Szeregowanie procesów w Linuxie - trendy rozwojowe

Gabriel Kłosiński Bartosz Łoś

8 grudnia 2007

Spis treści

Rozdział 1

Wstęp

1.1 Wstęp

Witamy państwa na kolejnej prezentacji z systemów operacyjnych. Dzisiejszą prezentację poprowadzą Bartosz Łoś i Gabriel Kłosiński. Zajmiemy się problemem szeregowania procesów w systemie operacyjnym Linuks.

1.2 Streszczenie

Na początku streścimy plan prezentacji. Zaczniemy od krótkiego wprowadzenia. Przedstawimy podstawowe definicje dotyczące szeregowania procesów tak, aby nawet osoba, która nie jest ekspertem, mogła wynieść z tej prezentacji jak najwięcej użytecznych informacji.

Następnie kolega opowie o tym, jak szeregowano procesy kiedyś. Zwrócimy przy tym szczególna uwagę na tworzone w tamtym czasie algorytmy, ich zachowanie oraz interpolację współbieżności wykonywanych procesów.

Kolejna część będzie poświęcona najnowszej historii planistów w systemach linuksowych (skupimy się tutaj szczególnie na jądrach 2.4 oraz 2.6). Poznamy genezę ewolucji schedulerów (od schedulera O(1) do wersji obecnej (CFS))

Kolejny punkt będzie dotyczył wydajności poszczególnych schedulerów. Przedstawimy najnowsze wyniki osiągane przez poszczególnych planistów.

Naszą prezentację zakończymy krótkim podsumowaniem. Następnie będzie czas (miejmy taką nadzieję) na zadawanie pytań dotyczących prezentacji.

1.3 Co to jest szeregowanie procesów?

Problem szeregowania procesów to... problem algorytmiczny!! Dotyczy nie tylko systemów operacyjnych czy baz danych, ale również np. linii produkcyjnych.

Spróbujmy odpowiedzieć na pytanie, o co chodzi z tym szeregowaniem procesów?? Za Wikipedią: Algorytm szeregowania (ang. scheduler - planista) to algorytm rozwiązujący jedno z najważniejszych zagadnień informatyki - jak rozdzielić czas procesora i dostęp do innych zasobów pomiędzy zadania, które w praktyce zwykle o te zasoby konkurują. Najczęściej algorytm szeregowania jest

implementowany jako część wielozadaniowego systemu operacyjnego (tak jest właśnie w Linuksie), odpowiedzialną za ustalanie kolejności dostępu zadań do procesora. Oprócz systemów operacyjnych dotyczy w szczególności także serwerów baz danych. Problem szeregowania procesów nie jest wyłącznie problemem informatycznym, występuje tak naprawdę wszędzie, gdzie kilka procesów (zadań, ludzi...) walczy o dostęp do zasobów (np. ja wynajmuję mieszkanie wraz z kolegami i codziennie rano walczymy o to, żeby dostać się do łazienki). Za najwcześniejsze prace kładące podwaliny pod teorię algorytmów szeregowania, można uznać wprowadzenie linii produkcyjnej przez Henry'ego Forda.

1.4 Definicja

Teraz przedstawimy definicję szeregowania procesów. Matematycznie rzecz ujmując: Mamy zbiór $J = \{J_1, ..., J_n\}$ n zadań. Algorytm szeregowania tychże zadań jest to takie przydzielanie zadań procesorowi (lub procesorom), że każde zadanie jest przetwarzane do momentu zakończenia. Jednakże przetwarzanie danego zadania może być przerywane na bliżej nieokreślony czas. Dla jednego procesora jest to funkcja:

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

taka że:

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

Zgodnie z tą definicją jest to funkcja, która dzieli czas na przedziały i każdemu przedziałowi przyporządkowuje jedną wartość naturalną będącą numerem procesu, który ma się w tym przedziale czasu wykonywać. Przyjęte jest, że przyporządkowanie wartości równej 0 oznacza procesor w stanie bezczynności. Numer nie musi mieć związku z priorytetem zadania. Chociaż wyjaśnienie wydaje się proste, zaprojektowanie i implementacja dobrego algorytmu szeregowania nastręcza wielu trudności.

Rozdział 2

Wprowadzenie

2.1 Wprowadzenie

Algorytem szeregowania procesów nazywamy algorytm rozwiązujący jedno z najważniejszych zagadnień informatyki - jak rozdzielić czas procesora i dostęp do innych zasobów pomiędzy zadania, które w praktyce zwykle o te zasoby konkurują.

