym priorytecie dynamicznym,
zanim dojdzie do ’minor rotation’ */

RSDL

runqueue cd.

struct rq

unsigned long prio_queued[MAX_PRIO];
/* Ilość zadań na każdym priorytecie */
unsigned long static_bitmap[6];
/* Bitmapa wszystkich priorytetów statycznych */
unsigned long dyn_bitmap[6];
/* Bitmapa wszystkich priorytetów dynamicznych */
int prio_level;
/* Priorytet aktualnie wykonywanych procesów */
unsigned long prio_rotation;
/* Ilość wykonanych 'major rotations' */

RSDL

task_struct

struct task_struct

int prio, static_prio;
/* Priorytet aktualny oraz statyczny */
unsigned long bitmap[6];
/* Bitmapa priorytetów, na których proces wykonywał się w aktualnej epoce */
unsigned long rotation;
/* Epoka, w której ostatnio wykonywał się proces */
unsigned int time_slice;
/* Czas do wykonywania przeznaczony w aktualnej epoce */
unsigned int quota;
/* Czas do wykonywania się na danym priorytecie. Zależy od stałej RR_INTERVAL i poziomu nice */

RSDL

Cechy RSDLa - według Cona Kolivasa

Całkowita sprawiedliwość

Nieprawda, ponieważ procesy, które śpią, tracą przez to czas procesora. Przesypiają kolejne epoki, dlatego ich time_slice się nie zwiększa.

RSDL

Cechy RSDLa - według Cona Kolivasa

Brak zagłodzeń

Prawda, ponieważ procesy w ramach każdego priorytetu działają według polityki Round Robin, a czas przeznaczony dla każdego priorytetu jest ograniczony w każdej epoce.

RSDL

Cechy RSDLa - według Cona Kolivasa

Ograniczone czasy oczekiwania na wykonanie

Prawda, poniważ procesy w tablicy aktywnych skończą się po z góry określonym czasie. Czasy te mogą być jednak dosyć duże.

RSDL

Cechy RSDLa - według Cona Kolivasa

Czas działania O(1), wysoka skalowalność

Prawda, podobnie jak w schedulerze z jądra 2.6.

RSDL

Cechy RSDLa - według Cona Kolivasa

Interaktywność

Nieprawda. Według autora procesy, które będą się budzić, będą zazwyczaj na wyższym priorytecie, niż pozostałe aktywne. Nie jest to jednak taka większość przypadków, jakiej Con by się spodziewał.

RSDL

Cechy RSDLa - według Cona Kolivasa

Prosta budowa

Sprawa dyskusyjna ;)

RSDL

Cechy RSDLa - według Cona Kolivasa

Brak heurystyk dotyczących procesów interakcyjnych

Prawda, oczywiście.

RSDL

Cechy RSDLa - według Cona Kolivasa

Brak pomiarów czasu działania, tylko proste obliczenia

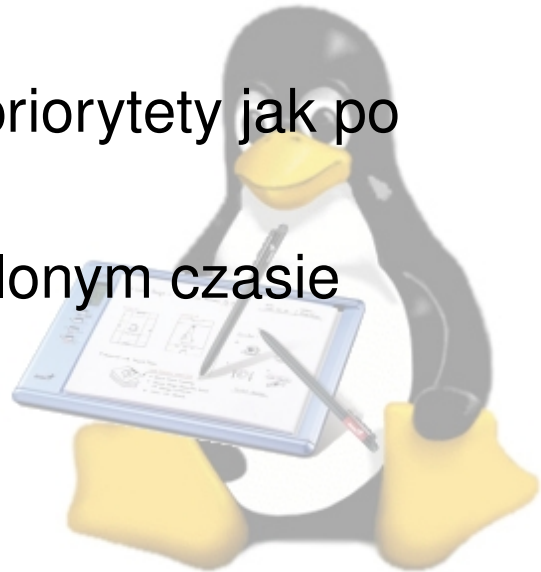
Prawda, co wszyscy widzieliśmy.

RSDL

Odszyfrowanie nazwy

Teraz już wiemy, skąd wzięła się nazwa schedulera:

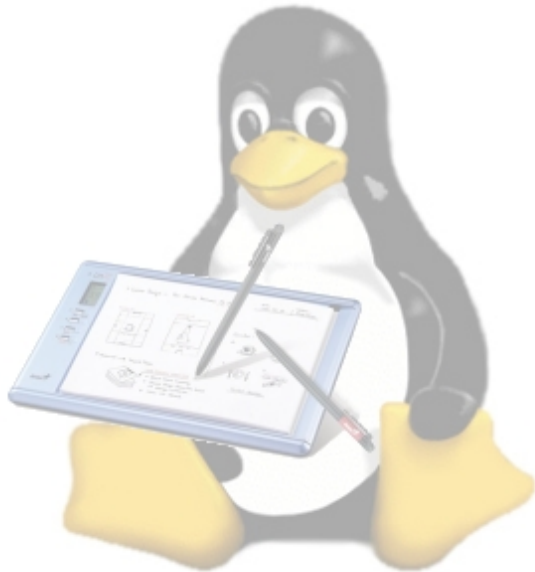
- **Rotating** - ciągle wykonywane są ’minor -’ i ’major rotations’;
- **Staircase** - procesy spadają na kolejne priorytety jak po schodach;
- **DeadLine** - każdy proces w z góry określonym czasie dostanie się do procesora;



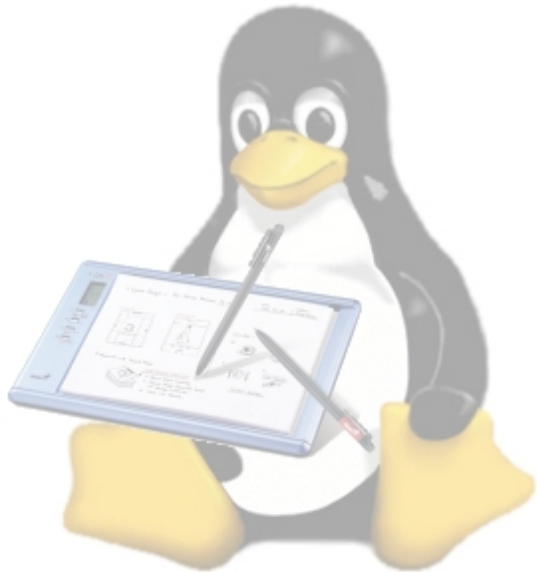
RSDL

Dlaczego go nie wybrano?

- CFS stworzony przez Ingo Molnara;
- wypadek;
- kłótnia o sched_yield();
- a może Linus Torvalds...



Completely Fair Scheduler



CFS

Idea powstania

O czym myślał Ingo, zanim napisał CFSa?

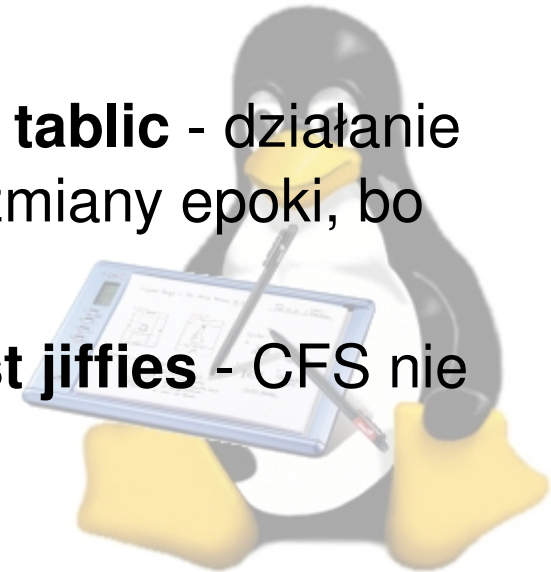
- **idealny procesor wielozadaniowy** - taki procesor, na którym N procesów wykonuje się jednocześnie, każdy z nich otrzymuje $\frac{1}{N}$ mocy procesora;
- **O(1) '<' O(140)** - 140 w schedulerze O(1) jest tylko stałą w kodzie, którą można zmieniać;
- $140 = log_2(2^{140})$**, a** 2^{140} **to bardzo dużo** - może więc lepiej napisać algorytm, działający w czasie logarytmicznym ze względu na ilość procesów.

CFS

Cechy CFSa - według Ingo Molnara...

.. czyli w czym CFS jest lepszy od O(1) i RSDLa:

- **CAŁKOWITA sprawiedliwość** - procesy interaktywne nie są faworyzowane, ani krzywdzone - każdy ma takie same prawa;
- **brak artefaktów podczas przełączania tablic** - działanie schedulera nie zatrzymuje się w czasie zmiany epoki, bo jej w CFSie nie ma;
- **nanosekundowa granularność zamiast jiffies** - CFS nie poleg na wartościach jiffies oraz HZ;

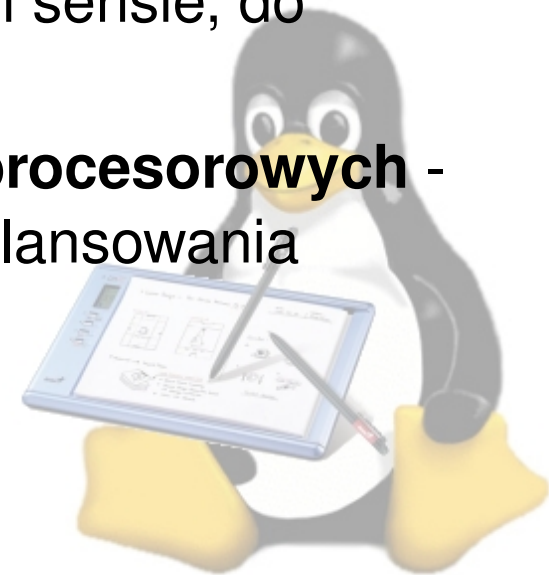


CFS

Cechy CFSa - według Ingo Molnara...

.. czyli w czym CFS jest lepszy od O(1) i RSDLa, cd.:

- **brak kwantów czasu** - nie wyliczamy każdemu procesowi czasu, jaki może się wykonywać, w takim sensie, do jakiego się przyzwyczailiśmy;
- **lepsze działanie na maszynach wieloprocesorowych** - od nowa napisane funkcje służące do balansowania procesów na różnych procesorach;
- **modularna budowa** - OOPS!



A propos modularnej budowy...

PlugSched

- idea opracowana przez Cona Kolivasa
- scheduler jako moduł, który można w każdej chwili wymienić (nawet w trakcie działania systemu)
- pomysł całkowicie skrytykowany przez Linusa Torvaldsa i innych developerów jądra
- aktualnie jest dostępne jako nakładka na jądro

Scheduler główny oraz prosta lista modułów-schedulerów uszeregowanych według priorytetów.

Scheduler główny odpytuje kolejne moduły, czy mają zadanie do wykonania, aż do momentu, gdy któryś z nich odpowie twierdząco.

Taki moduł musi implementować następujące metody:

- 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);

Metody modułu schedulera cd.

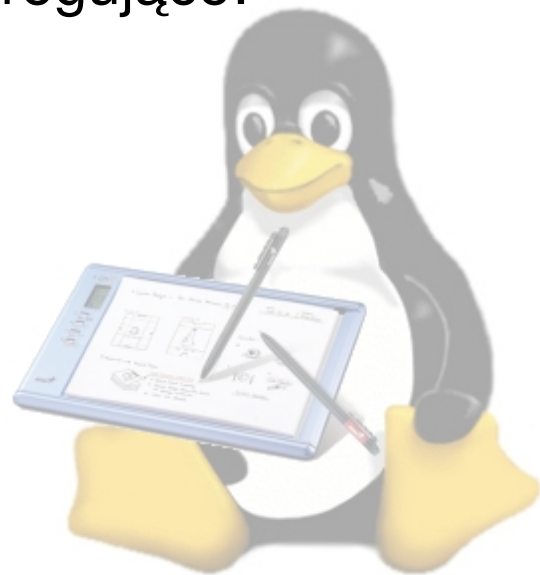
void (*task_tick) (struct rq *rq, struct task_struct *p);
/* Metoda wywoływana podczas przerwania zegarowego w celu przeliczenia czasów i być może uruchomienia następnego procesu */
struct task_struct * (*pick_next_task) (struct rq *rq);
/* Prośba do schedulera o wybranie procesu do uruchomienia */
struct task_struct * (*load_balance_start) (struct rq *rq);
struct task_struct * (*load_balance_next) (struct rq *rq);
/* Metody równoważące obciążenie procesorów */

CFS

Modularność w praktyce

W jądrze 2.6.23 dostępne są dwa moduły szeregujące:

- sched_rt (plik sched_rt.c)
- sched_fair (plik sched_fair.c)

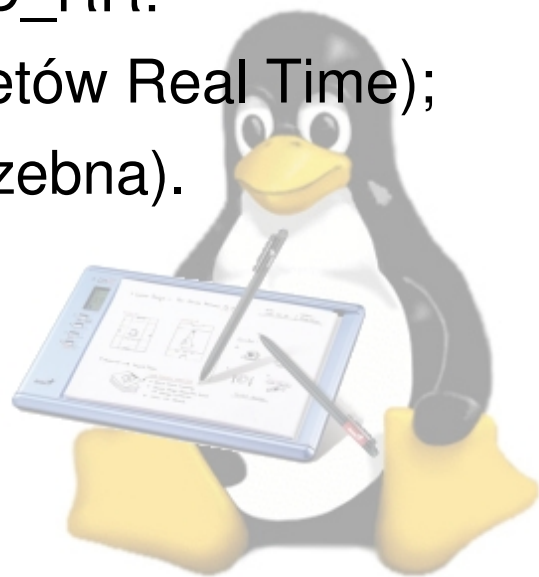


CFS

sched_rt

Implementacja polityk SCHED_FIFO i SCHED_RR:

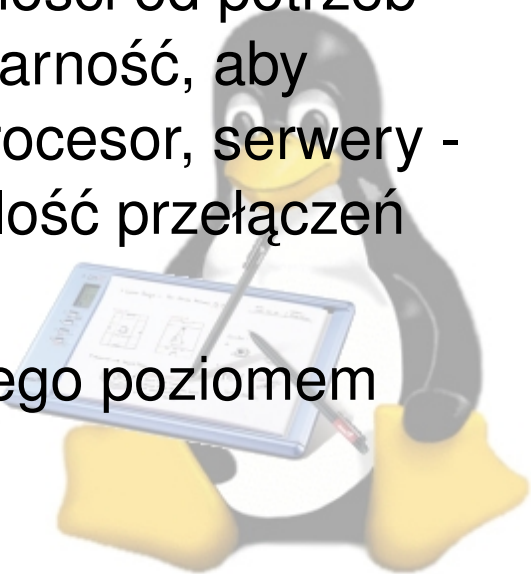
- 100 list procesów (dla wszystkich priorytetów Real Time);
- jedna tablica (tablica Expired jest niepotrzebna).



CFS

sched_fair
Odpowiednik polityk SCHED_OTHER i SCHED_BATCH:

- brak list procesów do wykonania;
- możliwość zmiany granularności w zależności od potrzeb (komputery osobiste - lepsza duża granularność, aby procesy interakcyjne szybko dostawały procesor, serwery - mniejsza granularność, żeby ograniczyć ilość przełączeń kontekstu);
- silny związek czasu działania procesu z jego poziomem nice.



CFS

Idea działania

s64 p->wait_runtime

czas, przez jaki dany proces powinien się wykonywać, aby być kompletnie uczciwym i zbalansowanym.

Na 'idealnym procesorze wielozadaniowym' wartość ta wynosiłaby: 0

CFS

Idea działania

s64 rq->fair_clock

czas procesora, jaki dostałby abstrakcyjny proces wykonujący się od startu systemu, gdyby wszystko odbywało się uczciwie.

Wartość ta jest zwiększana przy każdym przerwaniu zegarowym.

CFS

Idea działania

rq->fair_clock - p->wait_runtime

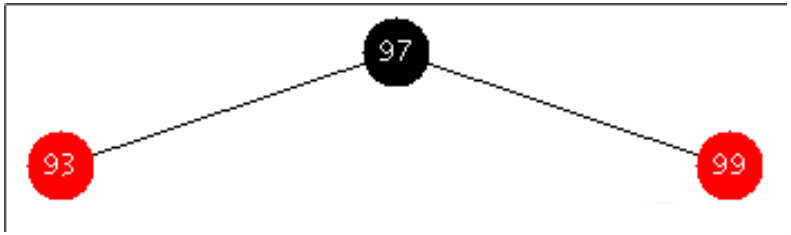
czas procesora, jaki byłby wykorzystany przez ten proces, gdyby działał on od startu systemu, a wszystko odbywałoby się uczciwie.

Ważna wartość! - używana jako klucz w drzewie czerwono-czarnym procesów.

CFS

Schemat działania

proces	wait_runtime	fair_clock – wait_runtime
P1	7	93
P2	3	97
P3	1	99
P4	(śpi) 10	90
P5	–	–
fair_clock: 100		

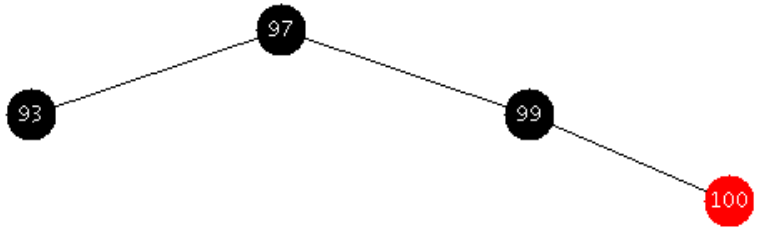


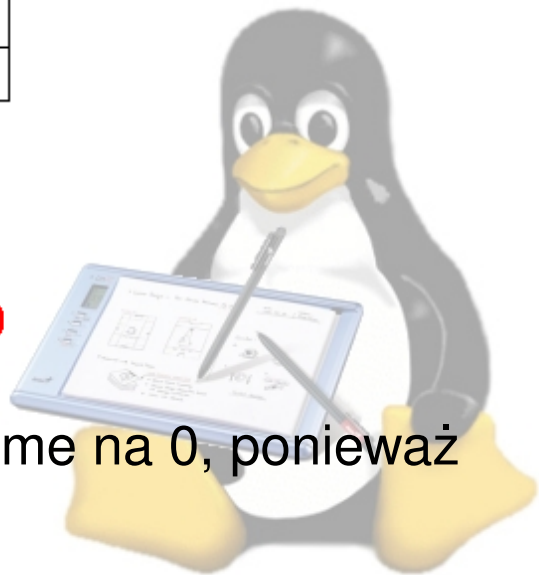
Rysunek: Stan początkowy: procesy aktywne umieszczone w drzewie czerwono-czarnym.

CFS

Schemat działania

proces	wait_runtime	fair_clock – wait_runtime
P1	7	93
P2	3	97
P3	1	99
P4	(śpi) 10	90
P5	0	100
fair_clock: 100		



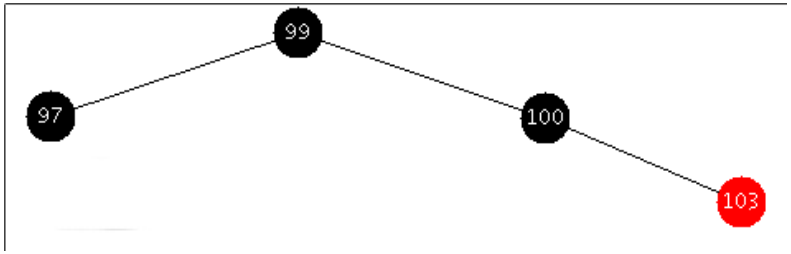


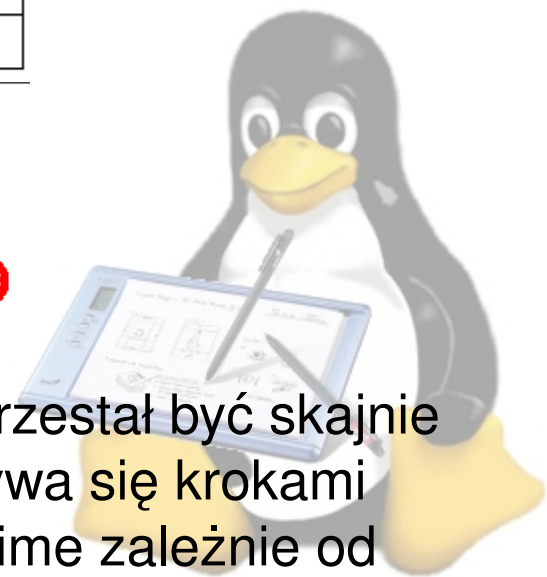
Rysunek: Pojawił się Proces 5. Ustawia wait_runtime na 0, ponieważ nie ma zaległości i dodaje się do drzewa.

CFS

Schemat działania

proces	wait_runtime	fair_clock – wait_runtime
P1	2	103
P2	3	97
P3	1	99
P4	(śpi) 10	90
P5	0	100
fair_clock: 105		



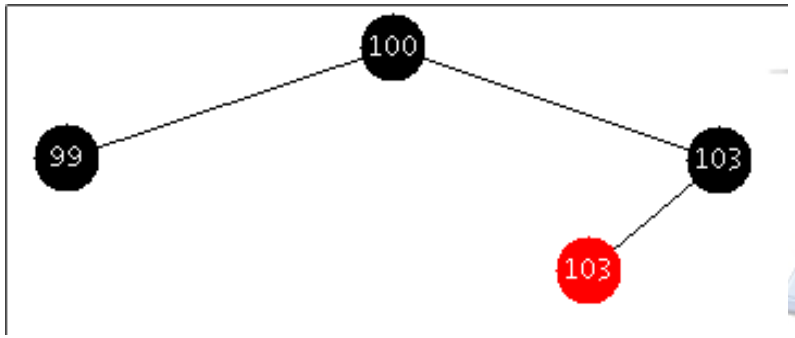


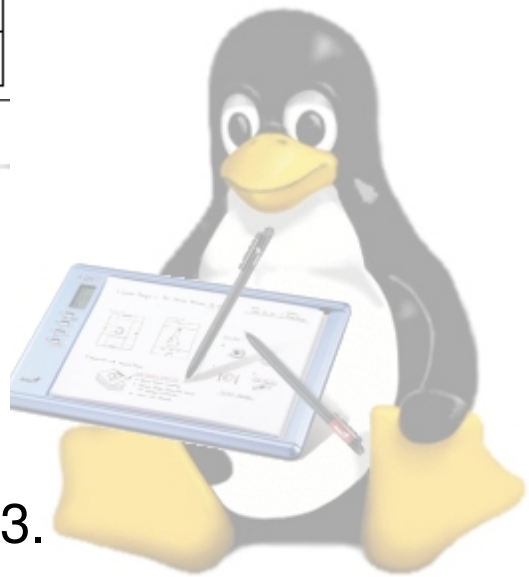
Rysunek: Proces 1 wykonywał się tak długo, aż przestał być skajnie lewym w drzewie. Wykonywanie się procesu odbywa się krokami (granularność), procesowi zmniejszamy wait_runtime zależnie od jego prio.

CFS

Schemat działania

proces	wait_runtime	fair_clock – wait_runtime
P1	2	103
P2	5	103
P3	1	99
P4	(śpi) 10	90
P5	0	100
fair_clock: 108		



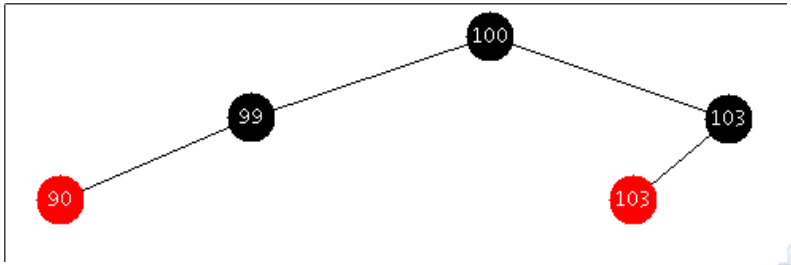


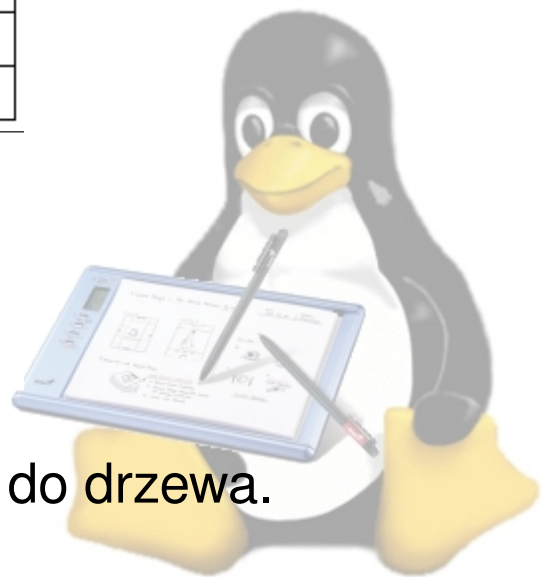
Rysunek: Wykonywał się Proces 3.

CFS

Schemat działania

proces	wait_runtime	fair_clock – wait_runtime
P1	2	103
P2	5	103
P3	1	99
P4	18	90
P5	0	100
fair_clock: 108		



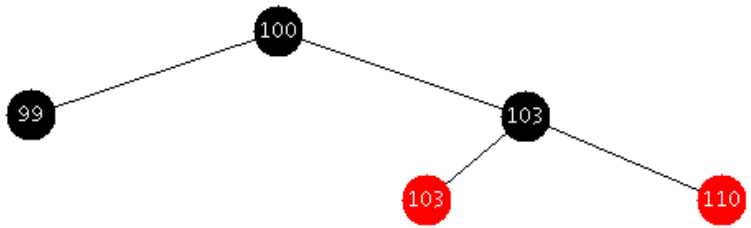


Rysunek: Proces 4 się obudził i dodał się do drzewa.

CFS

Schemat działania

proces	wait_runtime	fair_clock – wait_runtime
P1	2	103
P2	5	103
P3	1	99
P4	8	110
P5	0	100
fair_clock: 118		

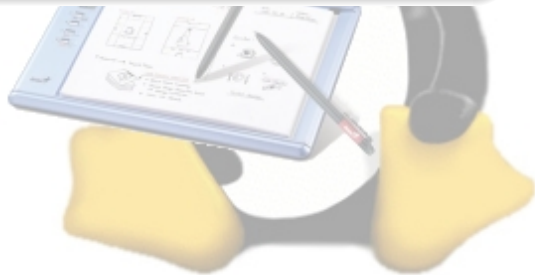


Rysunek: Wykonywał się proces 4 - po tym, jak się obudził, miał bardzo duże braki czasu wykonania.

Trendy rozwojowe

2.4 -> 2.6

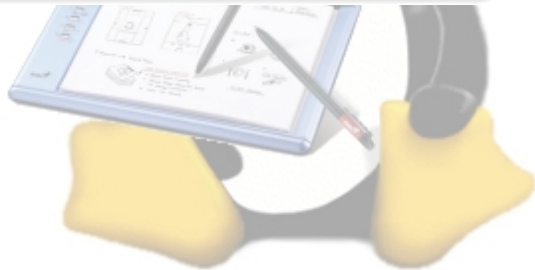
- lepsze wykorzystanie SMP
- lepsze działanie przy obciążonym systemie (dużo procesów)
- poprawienie interakcyjności



Trendy rozwojowe

2.6 -> 2.6.23

- wprowadzenie sprawiedliwości
- proste obliczenia, nie wyliczanie kwantów czasu
- nowe struktury danych
- modularność



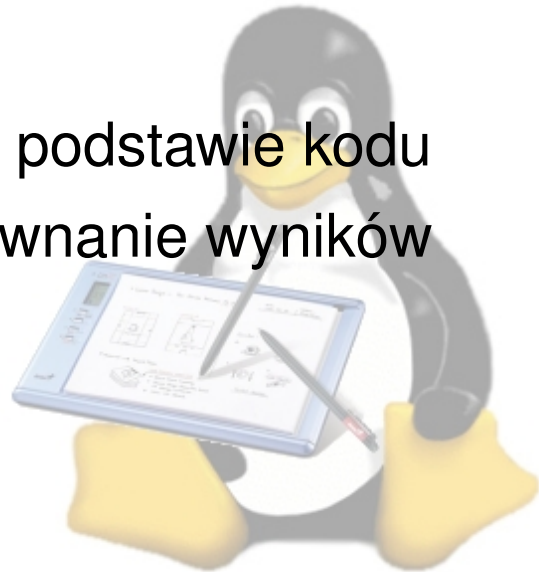
Trendy rozwojowe

2.6.23 -> ?

- coraz lepsze wykorzystywanie SMP
- oddzielenie schedulerów desktopowych od serwerowych
- większa modularność (schedulery jako strumienie)

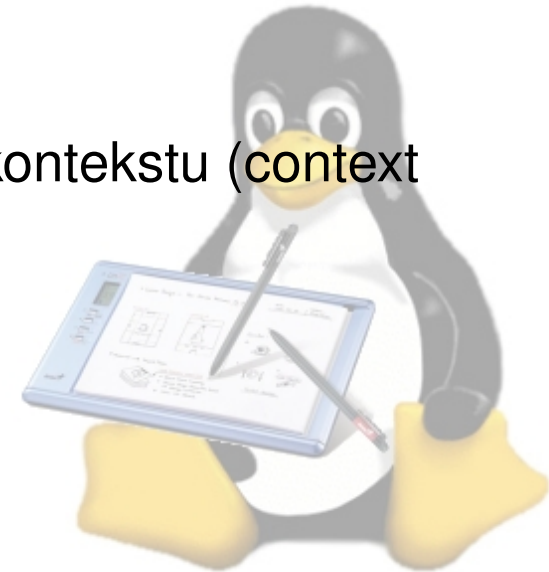
Jak mierzyć wydajność?