Definicja
Algorytm szeregowania n zadań stanowiących zbiór $J = \{J_1, ..., J_n\}$ jest to takie przydzielanie zadań procesorowi (lub procesorom), że każde zadanie jest przetwarzane do momentu zakończenia. Jednakże przetwarzanie danego zadania może być przerywane na bliżej nieokreślony czas. . Dla jednego procesora jest to funkcja:0

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

taka że:

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

Zgodnie z tą definicją jest to funkcja, która dzieli czas na przedziały i każdemu przedziałowi przyporządkowuje jedną wartość naturalną będącą numerem procesu, który ma się w tym przedziale czasu wykonywać. Przyjęte jest, że przyporządkowanie wartości równej 0 oznacza procesor w stanie bezczynności. Numer nie musi mieć związku z priorytetem zadania.

Rozdział 3

Algorytmy planowania

3.1 Zagadnienia planowania

Cel wieloprogromowania to jak najlepsze wykorzystanie procesora.Podstawowa idea polega na wykonywaniu procesu do chwili, w której będzie musiał czakać (np. na zakończenie operacji wejścia-wyścia). Korzyści polegają na lepszym wykorzystaniu procesora i wyższej przepustowości (ilość pracy wykonanej w jednostce czasu)

3.2 Własności algorytmów planowania:

- Wykorzystanie procesora
- Przepustowość (liczba procesów kończonych w jednostce czasu)
- Czas cyklu przetwarzania (czas upływający między chwilą nadejścia procesu do systemu a chwilą zakończenia procesu, tzn. suma okresów spędzonych na czekaniu na wejście do pamięci, czekaniu w kolejce procesów gotowych do wykonania, wykonywaniu procesu przez procesor i wykonywaniu operacji wejścia-wyjścia)
- Czas oczekiwania (można ograniczyć się do rozważania czasu, który proces spędza w kolejce procesów gotowych do wykonania, gdyż algorytm planowania nie ma wpływu na czas, w którym proces działa lub wykonuje operację wejścia-wyjścia)
- Czas odpowiedzi (w systemach interaktywnych czas cyklu przetwarzania może nie być najlepszym kryterium. Często bywa tak, że proces produkuje wyniki dość wcześnie i wykonuje następne obliczenia, podczas gdy rezultaty są prezentowane użytkownikowi. Toteż dodatkową miarą jest czas upływający między przedłożeniem zamówienia a pojawieniem się pierwszej odpowiedzi. Ta miara, nosząca nazwę czasu odpowiedzi, określa, ile czasu upływa do rozpoczęcia odpowiedzi bez uwzględnienia czasu potrzebnego na jej wyprowadzenie. Czas cyklu przetwarzania jest na ogół uzależniony od prędkości urządzenia wyjściowego.

Dąży się do maksymalizacji wykorzystania procesora i zwiększenia przepustowości oraz do minimalizacji czasu cyklu przetwarzania, oczekiwania i odpowiedzi. W większości przypadków optymalizuje się miarę średnią. Jednakże niekiedy optymalizacja wartości minimalnych lub maksymalnych może być bardziej pożądana niż troska o wynik średniej. Jeśli np. dbamy o dobrą obsługę wszystkich użytkowników, to może nam najbardziej zależeć na zmiejszeniu maksymalnego czasu odpowiedzi. Istnieje pogląd, że w systemach interakcyjnych ważniejsze jest minimalizowanie wariancji czasu odpowiedzi, aniżeli minimalizowanie średniego czasu odpowiedzi. System z sensownym i przewidywalnym czasem odpowiedzi może być bardziej pożądany niż system, który ma przeciętnie szybszy, ale zmienny czas reakcji.

3.3 Różne algorytmy planowania przydziału procesora

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

  Według tego schematu proces, który pierwszy zamówi procesor, pierwszy go otrzyma. Algorytm implementuje się za pomocą kolejki FIFO. średni czas oczekiwania przy zastosowaniu metody FCFS nie jest na ogół minimalny i może wykonywać znaczne wahania. Algorytm FCFS jest niewywłaszczający. Po objęciu kontroli nad procesorem proces utrzymuje ją do czasu, aż sam zwolni procesor wskutek albo zakończenia swego działania, albo żądania operacji wejścia-wyjścia.

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

  Algorytm wiąże z każdym procesem długość jego najbliższej z przyszłych faz procesora. Gdy procesor staje się dostępny, wówczas zostaje przydzielony procesowi mającemu najkrótszą następną fazę procesora. Jeśli dwa procesy, mają mają następne fazy procesora równej długości, to stosuje się algorytm FCFS. Algorytm SJF jest optymalny, tzn. daje minimalny średni czas oczekiwania dla danego zbioru procesów. Jednak nie może on być zaimplementowany na poziomie krótkoterminowego planowania przydziału procesora. Nie jesteśmy w stanie poznać długości następnej fazy procesora, lecz możemy spróbować oczacować jej wartość na podstawie długości faz poprzednich. Algorytm SJF może być wywłaszczający lub niewywłaszczający. W kolejce procesów gotowych może pojawić się proces, mający krótszą następną fazę procesora, niż to, co jeszcze pozostało do wykonania w procesie bieżącym. Wywłaszczający algorytm SJF usunie w tej sytuacji dotychczasowy proces z procesora (metoda najpierw najkrótszy pozostały czas), podczas gdy niewywłaszczający algorytm SJF pozwoli bieżącemu procesowi na zakończenie fazy procesora.