- Nie ma jednej wyroczni, która powie: ten scheduler jest najlepszy i będzie miała zawsze rację...
- ...może z wyjątkiem Linusa Torvaldsa
- Nie da się odpowiedzieć na to pytanie na podstawie kodu
- Pozostaje przeprowadzenie testów i porównanie wyników



Rodzaje testów

1. Mierzą konkretne cechy schedulera:
   - Oczekiwanie na wykonanie (latency)
   - Długość przydzielonego czasu (runslice)
   - Wydajność (performance)
   - Narzut (overhead)
   - Skalowalność (scalability)
   - Szybkość i częstotliwość przełączania kontekstu (context switch)
   - Interaktywność (interactivity)
   - Przepustowość (throughput)
   - Reaktywność (responsiveness)

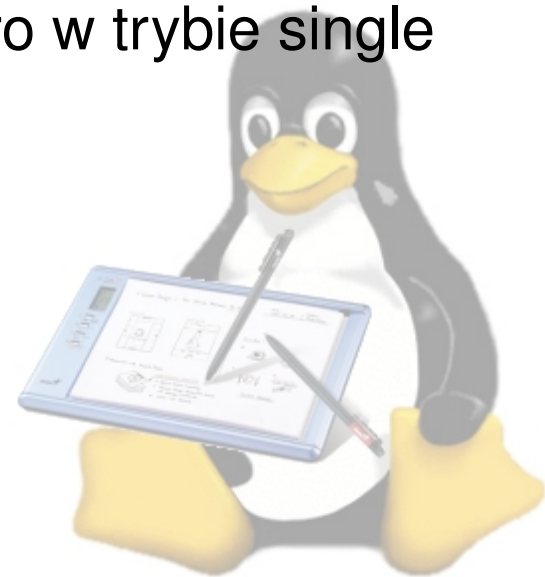


Rodzaje testów

2. Symulują działanie całego systemu w konkretnych warunkach
   - Modeluje się obciążenie systemu (np. używając logów z serwerów)
   - Przyjmuje się określone profile aplikacji
     - wymagania dot. przydziału procesora
     - sposób działania
     - czas reakcji
   - Uruchamia się prawdziwe aplikację i normalnie pracuje

Jak przeprowadzać testy

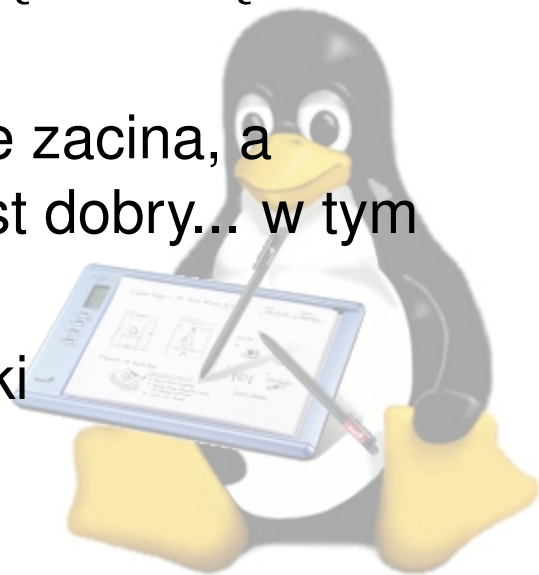
1. Kompilujemy jądro z odpowiednim schedulerem (patche, plugsched)
2. Ściągamy odpowiedni test (lub piszemy własny:)
3. Czytamy README
4. Najczęściej uruchamiamy testowane jądro w trybie single user mode
5. Uruchamiamy test (np. 100 razy)
6. Czekamy...
7. Jemy obiad...
8. Czekamy...
9. Odczytujemy wyniki



Test „zrób to sam”

Czy musimy symulować, mierzyć, używać specjalnych testów, żeby przetestować scheduler? Nie :) Prosty przepis na test:

1. Uruchom kompresję muzyki (oczywiście legalnej)
2. Uruchom Quake’a (lub inna oryginalną grę w którą lubisz grać)
3. Jeśli się zacina goto end. Jeśli gra się nie zacina, a muzyka kompresuje tzn. że scheduler jest dobry... w tym przypadku ;)
4. Uruchom kolejny proces kompresji muzyki
5. goto 3

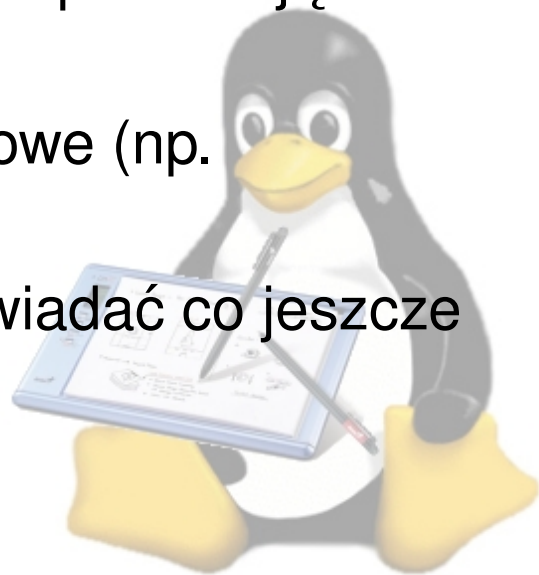


Testy - minusy

- Sprawdzają tylko ograniczoną liczbę przypadków użycia
- Mówią, że coś działa w danym przypadku, a nie zawsze
- Są wrażliwe na działanie: innych procesów i sprzętu
- Zwykle trzeba powtarzać je wiele razy

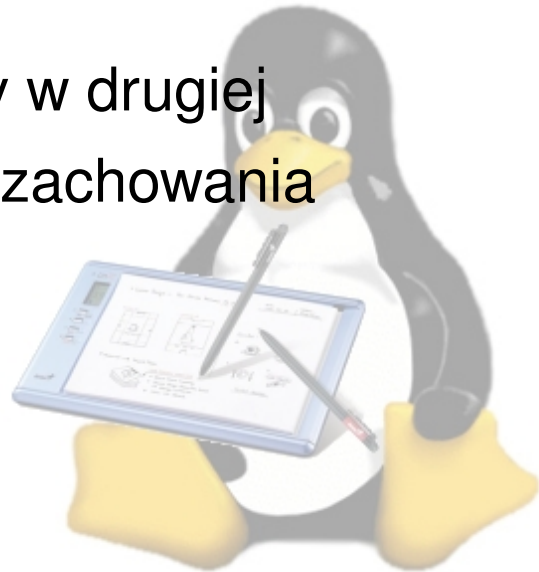
Testy - plusy

- Są ;)
- Dostarczają pewnych informacji o schedulerze
- Odpowiednio (wiele razy) przeprowadzone pozwalają na porównanie działania schedulerów
- Wykrywają pewne błędy i sytuacje brzegowe (np. massive_intr)
- Wszystkie otrzymane dane mogą podpowiadać co jeszcze zoptymalizować



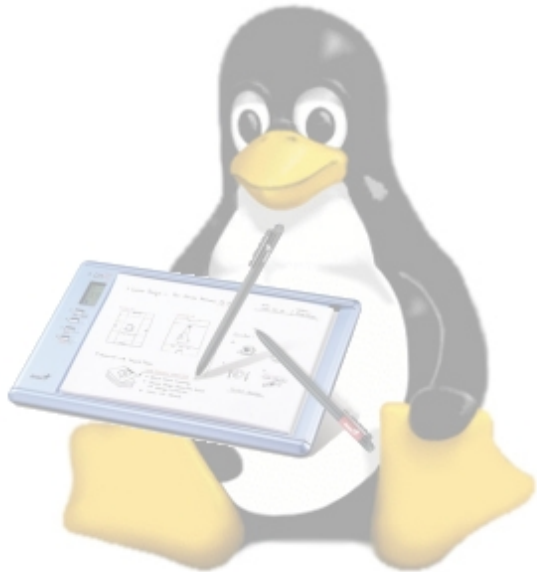
Testy - problem z wynikami

- Niektóre wielkości nie są lepsze lub gorsze (częstotliwość przełączania kontekstu)
- Lepszy wynik w jednej kategorii, a gorszy w drugiej
- Różne zastosowania wymagają różnego zachowania



Programy testujące

- Pipe_test
- Lat_ctx
- Hackbench
- Interbench
- Kernbench
- dbench
- contest
- starve
- massive_intr
- sysbench
- ocbench



Nasze testy

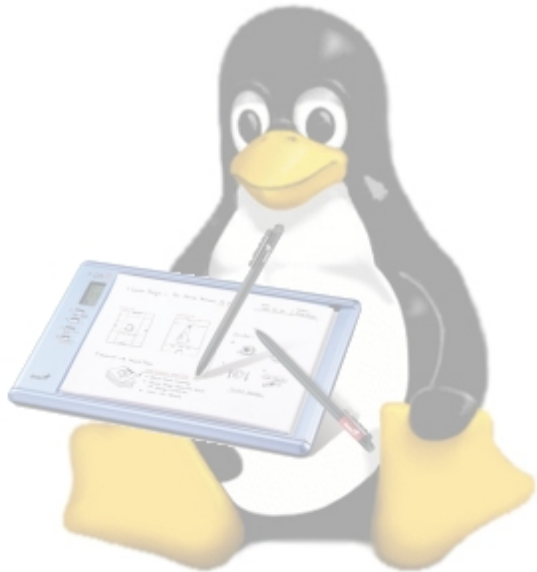
Przeprowadziliśmy własne testy, aby porównać z wynikami dostępnymi w sieci. Na jądrze 2.6.21.7 w trybie single-user przetestowaliśmy schedulery:

- O(1)
- CK2 (lub SD) - ostatnia nakładka na O(1) często mylona z RSDL
- RSDL (v0.33)
- CFS (v24)

Pipe_test

Tworzy 2 procesy, które N razy przekazują sobie żeton
Mierzy:

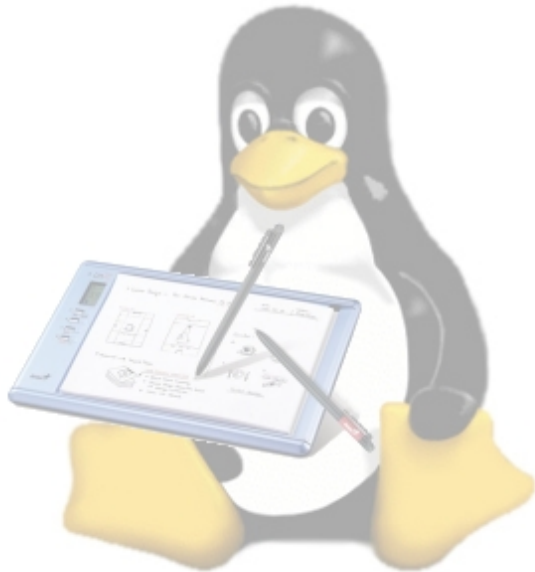
- szybkość przełączania kontekstu (context switch)



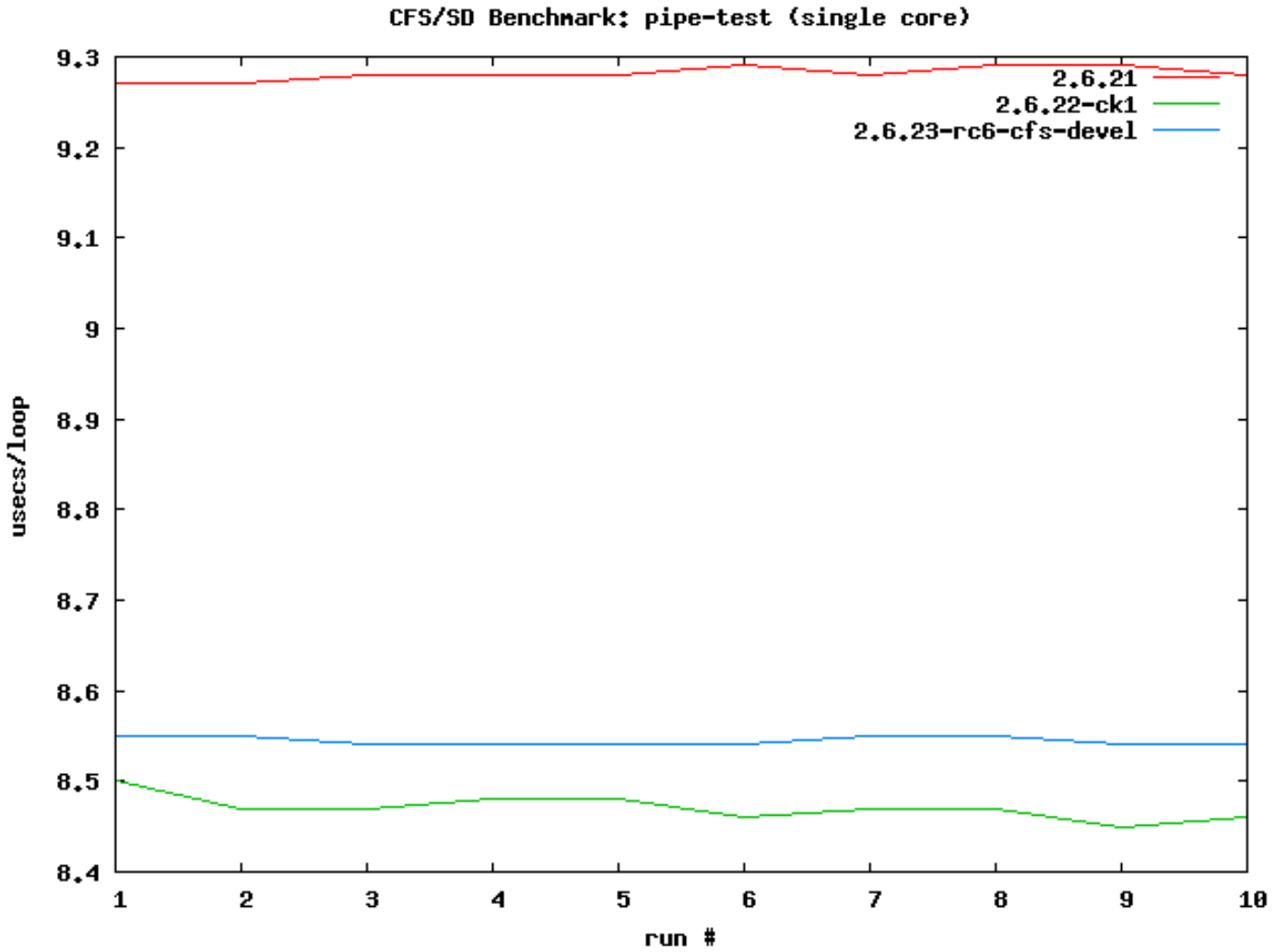
Pipe_test

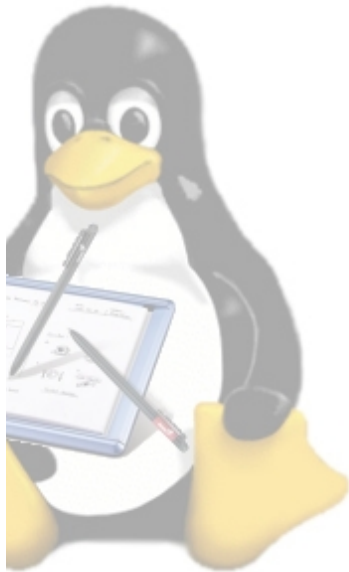
Tworzy 2 procesy, które N razy przekazują sobie żeton
Mierzy:

- szybkość przełączania kontekstu (context switch)

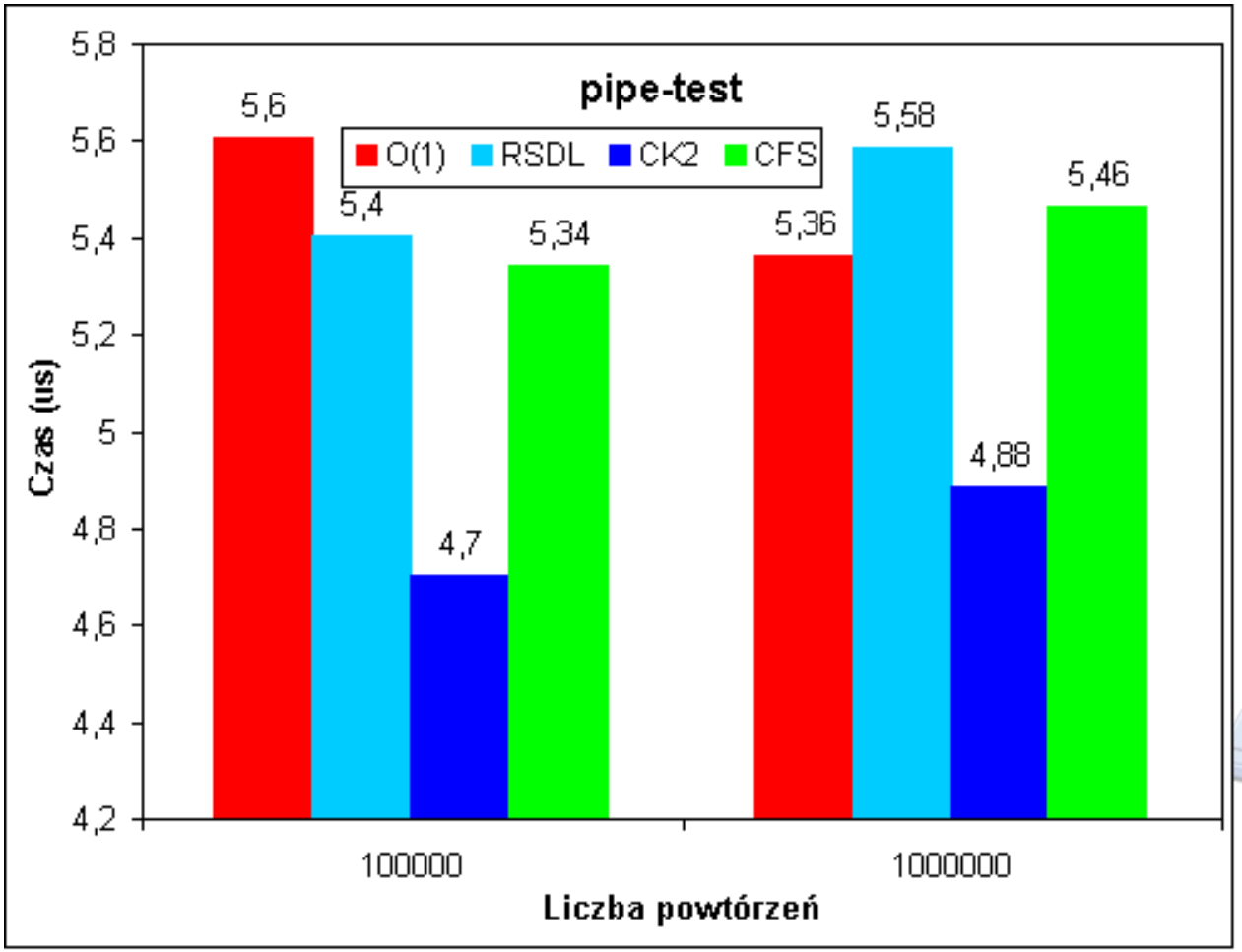


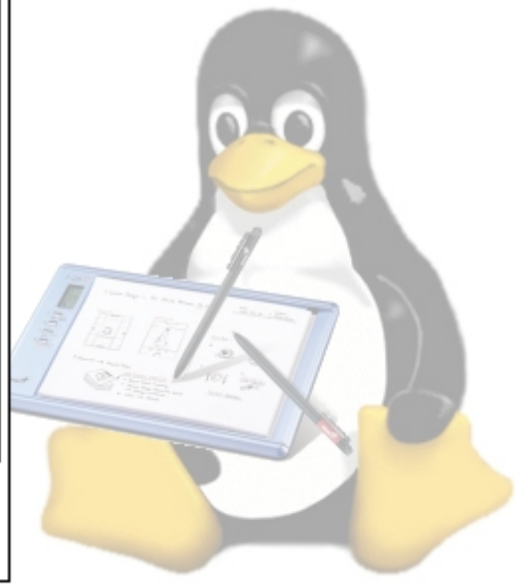
Pipe_test - wyniki





Pipe_test - nasze wyniki





Pipe_test - krótki komentarz

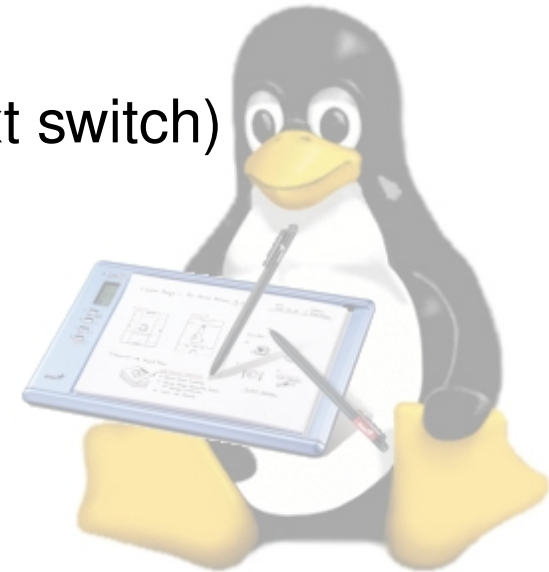
1. Niewielką przewagę nad CFS uzyskał CK1 (poprzednia wersja CK2), prawie 10% traci O(1)
2. W naszych testach najlepiej spisywał się ze sporą (ok. 15%) przewagą CK2. RSDL, CFS i O(1) prezentowały podobny poziom.

Najnowsze schedulery nie okazały się lepsze od poprzednich.

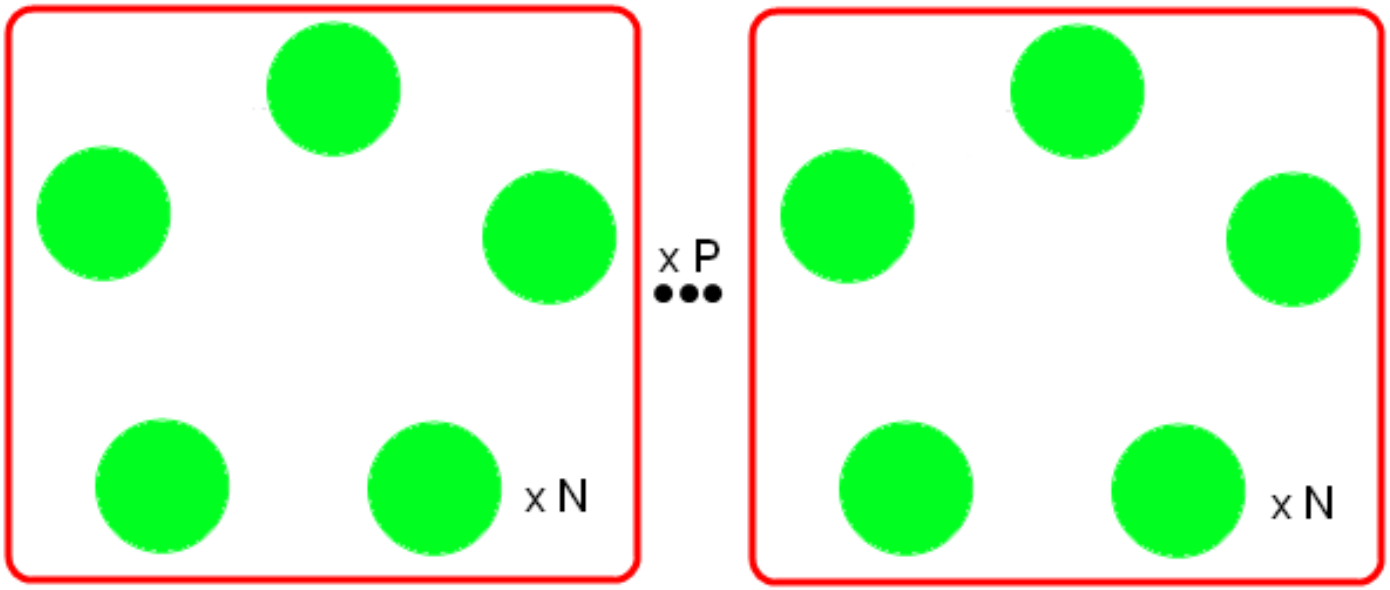
Lat_ctx

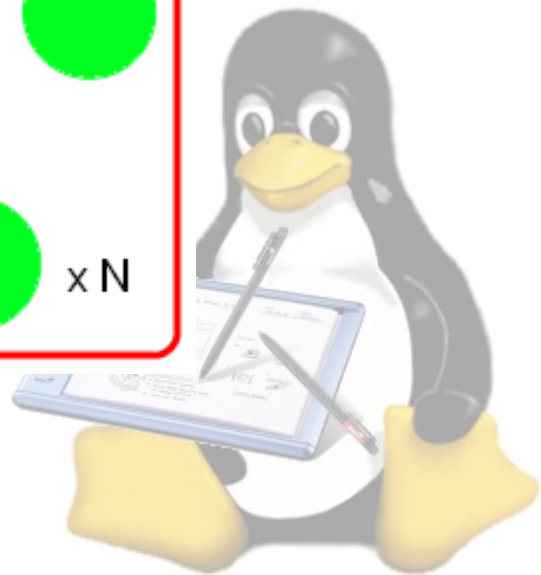
Tworzy P pierścieni z N procesów, które przekazują sobie żeton i ew. wykonują jakąś pracę
Mierzy:

- szybkość przełączania kontekstu (context switch)
- skalowalność (scalability)