- **Planowanie priorytetowe**

Każdemu procesowi przypisuje się pewien priorytet, po czym przydziela się procesor temu procesowi, którego priorytet jest najwyższy. Procesy o równych priorytetach planuje się w porządku FCFS. Algorytm SJF jest zatem szczególnym przypadkiem algorytmu priorytetowego (priorytet to odwrotność przewidywanej fazy procesora). Priorytety mogą być definiowane wewnętrznie (na podstawie mierzalnej właściowości procesu np. limity czasu, wielkość obszaru wymaganej pamięci, liczba otwartych plików, stosunek średniej fazy wejścia-wyjścia do średniej fazy procesora) albo zewnętrznie (na podstawie kryteriów zewnętrznych względem systemu operacyjnego np. ważność procesu). Planowanie priorytetowe może być wywłaszczające lub niewywłaszczające. Wywłaszczający algorytm priorytetowy spowoduje odebranie procesora bieżącemu procesowi, jeśli jego priorytet jest niższy od priorytetu nowo przybyłego procesu. Niewywłaszczający algorytm priorytetowy ustawi po prostu nowy proces na czele kolejki procesów gotowych do wykonania.

Podstawowym problemem jest nieskończone blokowanie niskopriorytetowych procesów przez stały napływ procesów o wyższych priorytetach. Rozwiązaniem tego problemu jest postarzanie procesów (stopniowe podwyższanie priorytetów procesów długo oczekujących w systemie, wtedy każdy proces uzyska w końcu najwyższy priorytet w systemie i zostanie wykonany)

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

  Ustala się małą jednostkę czasu: kwant czasu. Kolejka procesów gotowych do wykonania jest traktowana jako kolejka cykliczna. Nowe procesy są dołączane na końcu kolejki procesów gotowych. Planista przegląda kolejkę i każdemu procesowi przydziela odcinek czasu. Proces może mieć fazę procesora krótszą niż jeden kwant czasu. Wówczas proces z własnej inicjatywy zwolni procesor. W przeciwnym razie nastąpi przerwanie zegarowe systemu operacyjnego. Dokona się przełączenie kontekstu i proces zostanie odłożony na koniec kolejki procesów gotowych, planista zaś wybierze następny proces z kolejki. Kwant czasu powinien być długi w porównaniu z czasem przełączania kontekstu.

- **Wielopoziomowe planowanie kolejek**

  Przyporządkowujemy procesy do różnych grup np. pierwszoplanowe (interakcyjne) i drugoplanowe (wsadowe). Algorytm wielopoziomowego planowania kolejek rozdziela kolejkę procesów gotowych na osobne kolejki. Procesy zostają na stałe przypisane do jednej z tych kolejek na podstawie np. typu procesu lub rozmiaru pamięci. Każda kolejka ma własny algorytm planujący. Planowanie między kolejkami może polegać na stałopriorytetowym planowaniu wywłaszczającym (każda kolejka ma bezwzględne pierwszeństwo przed kolejkami o niższych priorytetach) lub operowaniu przedziałami czasu między kolejkami (każda kolejka dostaje pewną porcję czasu procesora, aby go rozplanować między znajdujące się w niej procesy)

Rozdział 4

Scheduler w wersji jądra 2.4

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

4.1 Obliczenie priorytetów procesów

Funkcja goodness() oblicza dla danego procesu p wartość określającą ważność przełączenia aktualnego procesu na proces p. Jest ona wykorzystywana przez funkcję schedule().

- Dla procesu, który zgłosił chęć oddania procesora (ustawiony bit SCHED_FIELD w polu policy deskryptora procesu) zwraca wartość -1
- Dla procesu czasu rzeczywistego (SCHED_FIFO i SCHED_RR) zwraca wartość 1000 + rt priority
- Dla zwykłych procesów (SCHED_OTHER), którym zakończył się już kwant czasu w tej epoce ($counter == 0$) zwraca wartość 0
- Dla zwykłych procesów zwraca wartość równą ilości taktów, które pozostały procesowi do wykorzystania w tej epoce powiększoną o wartość $20 - nice$, gdzie nice jest bazowym kwantem procesu. Dodatkowo proces jest nagradzany, jeżeli w wyniku przełączenia kontekstu nie trzeba będzie ładować nowych stron pamięci.