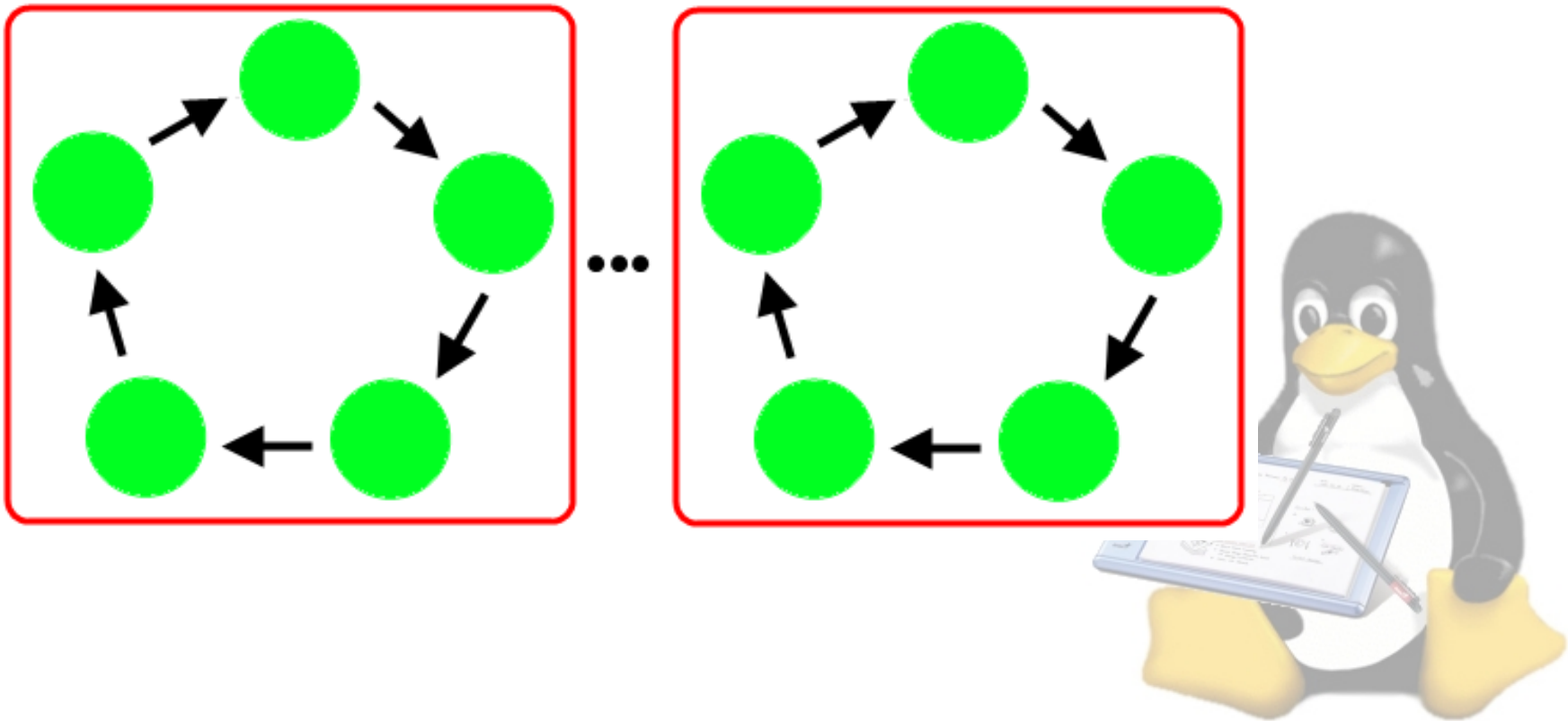


Lat_ctx

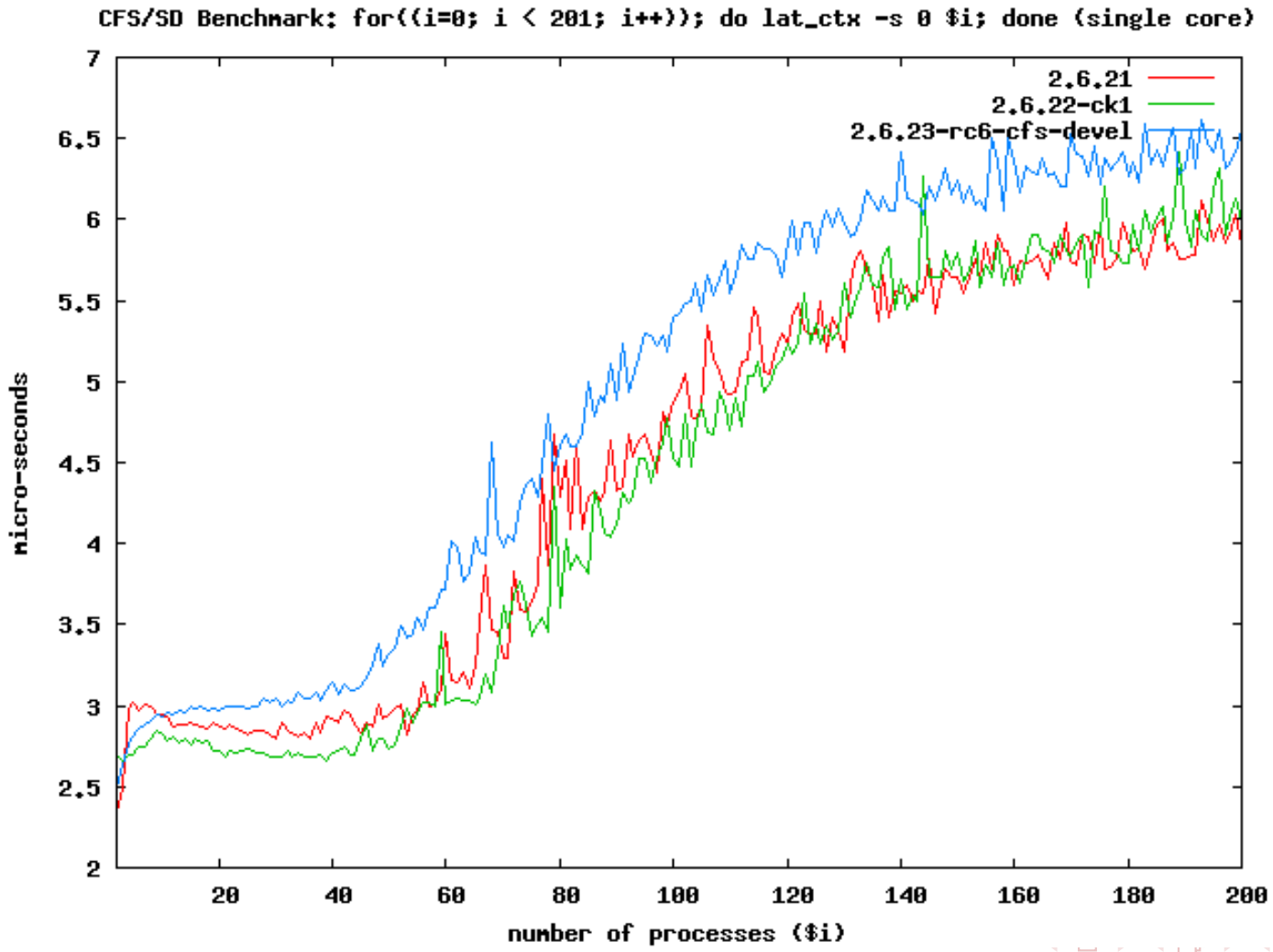


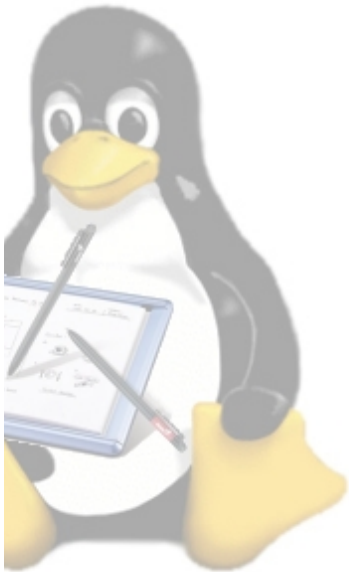


Lat_ctx

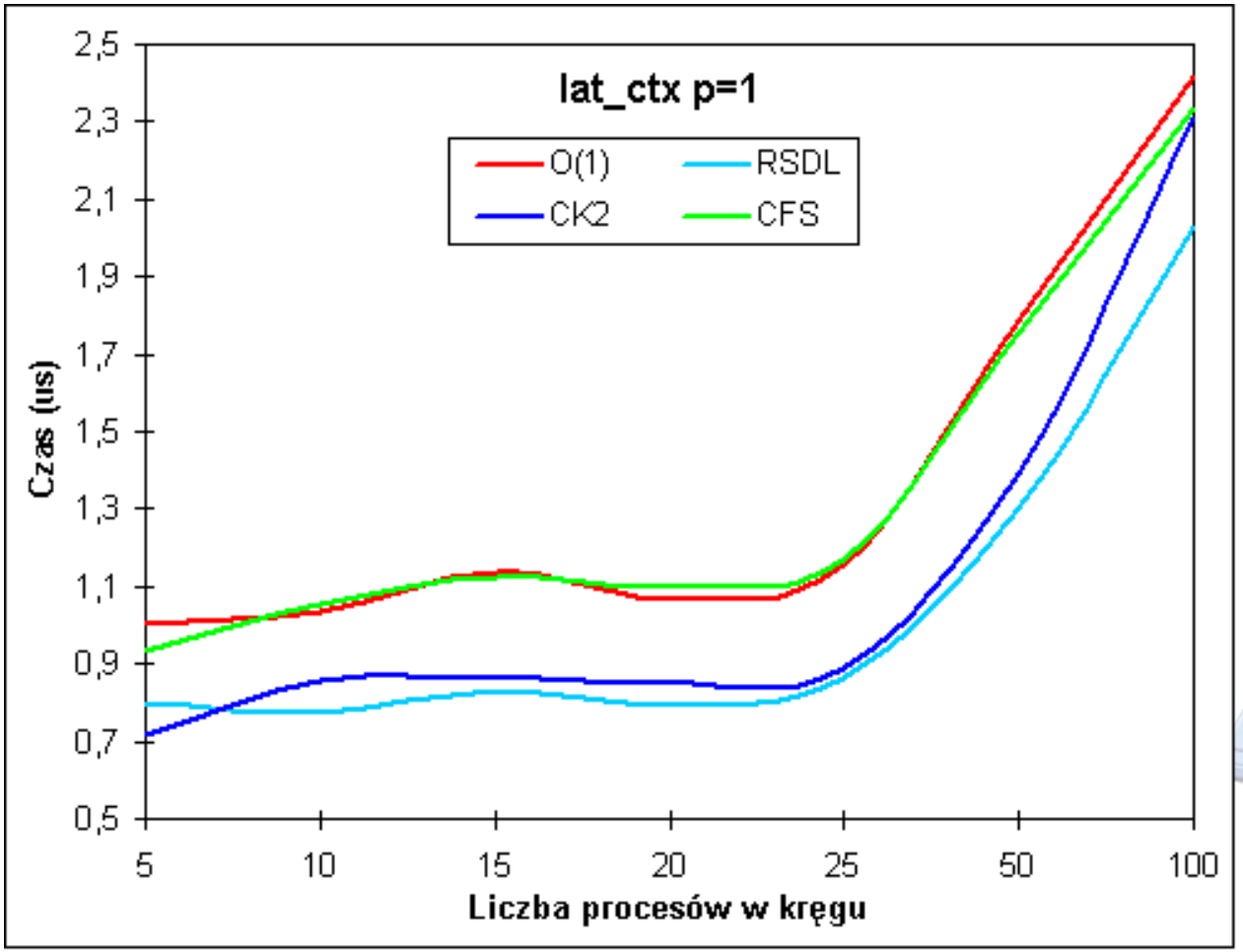


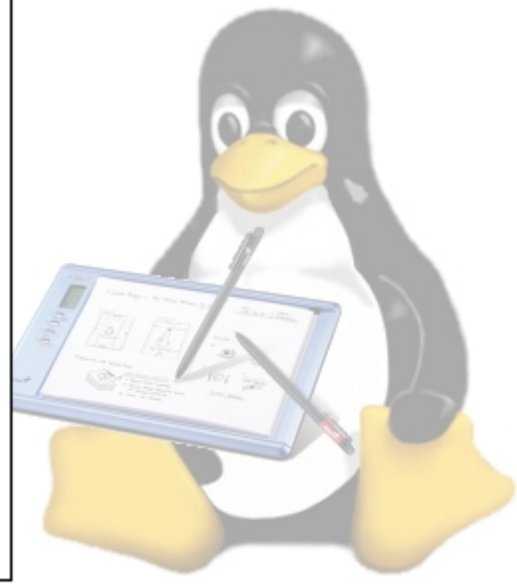
Lat_ctx - wyniki



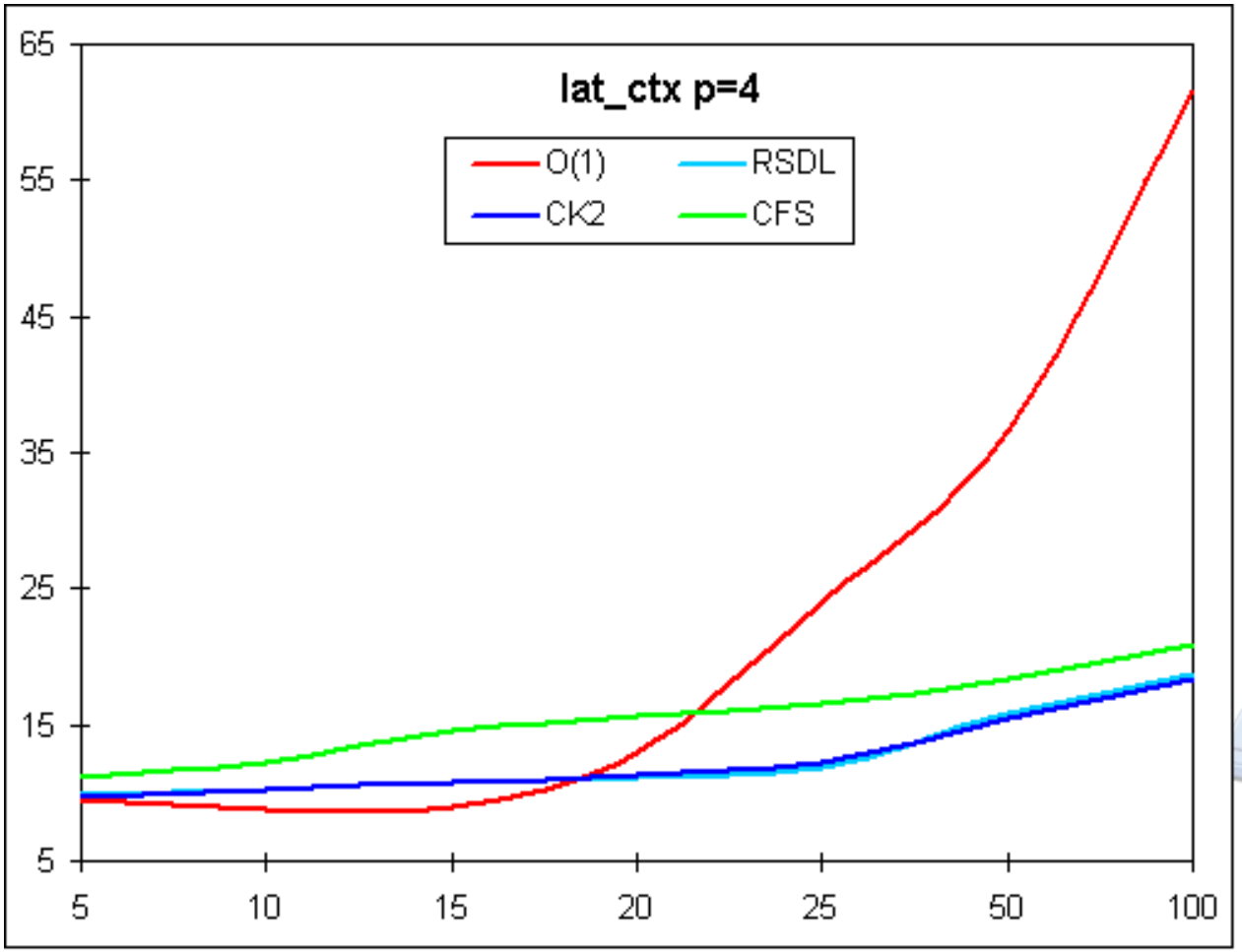


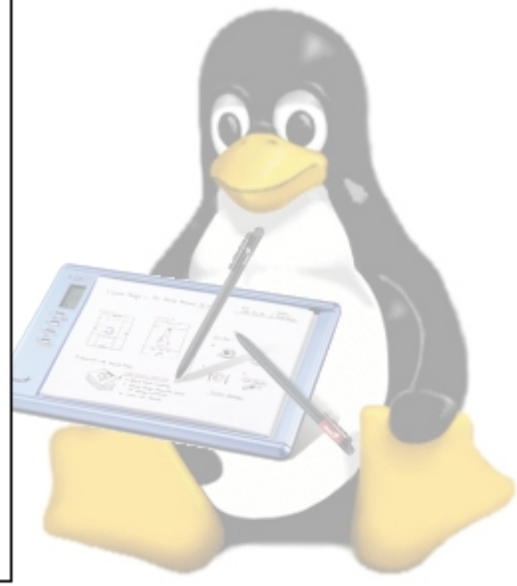
Lat_ctx - nasze wyniki





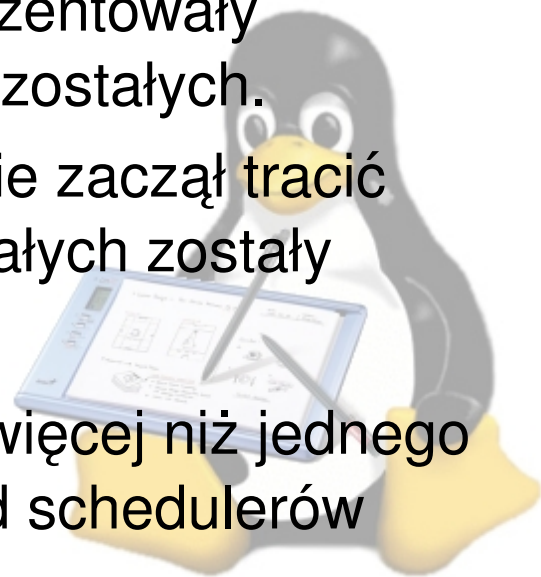
Lat_ctx - nasze wyniki





Lat_ctx - krótki komentarz

1. Dla jednego pierścienia O(1) i CK1 prezentują praktycznie identyczny poziom. CFS traci ok. 10%
2. W naszych testach najlepiej spisywał się RSDL z minimalną przewagą nad CK2, większą uzyskał dopiero przy ponad 50 procesach.CFS i O(1) prezentowały podobny poziom gorszy o ok. 20% od pozostałych.
3. Przy czterech pierścieniach zdecydowanie zaczął tracić O(1) (4 razy od reszty), proporcje pozostałych zostały podobne do poprzedniego testu.

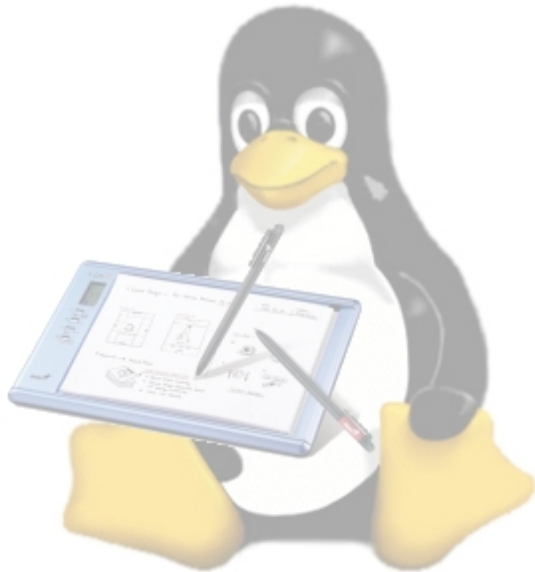


O(1) okazał się bardzo wrażliwy na testy dla więcej niż jednego pierścienia. CFS znowu okazał się słabszy od schedulerów Cona.

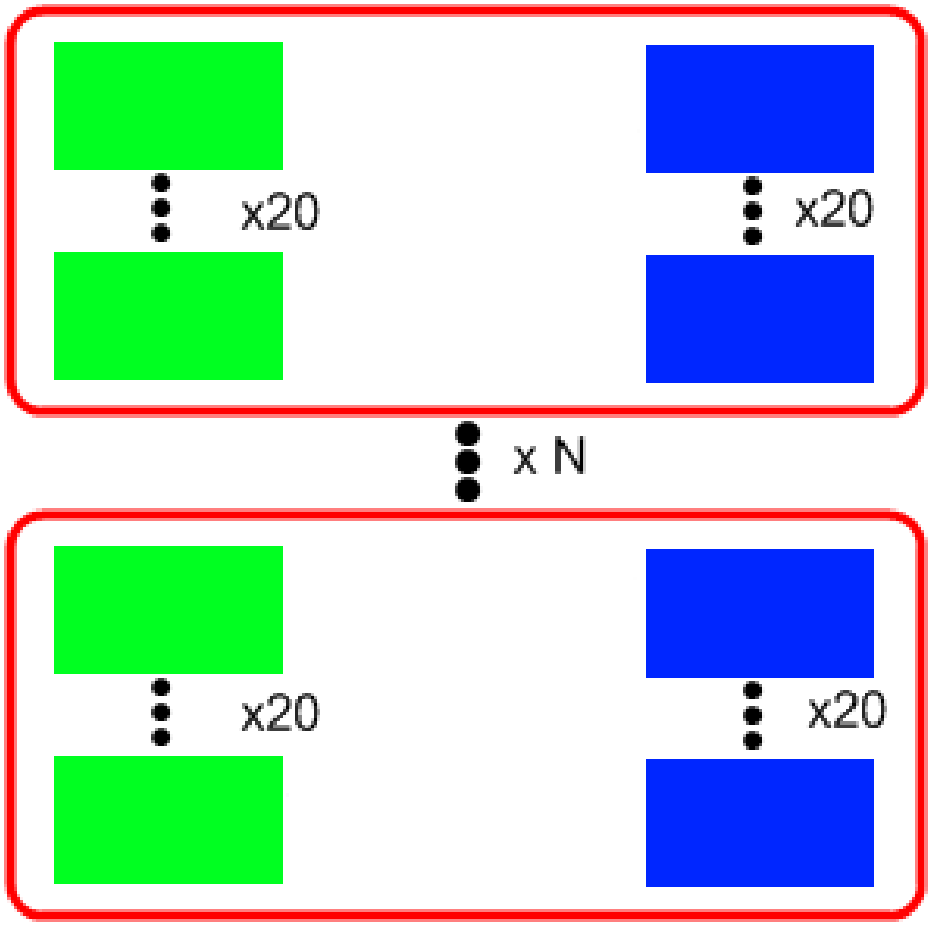
Hackbench

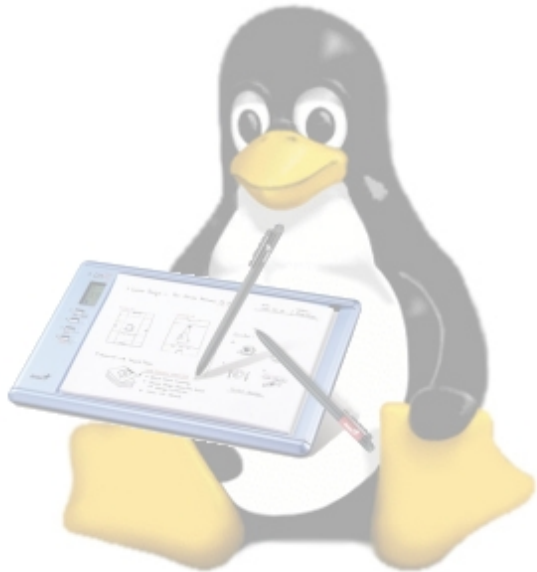
Tworzy N grup procesów, w każdej 20 klientów wysyła po 100 wiadomości do każdego z 20 serwerów.
Mierzy:

- wydajność (performance)
- narzut (overhead)
- skalowalność (scalability)