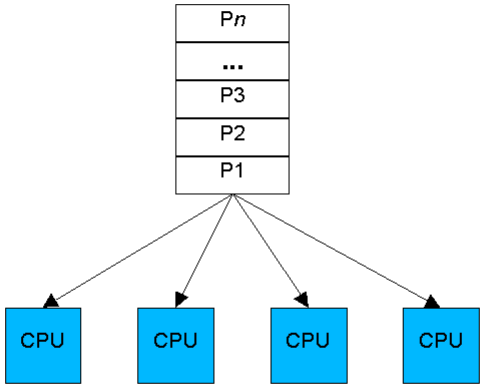
Zauważmy, że w ten sposób procesy czasu rzeczywistego otrzymują zawsze priorytet wyższy od dowolnego zwykłego procesu.

W szczególnym przypadku możliwe jest poinformowanie planisty, iż proces powinien być uruchomiony na konkretnym procesorze, nawet jeśli inny proces zostanie zwolniony wcześniej

Funkcja preemption_goodness zwraca różnicę pomiędzy ważnością zmiany aktualnego procesu na proces p a pozostawieniem aktualnego procesu działającego dalej. Będzie ona używana w funkcji reschedule_idle.

4.2 Opis algorytmu szeregowania procesów dla jądra 2.4

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



Ten model ma zaletę, iż jest prosty do implementacji oraz umożliwia stosunkowo proste debugowanie.

Wybór przez procesor kolejnego procesu wiąże się z założeniem blokady na kolejkę typu czytelnika-pisarza. To pozwala kilku procesorom na szukanie współbieżnie zadania do wykonania. Zmiany wymagają wyłącznego dostępu.

To rozwiązanie posiada również znaczące wady. Pojedyncza blokada typu czytelnika-pisarza stała się powodem konfliktu, zarówno w przeciążonych systemach, jak i systemach z czterema lub większą liczbą procesorów. Tylko jedna kolejka procesów jest używana przez wszystkie procesory. Kolejka ta jest przeglądana w całości. W przypadku, gdy system jest coraz bardziej obciążony, kolejka ta wydłuża się. Wtedy wydłuża się również algorytm liniowego przeszukiwania najlepszego kandydata. W wyniku tego, opóźnia się moment w którym zdecydujemy, który proces zostanie uruchomiony. Zatem później założymy wyłączną blokadę na kolejkę, aby usunąć proces z procesów gotowych oraz później oznaczymy go jako uruchomiony. Wtedy możliwe jest iż jakiś procesor wybrał z listy zadań proces, który już został już uruchomiony przez inny procesor. Wtedy procesor ten musi powtórzyć liniowy algorytm szukania najlepszego kandydata.

W miarę gdy system jest coraz bardziej zajęty, planista konsumuje coraz więcej czasu procesora, do momentu kiedy szeregowanie procesów zajmuje więcej czasu, niż ich uruchamianie. To może doprowadzić do zawieszenia systemu.

4.3 Pola task_struct istotne dla scheduler'a 2.4

- $p->counter$ - liczba cykli pozostałych do działania w tym czasie, który został przydzielony procesowi, aktualizowany przez zegar. W przypadku, gdy wartość staje się mniejsza lub równa 0, zostaje resetowany na 0 oraz $p->need_resched$ jest ustawiany. Pole to nazywane bywa "dynamicznym priorytetem"procesu, ponieważ może być przez niego zmieniane

- $p->priority$ - statyczny priorytet procesu, zmieniany jedynie przez wywołania systemowe, takie jak: nice, sched_setparam (POSIX.1b) oraz setpriority (4.4BSD/SVR4)

- $p->rt\_priority$ - priorytet czasu rzeczywistego

- $p->policy$ - specyfikuje do której klasy szeregowania procesów dany proces należy. Proces może zmieniać swoją klasę szeregowania poprzez użycie wywołania systemowego: sched_setscheduler. Możliwe wartości to: SCHED_OTHER (zwyczajny proces w unix), SCHED_FIFO (POSIX.1b FIFO proces czasu rzeczywistego) and SCHED_RR (POSIX round-robin proces czasu rzeczywistego). Każda z tych wartości może dodatkowo podana z OR SCHED_YIELD na zaznaczenie, iż proces może zostać wywłaszczony, np. poprzez wywołanie systemowe sched_yield. Procesy FIFO czasu rzeczywistego działają dopóki: a) W kolejce procesów gotowych pojawia się proces czasu rzeczywistego o wyższym priorytecie (o wyższej wartości $p->rt\_priority$) b) Zablokuje się w oczekiwaniu na zasób c) Dobrowolnie zrezygnuje z procesora d) Zakończy się Klasa SCHED_RR jest tym samym co SCHED_FIFO, poza tym, iż w przypadku, gdy minie przyznany kwant czasu, proces wraca na koniec kolejki procesów oczekujących na uruchomienie

4.4 Algorytm schedule() dla scheduler'a 2.4

Najważniejsze zmienne stosowane przez schedule():

- prev deskryptor procesu, który był do tej pory aktualny

- next na tą zmienną musi zostać zapisany wskaźnik do deksryptora procesu, który otrzyma procesor w wyniku szeregowania

- c priorytet znalezionego do tej pory najlepszego kandydata na przydział procesora

1. Jeżeli jesteśmy w obsłudze przerwania to błąd.

2. Jeżeli poprzedni proces był procesem czasu rzeczywistego Round Robin przerzuca się go na koniec kolejki oraz jeżeli potrzeba odnawia kwant czasu.