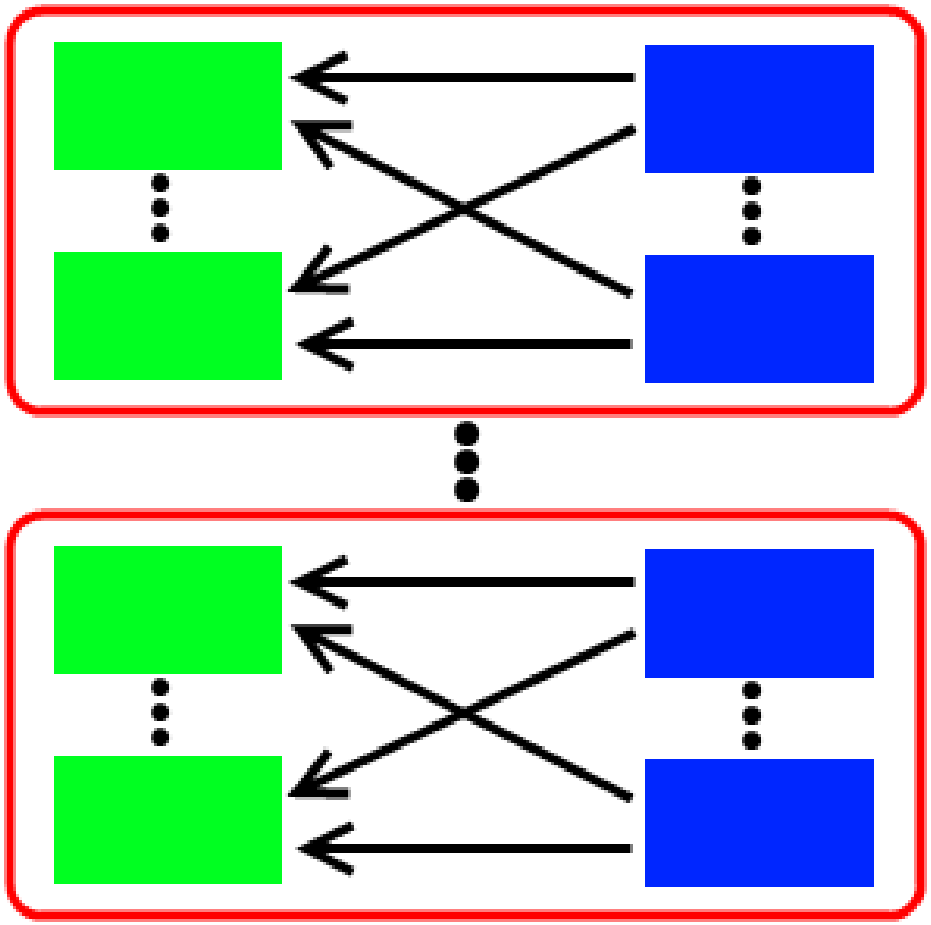


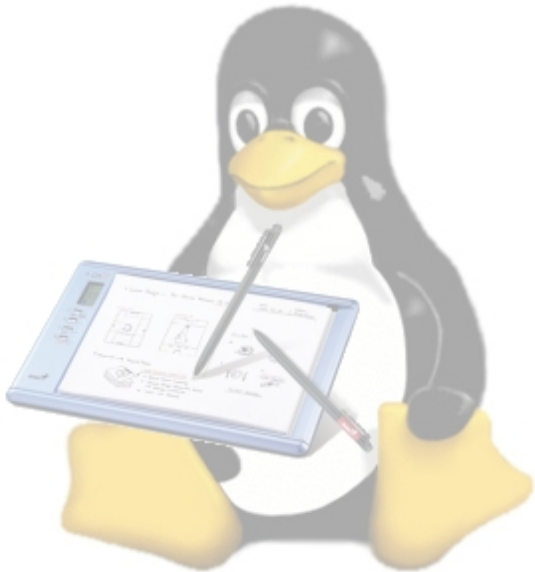
Hackbench



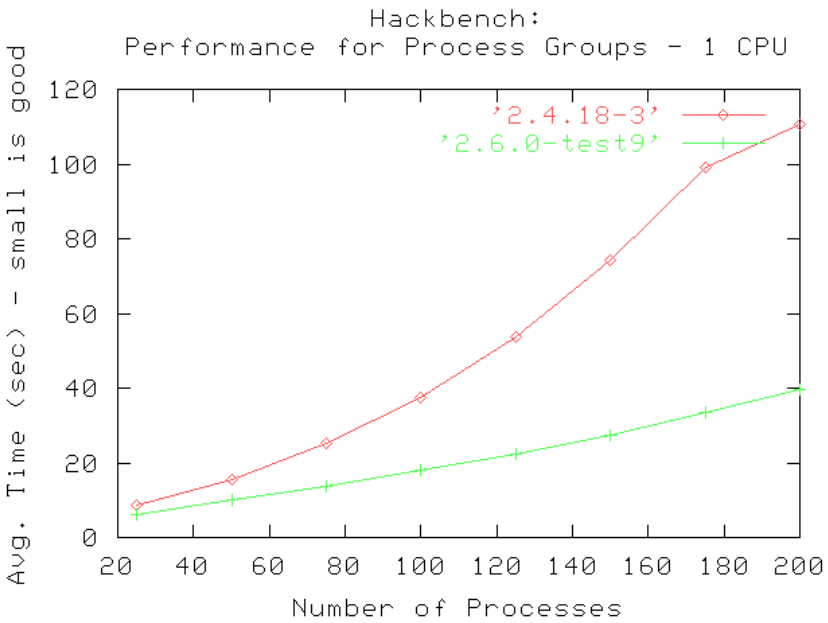


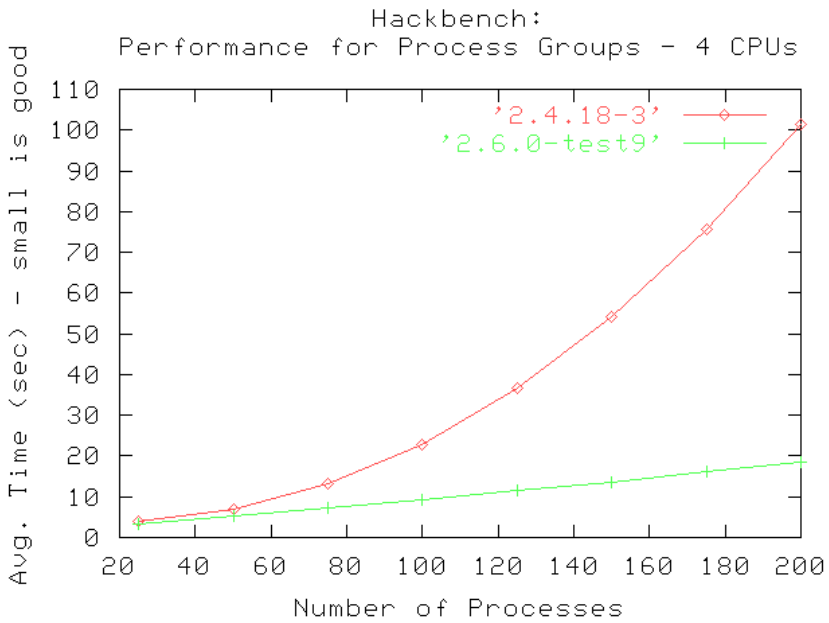
Hackbench



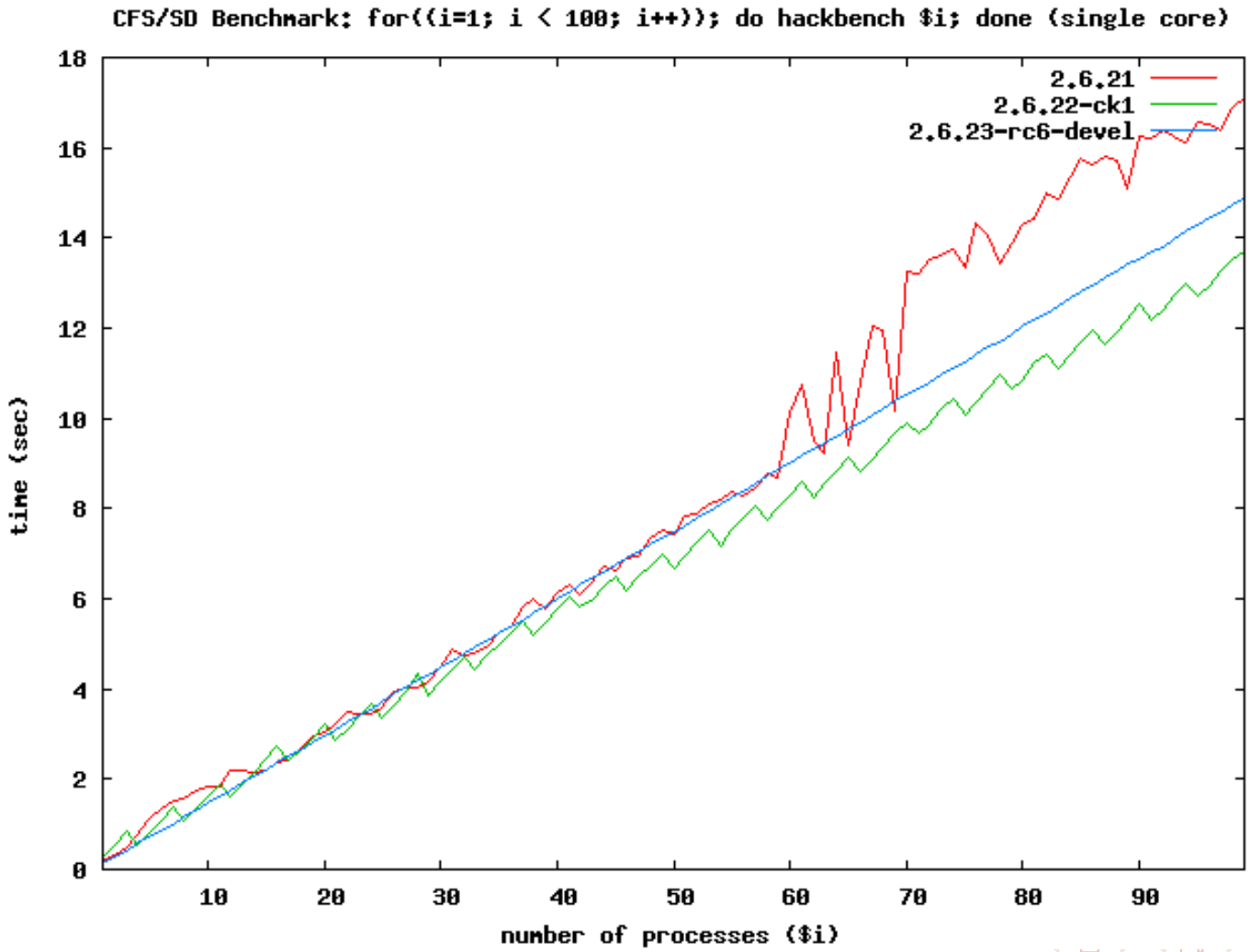


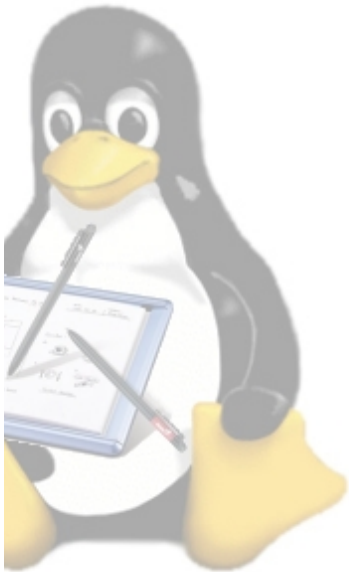
Hackbench - wyniki



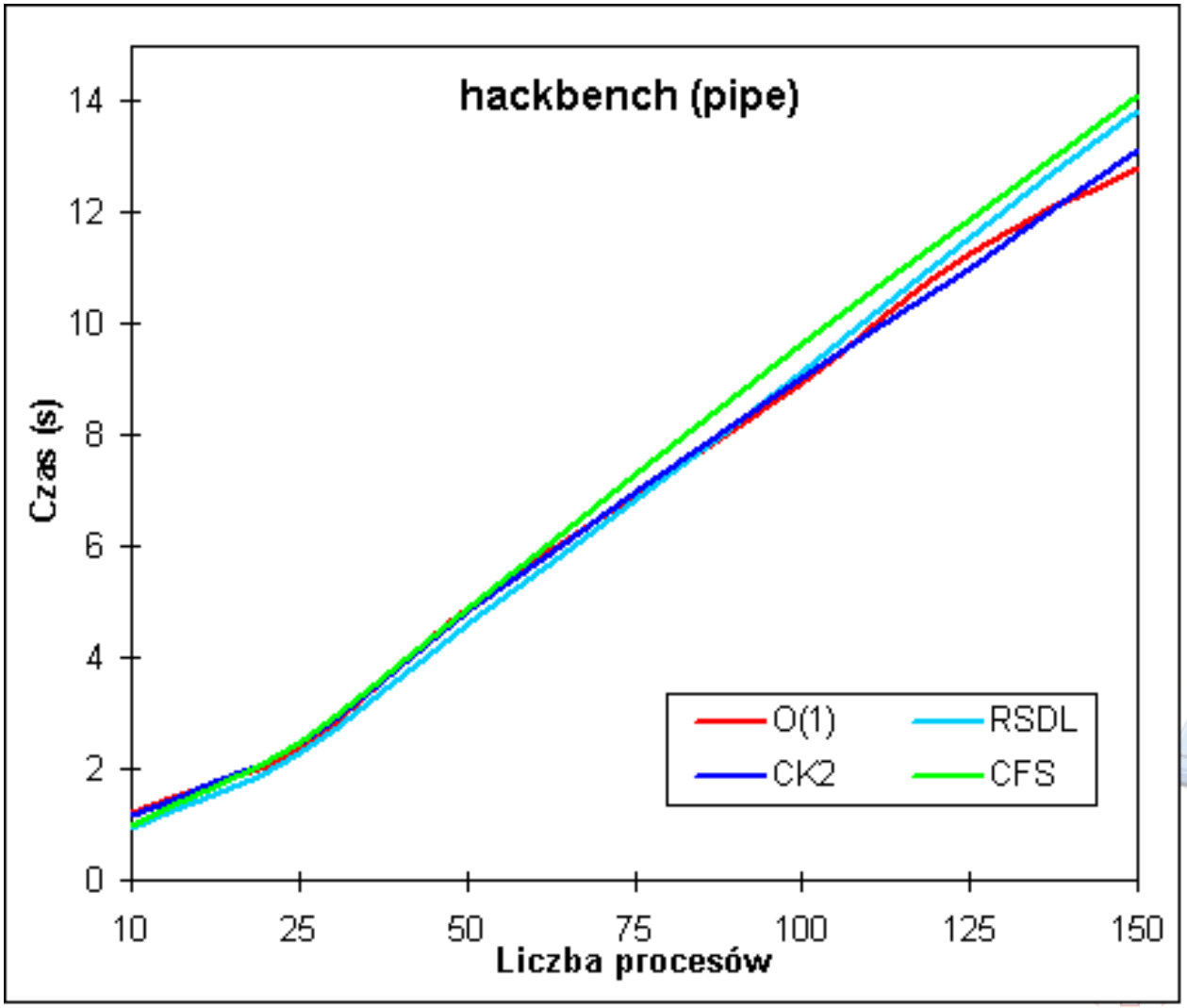


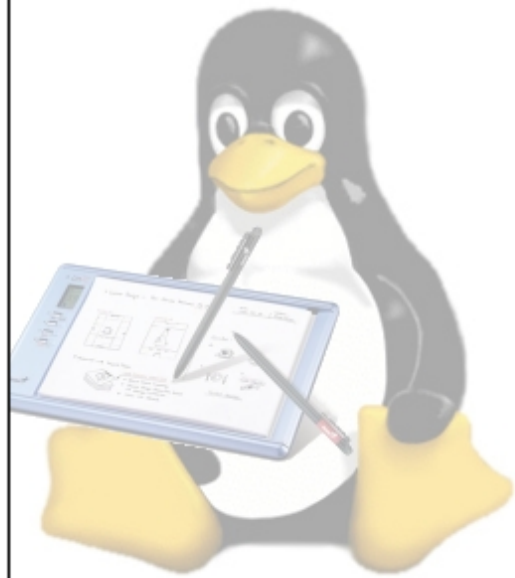
Hackbench - wyniki





Hackbench - nasze wyniki

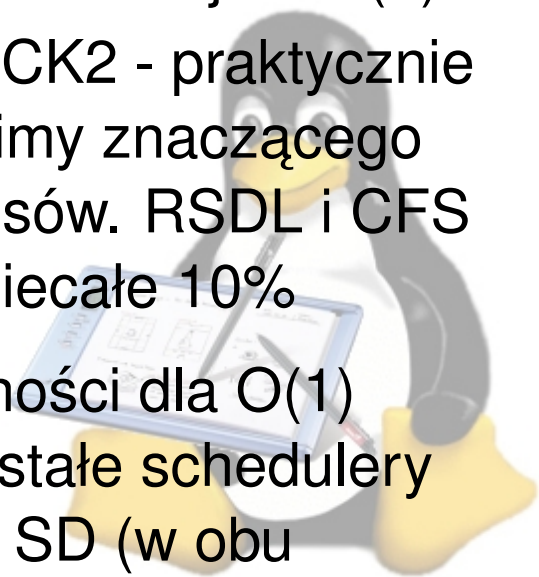




Hackbench - krótki komentarz

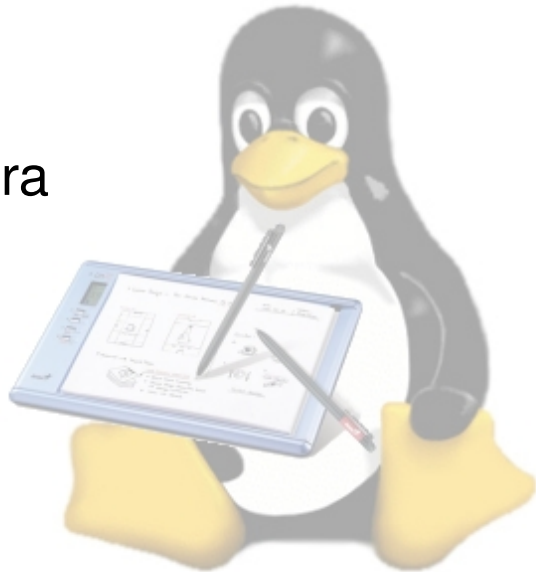
1. Potwierdzenie tezy, że scheduler jądra 2.4 jest dużo wolniejszy od O(1). Widać szczególnie brak dobrego wykorzystania wieloprocesorowości.
2. CK1 kolejny raz okazuje się najlepszy. CFS traci do niego 5%, wyraźnie wolniejszy dla ponad 60 procesów jest O(1)
3. W naszych testach najlepiej spisywał się CK2 - praktycznie na równi z O(1) dla którego nie zauważylimy znaczącego spadku wydajności dla dużej liczby procesów. RSDL i CFS tracą przy największej liczbie procesów niecałe 10%

Nie zauważyliśmy znaczącego spadku wydajności dla O(1) który jest widoczny w pierwszym teście. Pozostałe schedulery zachowują sie podobnie, jednak najlepszy był SD (w obu wersjach).

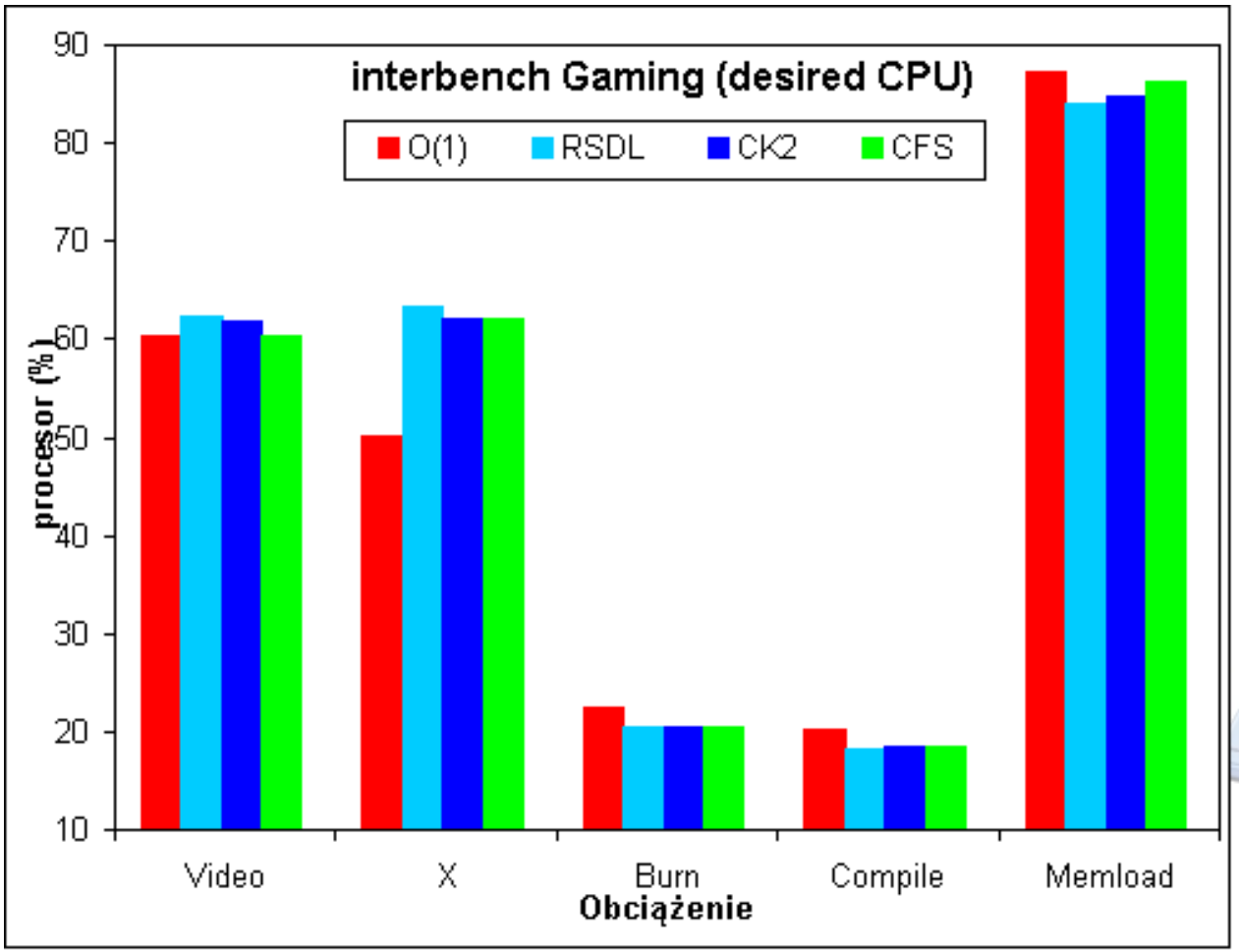


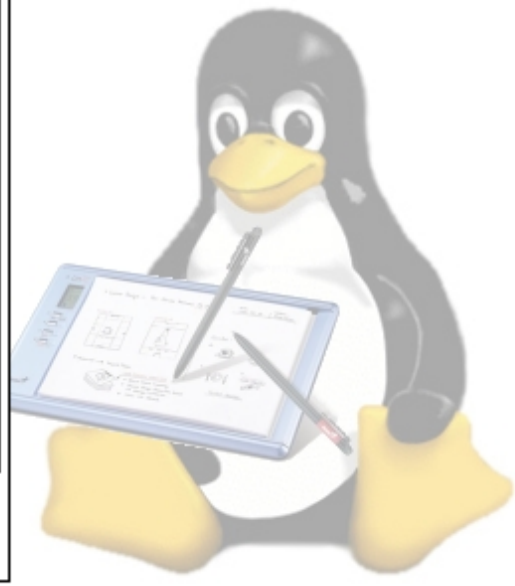
Interbench

- Mierzy interaktywność
- Symuluje działanie czterech typów aplikacji (Audio, Video, X, Games)
- Testując dany typ aplikacji obciąża dodatkowo system różnymi innymi procesami:
  - None - bez obciążenia
  - Burn - 4 procesy zajmują 100% procesora
  - Write - zapisuje duży plik na dysk
  - Read - wczytuje duży plik z dysku
  - Compile - trzy powyższe naraz
  - Memload - zajmuje 110% pamieci
  - Hack - uruchamia hackbench



Interbench - nasze wyniki





Interbench - krótki komentarz

W przypadku testu interaktywności w profilu dla gier schedulery spisywały się bardzo podobnie (w większości przypadków różnice na poziomie 5% na korzyść raz dla jednego raz dla drugiego). Tylko O(1) dla obciążenia serwerem X-ów zdecydowanie gorzej przydzielał procesor.

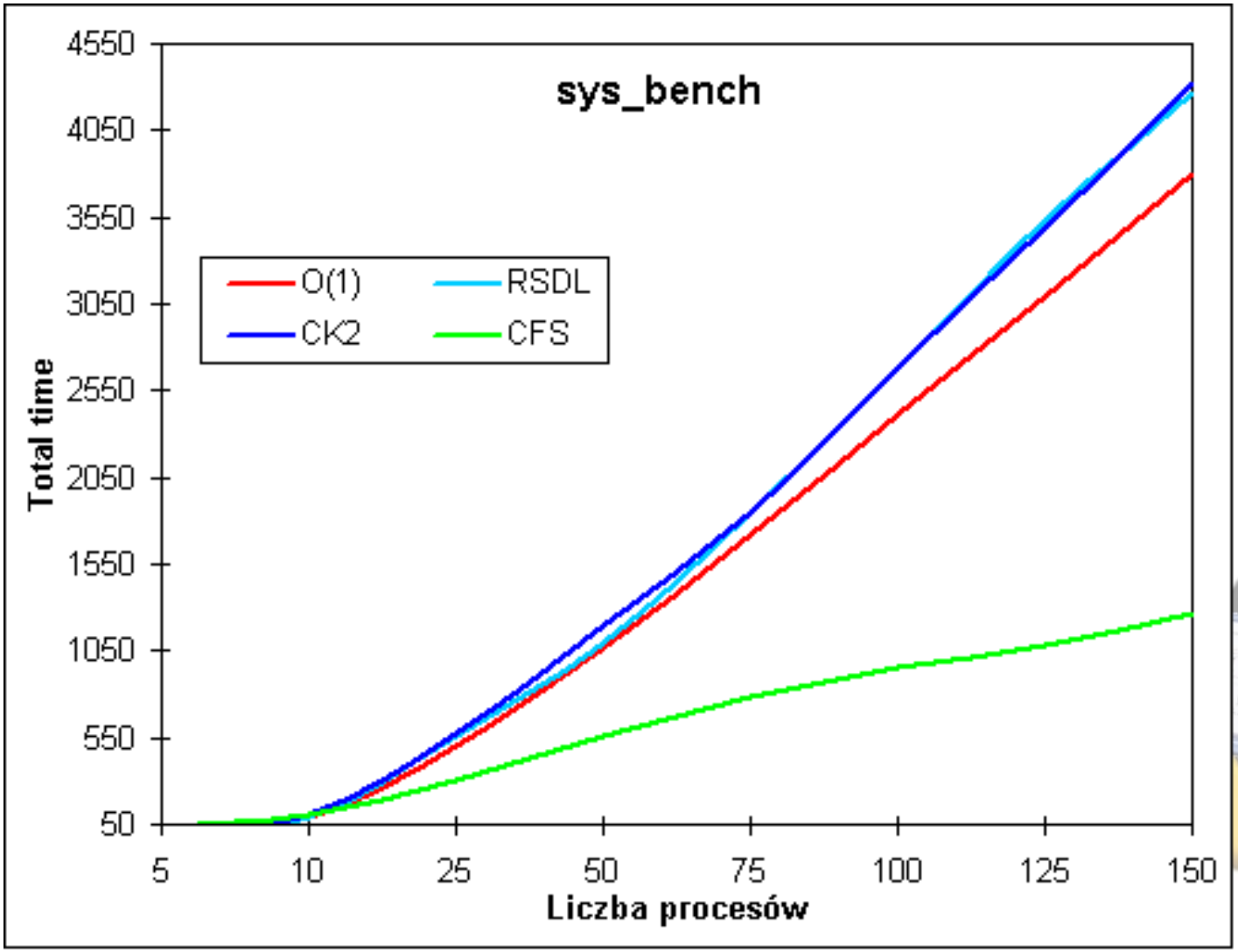
Dotychczasowe wyniki

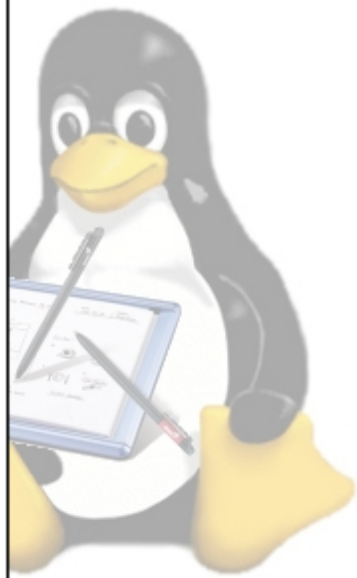
Jak narazie remis RSDL i CK2 (punkty przydzielane w sposób w miarę nielosowy)

- O(1) - 1
- CK2 - 3
- RSDL - 3
- CFS - 0

Najnowszy scheduler wcale nie jest taki dobry. Ale to jeszcze nie koniec...

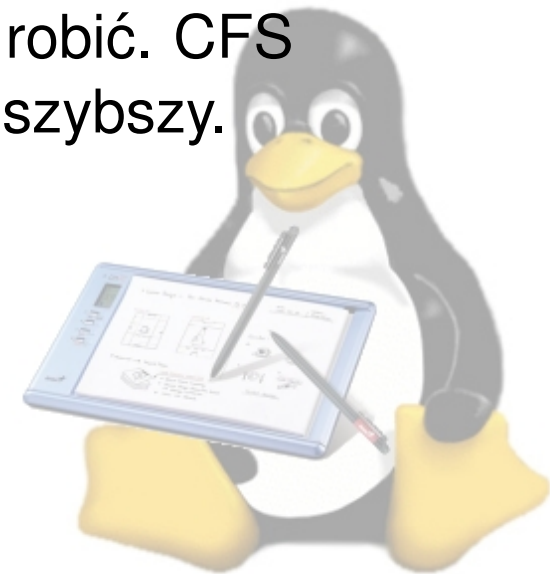
Sysbench - nasze wyniki



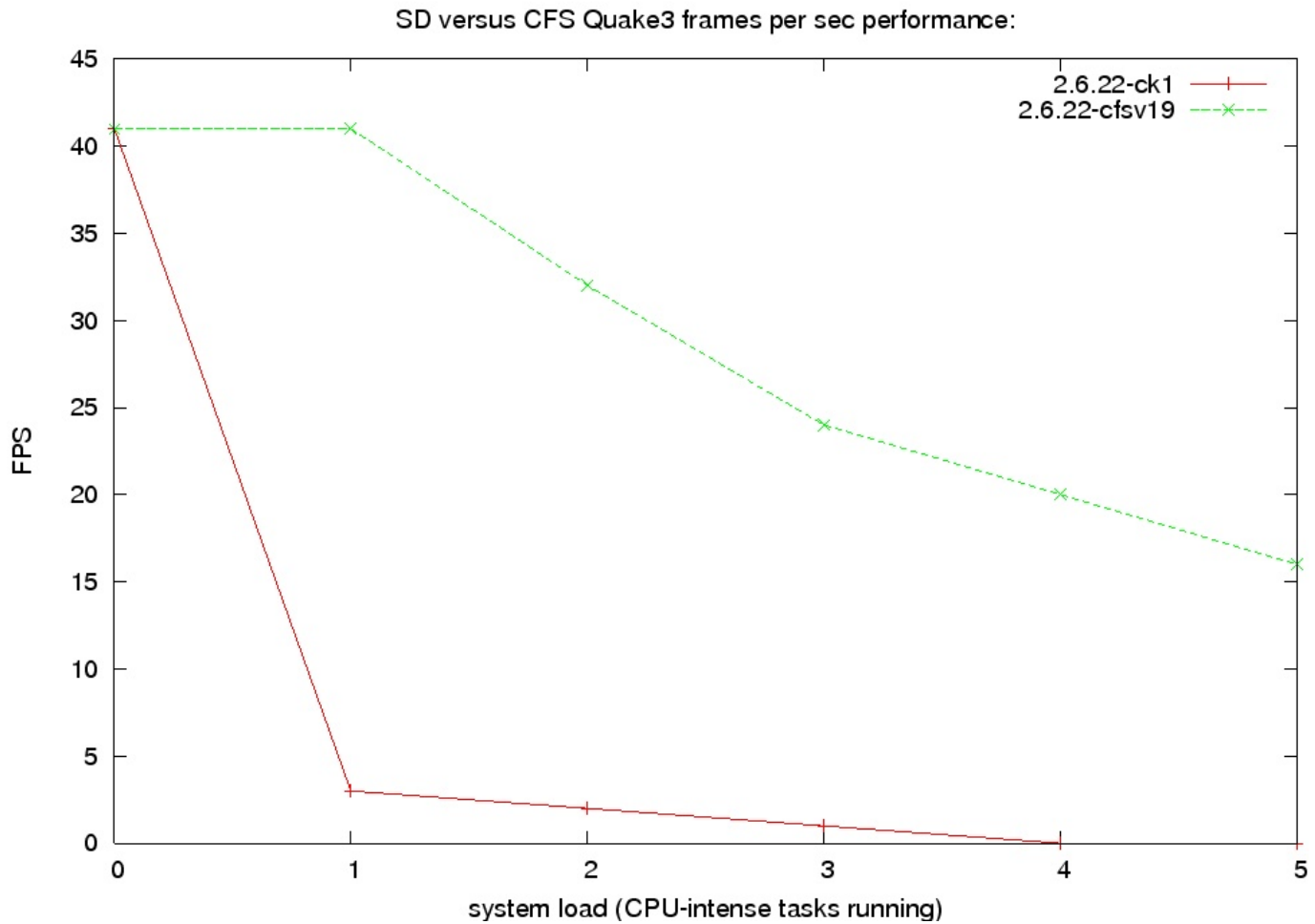


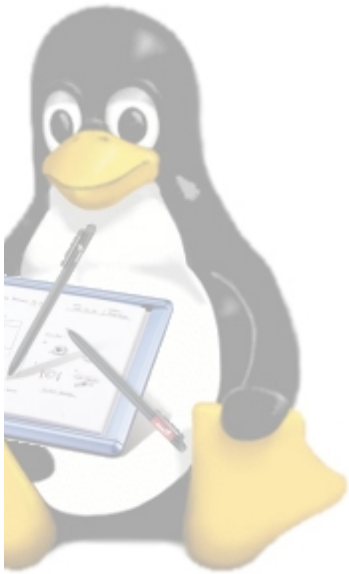
Sysbench - krótki komentarz

Sysbench to test który testuje system pod kątem MySQL. Jest bardzo dużo procesów i scheduler ma też co robić. CFS deklasuje rywali. Okazuje się nawet 3-4 razy szybszy.



Inne testy - Wine & Quake3

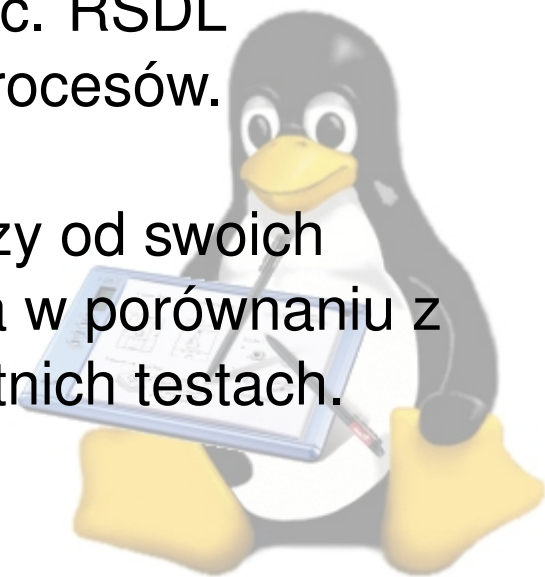




Wine & Quake3 - krótki komentarz

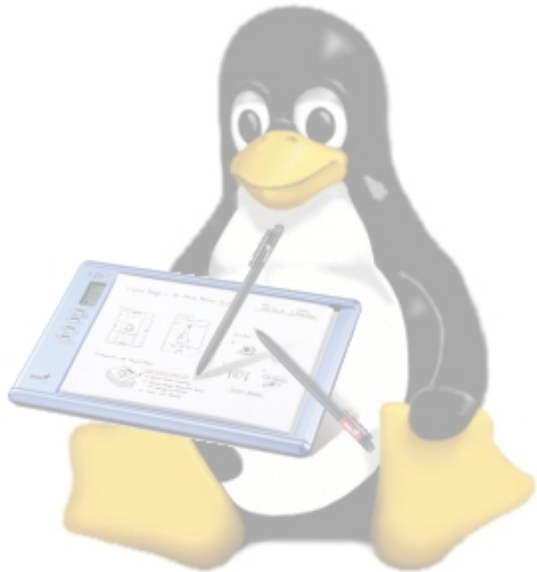
W teście przedstawiony jest CK1, ale zachowuje się on podobnie do CFS. RSDL znowu deklasuje rywala. Przy uruchomionym jednym intensywnie liczącym procesie w przypadku CFS praktycznie nie da się już grać. RSDL wytrzymuje obciążenie nawet do 3-4 takich procesów.

CFS w większości testów okazał się wolniejszy od swoich konkurentów, jednak różnica ta była niewielka w porównaniu z jego dużo lepszym sprawowaniem się w ostatnich testach.



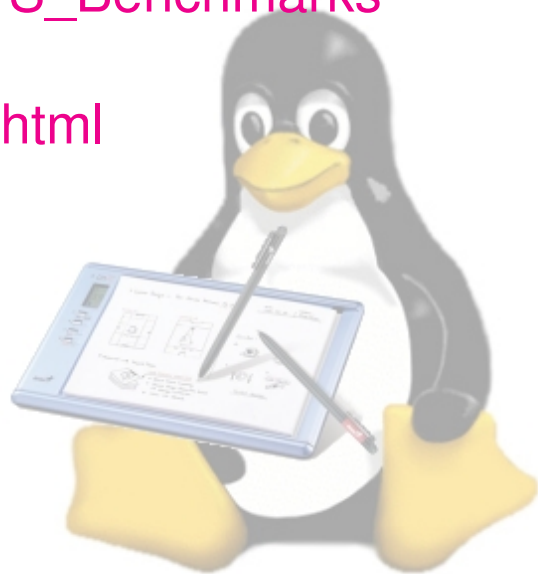
Bibliografia

- Understanding the Linux kernel / Daniel P. Bovet and Marco Cesati. 2nd ed.
- Scheduler 2.4
- RSDL na LWN.net
- CFS na LWN.net
- CFS na Kernel Newbies
- CFS



Bibliografia cd.

- Wyniki testów:
  - http://kerneltrap.org/Linux/Additional_CFS_Benchmarks
  - http://devresources.linux-foundation.org/craiger/hackbench/index.html
  - http://kerneltrap.org/node/14023



Bibliografia cd.

- Materiały o testowaniu schedulerów:
  - http://kerneltrap.org/Linux/Benchmarking_CFS
  - http://kerneltrap.org/node/14023
  - http://kerneltrap.org/node/14008?page=1
  - http://eaglet.rain.com/rick/linux/staircase/scase-vs-noscase-vs-1q.html
  - http://eaglet.rain.com/rick/linux/staircase/scase-vs-noscase.html
  - http://www.ibm.com/developerworks/linux/library/l-web26/
  - http://kerneltrap.org/node/3589
  - http://www.samag.com/documents/s=9367/sam0509c/0509c.htm
  - http://lkml.org/lkml/2007/4/21/81
  - http://kerneltrap.org/node/8082
  - http://www.2cpu.com/articles/98_1.html

Bibliografia cd.

- Patche:
  - http://www.kernel.org/pub/linux/kernel/people/ck/patches/2.6/2.6.2 ck2/
  - http://ck.kolivas.org/patches/staircase-deadline/
  - http://people.redhat.com/mingo/cfs-scheduler/