3. Jeżeli stan procesu jest różny od TASK_RUNNING to znaczy, że proces ma zostać zablokowany (tzn. usunięty z runqueue). Jednak jeżeli proces jest w stanie TASK_INTERRUPTIBLE i ma jakieś nie obsłużone sygnały, to daje się mu szansę je obsłużyc.

4. Czyszczenie pola need_resched:

$$prev->need\_resched = 0;$$

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

$$next = idle\_task(this\_cpu);\ c = -1000;$$

6. Jeżeli proces ostatnio wykonywany jest dalej w trybie TASK_RUNNING to jest wybierany jako pierwszy kandydat do procesora zamiast procesu idle.

7. Przeszukanie kolejki runqueue w poszukiwaniu procesu o najwyższym priorytecie. Procesowi temu zostanie przydzielony procesor.

8. Sprawdzenie czy nie zakonczyła się epoka. Jeśli tak, to przydzielenie wszystkim procesom nowych kwantów czasu.

9. Jeżeli nowowybrany proces jest tym samym procesem, który do tej pory był aktywny, to nic więcej już nie trzeba robić poza wyzerowaniem flagi SCHED_YIELD.

10. Rozpoczęcie przełączania procesów

4.5 Porównanie scheduler'a z wersji 2.6 z 2.4

Dotychczas scheduler działał według następującego algorytmu:

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

Jak się łatwo domyślić, im więcej procesów działa w systemie, tym dłużej trwa wybieranie tego najważniejszego. Może nawet dojść do sytuacji, w której wybór procesu zajmie więcej czasu niż jego wykonanie.

Z tego powodu zabrano się za stworzenie nowego schedulera, który nie dość, że będzie wydajniejszy, to jeszcze będzie działał idealnie na maszynach wieloprocesorowych. Przy jego projektowaniu zwracano również uwagę na to, żeby żaden proces nie zabrał zbyt wiele czasu procesora, ani żadne zadanie nie zostało tego czasu kompletnie pozbawione.

Rozwiązaniem jest tzw. scheduling $O(1)$, który zapewnia stały czas przełączania między procesami, bez względu na ich ilość.

Nowy scheduler ma jedną podstawową cechę - cały czas śledzi wszystkie procesy i ma ich listę, na której znajdują się priorytety poszczególnych zadań. Algorytm działania schedulera O(1) jest następujący:

- pobierz priorytet najbardziej "pilnego"procesu
- pobierz z listy pierwszy proces, który ma taki priorytet
- uruchom zadanie

Dzięki temu nie jest konieczne przeglądanie tablic wszystkich procesów w systemie, bo wszystkie dane są zgromadzone w jednym miejscu.

Drugą ważną sprawą, o której często zapominamy, jest skalowalność na maszyny wieloprocesorowe. W jądrach serii 2.4 wyglądało to tak, że istniała jedna kolejka procesów, z których scheduler pobierał procesy do wykonania. Problem polegał na tym, że sam kod schedulera mógł być wykonywany tylko na jednym procesorze na raz - w przeciwnym wypadku działające jednocześnie schedulery mogłyby nawzajem namieszać sobie w globalnej kolejce.

Rozwiązanie tego problemu jest bardzo proste - trzeba stworzyć odrębne kolejki dla każdego procesora. Dodatkową korzyścią z tego płynącą jest fakt, że teraz zadanie nie będzie skakać z procesora na procesor, lecz będzie wykonywane na jednym aż do momentu zakończenia lub pojawienia się dużej różnicy w obciążeniu procesorów.

Jaki wpływ na ostateczną szybkość systemu ma wprowadzenie nowego schedulera? Jak wynika z testów - ogromny. W ciągu jednej sekundy, pod kontrolą jądra 2.4, testowe procesy wymieniły między sobą około 80 tysięcy komunikatów. W tym samym czasie, te same procesy pod kontrolą 2.5 wymieniły tych komunikatów ponad 612 tysięcy!

Rozdział 5

Scheduler O(1)

5.1 Podstawowe fakty

W wersji 2.5 (eksperymentalna wersja jądra) Ingo Molnar opracował scheduler O(1) („Nowy Skalowalny Algorytm Szeregowania Procesów"). Nowy scheduler okazał się na tyle lepszym rozwiązaniem od O(n), że bardzo często był backportowany do wersji jądra 2.4 w komercyjnych wersjach linuksa.

„Nowy" scheduler ma 7200 lini kodu. Głównym celem nowego algorytmu jest zachowanie wszystkich tych dobrych rzeczy, które znamy ze schedulerza O(n) i wyeliminowanie tych przypadków, z którymi O(n) sobie nie radzi, m.in.:

- Szybsze działanie (wyeliminowanie „zmiany epoki")
- Lepsza współpraca z komputerami wieloprocesorowymi
- Polepszenie interaktywności

Dokładny opis działania schedulera O(1) został przedstawiony na wykładzie. Dla przypomnienia przedstawimy kilka najważniejszych zmian, które zostały wprowadzone. Rdzeń nowego algorytmu szeregującego stanowią następujące mechanizmy:

- dwie, uszeregowane względem priorytetu „tablice" na procesor. Jest tablica „aktywna" i tablica „wygasła". Tablica aktywna zawiera wszystkie zadania, które są spowinowacone z tym procesorem i jeszcze został im jakiś kwant czasu. Tablica wygasła zawiera wszystkie zadania, które już zużyły swoje kwanty czasu, ale ta tablica również jest posortowana. Do żadnej z tych tablic nie odwołujemy się bezpośrednio, ale przez dwa wskaźniki w strukturze runqueue dla każdego CPU. Gdy wszystkie zadania już wykorzystają swój czas, wówczas „zamieniamy" te dwa wskaźniki i teraz tablica wygasła staje się tablicą aktywną, a pusta tablica aktywna zaczyna służyć jako zbiór zadań, które zużyły swój kwant czasu.
- rozwiązanie z dwoma tablicami pozwala nam na posiadanie dowolnej liczby aktywnych i wygasłych zadań, a ponowne przeliczenie kwantów czasu może być wykonane, gdy tylko przydzielony kwant czasu się skończy. Ponieważ tablice są zawsze dostępne za pośrednictwem wskaźników w kolejce zadań do wykonania, zamienianie tablic może być wykonane bardzo szybko.

- dla każdego zadania istnieje „estymator obciążenia".
- staramy się „dożywiać" interaktywne zadania i „karać" zadania, które chcą zużyć więcej czasu procesora niż jest dostępne.
- każdy procesor otrzymuje oddzielne tablice z procesami
- wprowadzono funkcję load_balancer. Jej zadaniem jest balansowanie zużycia procesorów (uruchamiana co 200ms)

5.2 Wady O(1)

Niestety, okazało się, że O(1) również ma swoje wady. Pomimo tego, że działał zdecydowanie lepiej od swojego poprzednika, ciągle nie spełniał oczekiwań wszystkich internautów. Użytkownicy Linuksa zwracali uwagę na przypadki, w których scheduler nie zapewniał pełnej interakcji. Scheduler przydzielał zbyt mało czasu procesora na takie funkcje jak wyświetlanie interfejsu użytkownika czy odtwarzanie dźwięku (ogólnie - programy wykonujące nieskończone pętle). Z punktu widzenia użytkownika tworzyło to wrażenie zacinania się i małej reaktywności.

Drugą ważną wadą schedulera O(1) było nieracjonalne przydzielanie czasu procesora dla poszczególnych zadań w zależności od priorytetu. Dla przykładu rozważmy następujące przypadki. Mamy jeden procesor, zmiana kontekstu następuje np. po 100ms. Rozważmy kilka przypadków:

- Dwa procesy z priorytetem nice ustawionym na 19.
- Dwa procesy z priorytetem nice ustawionym na 0.

W obu przypadkach procesy otrzymają połowę czasu procesora, ale w pierwszym przypadku zmiana kontekstu będzie następowała co 5ms, a w drugim co 100ms.

Jak widać, schedulery O(n) i O(1) miały dość poważne wady i ograniczenia. Ciągle pracowano nad schedulerami i estymatorem interaktywności. Jedną z osób zaangażowanych w pracę był Con Kolivas. Osiągnął bardzo pozytywne wyniki. Jego ścieżką podążyli inni -Mike Galbraith, Davide Libenzi, Nick Piggin. Wszyscy oni próbowali polepszyć schedulera. Jednak okazało się, że rozwiązanie nie jest całkiem proste. Istniały przypadki, gdzie nowe metody sobie nie radziły lub działały dużo gorzej. Próba rozwiązania jednego problemu powodowała pojawienie się nowych.

Rozdział 6

Schedulery sprawiedliwe - RSDL i CFS

6.1 Rozwiązanie problemu braku interaktywności

Problemem okazał się estymator interaktywności, czyli kawałek kodu odpowiedzialny za interakcję z użytkownikiem. Con Kolivas jako pierwszy stwierdził, że estymator jako taki nie jest dobrym rozwiązaniem. Powoduje więcej problemów niż je rozwiązuje.

Po prostu estymator nie był w stanie przewidzieć i obsłużyć wszystkich pojawiających się sytuacji. Problemem było założenie, jak ma działać algorytm. Oryginalnie używał on statystyk do tego, by przewidzieć przyszłość wykorzystując do tego różne heurystyki. Trzeba było wymyślić coś innego, gdyż praca nad obecnym kodem, przy tych założeniach, nie mogła dać znacząco lepszych rezultatów.

W związku z tym Con rozpoczął pracę na zmianą koncepcji szeregowania procesów. Po wielu próbach i testach stwierdził, że najlepszym rozwiązaniem będzie traktowanie wszystkich procesów równo (koncepcja „fairness"). Tylko takie podejście do kwestii szeregowania zadań gwarantuje sprawiedliwy przydział czasu procesora.

Należy podkreślić, że sam pomysł był genialny w swojej prostocie, zresztą najczęściej jest tak, że proste rozwiązania są najlepsze. Za wpadnięcie na taki pomysł Conowi należą się wielkie słowa uznania.

6.2 RSDL

Kiedy Con wpadł na swoją genialną myśl, zaczął wprowadzać swój pomysł w życie. Efektem tych prac był nowy Scheduler RSDL - Rotating Staircase Deadline Scheduler. W nowym schedulerze, zgodnie z koncepcją "fairness", wszystkie procesy są traktowane równo. Estymator został wyrzucony. Każdemu procesowi dajemy ten sam kawałek czasu (kwant). Nie staramy się ustalić, któremu procesowi należałoby przyznać więcej czasu.

Nowy scheduler okazał się sukcesem. Udało się wyeliminować wszystkie wady poprzednich schedulerów, między innymi „interaktywność". Wyniki uzyskiwane przez RSDL-a w testach były zdecydowanie lepsze niż jego poprzedników. Wszyscy byli pod wielkim wrażeniem nowego planisty, który wręcz deklasował inne schedulery. Planowano dołączyć RSDL-a do kolejnych oficjalnych wersjach Linuksa.

Niestety, rzeczywistość okazała się o wiele bardziej skomplikowana. RSDL, jak każdy nowy kod, wymagał ciągłych testów i poprawek. Niestety, Con bardzo niechętnie współpracował z osobami wskazującymi na potencjalne wady schedulera, kierującymi uwagi odnośnie najnowszego planisty, chcącymi pomóc w udoskonalaniu jego „dziecka".

6.3 CFS (Completely Fair Scheduler)

W związku z pojawieniem się nowego, doskonałego schedulera, twórca planisty O(1), Ingo Molnar, zdecydował się na stworzenie swojej wersji "sprawiedliwego" planisty. Najnowszy scheduler otrzymał nazwę CFS - Completely Fair Scheduler, (czyli planista traktujący wszystkie procesy równo). Głównym pomysłem, podobnie zresztą jak w przypadku RSDL, było równe traktowanie procesów.

Sam scheduler powstał tak naprawdę znikąd. Ingo nie informował nikogo o swoich planach, więc jego poczynania nie były wcześniej dyskutowane w środowisku linuksowym. Pierwsza wersja CFS-a, 100K patch, został napisany zaledwie w 62h. Ingo rozpoczął pracę w środę o 8.00, a zakończył w piątek o 22.00 (wtedy Ingo stwierdził: „okay, to wygląda nieźle"). CFS pojawił się w sieci 6 godzin później. W tym czasie Ingo przeprowadzał niezbędne testy, by, jak to określił, „nie wysadzić komputerów". W czasie ostatnich dwóch godzin testów, Ingo po raz pierwszy udostępnił swój kod dwóm osobom, które znalazł na IRC-u.

W odróżnieniu od Cona, Ingo podjął współpracę nad poprawianiem kodu schedulera. Osobami szczególnie zaangażowanymi w pracę nad najnowszym schedulerem byli: Mike Galbraith, Peter Zijlstra, Thomas Gleixner, Suresh Siddha. Przy tworzeniu nowego schedulera ok. 70% kodu poprzedniego schedulera zostało zmienione. Obecnie CFS jest o wiele bardziej rozbudowany od swojego konkurenta (RSDL – 88K; CFS – 290K).

Pojawił się probem, który scheduler wybrać do głównej linii jądra? Sprawa była głośno komentowana w środowiskach linuksowych. Przeprowadzono przeróżne testy mające pokazać wyższość jednego planisty nad drugim. W końcu wybrano CFS... Wtedy Con ogłosił, że wycofuje się z tworzenia jądra linuksa. Przypuszczano, że jest to spowodowane tym, że wybrano CFS-a. Później Con publicznie oświadczył, ze nie ma to związku z tą sprawą. Jaka jest prawda, niewiadomo. Wydaje się, że jedynym powodem wyboru CFS-a były względy czysto techniczne.

6.4 Koncepcja "sprawiedliwości" w CFS

Głównym pomysłem, podobnie zresztą jak w przypadku RSDL, było równe traktowanie procesów. W idealnym przypadku powinno być tak, że CPU rozdziela po równo swoją moc obliczeniową dla każdego procesu. W momencie, kiedy na maszynie jednoprocesorowej działa n procesów, każdy z nich powinien otrzy-

mać 1/n mocy obliczeniowej procesora. Na komputerach jednoprocesorowych może jednocześnie działać tylko jeden proces, więc inne procesy muszą czekać na swoją kolej – powoduje to pewnego rodzaju niesprawiedliwość. Scheduler stara się ją wyeliminować.

6.5 Praktyka

CFS po prostu interpoluje idealne równoległe wykonywanie zadań na komputerze.

Jak to działa w praktyce? Zamiast list procesów zaimplementowano drzewa czerwono – czarne. Kolejny proces, który ma dostać zasoby, jest wybierany z drzewa czerwono – czarnego jako proces o najmniejszym kluczu.

W CFS-ie opóźnienie jest odnotowywane i śledzone dla każdego procesu poprzez wartość p->wait_runtime (jednostką czasu jest nanosekunda). Dokładniej rzecz ujmując, ta wartość oznacza czas, w którym proces powinien teraz dostać CPU, by powrócić do idealnego stanu (w idealnym komputerze wartość p->wait_runtime zawsze powinna wynosić zero)

Wybór, który proces powinien się w danym momencie wykonywać, jest oczywisty – wybieramy ten proces, którego wartość p->wait_runtime jest największa – w ten sposób staramy się jak najbardziej zbliżyć się do idealnego wzoru.

Wyobraźmy sobie taką sytuację. System uruchamia proces przez jakiś czas, notuje, ile czasu dostał ten proces poprzez odjęcie tego kwantu czasu od p->wait_runtime. Kiedy ta wartość spadnie wystarczająco nisko (tak, by inny proces miał większą wartość), w drzewie czerwono – czarnym procesów inny proces będzie najbardziej na lewo i wtedy właśnie ten proces rozpocznie swoją pracę. Należy jeszcze dodać, że to nie działa tak do końca. Staramy się uniknąć sytuacji, by np. dwa procesy, co jeden kwant czasu zamieniały się rolami, więc przyjmuje się jakąś ustaloną wartość akceptowalnej różnicy (zmiana kontekstu też kosztuje!!) - wielkość rq->fair_clock śledzi, ile czasu procesora wykorzysta dany proces. Dzięki tej wartości możemy mierzyć oczekiwany czas, jaki powinien zostać przydzielony innym procesom. Wszystkie procesy gotowe (oczekujące na procesor) są posortowane w drzewie czerwono – czarnym. Kluczem w drzewie jest wartość rq->fair_clock - p->wait_runtime. CFS wybiera proces znajdujący się najbardziej w lewo. Po przydzieleniu temu procesowi kolejnych kwantów czasu, zostaje on przesunięty bardziej w prawo, co z kolei daje szanse innym procesom.

Oczywiście implementacja sprawiedliwego przydzielania czasu procesora nie jest jedyną różnicą pomiędzy CFS-em a wcześniejszymi schedulerami.

W przypadku liczenia „sprawiedliwego" przydziału procesora, bierze się pod uwagę czas uśpienia procesora. Oznacza to, że w CFS, procesy, które często zasypiają, dostają więcej czasu procesora niż procesy wykonujące się ciągle. Oczywiście wszystko jest tak wyważone, by zapewnić jak najlepsze działanie: wysoką wydajność i wystarczającą interaktywność.

Kolejną ważną różnicą, jest zmiana podejścia do wyliczania czasu działania procesu. W CFS podstawową jednostką czasu jest nanosekunda. We wcześniejszych schedulerach obliczenia opierano na jiffies czy Hz.

Czas, jaki zostanie przydzielony kolejnemu procesowi, jest wyliczany dynamicznie, ponieważ jesteśmy w stanie odtworzyć informacje o dotychczasowych przydziałach.

CFS-a można w bardzo prosty sposób dostosować stopień interaktywności (zgodnie z sugestiami Linusa). Wartość ziarnistości (granularności) opisuje, jak szybko/często będzie następowała zmiana kontekstu.. Częstsza zmiana kontekstu jest lepsza dla zastosowań domowych (większa interaktywność). Z kolei dla serwerów interaktywność nie jest aż tak ważna, a rzadsza zmiana kontekstu oznacza nieco większą wydajność.

CFS zawiera jeszcze jedną cechę, która wzbudziła zdziwienie wszystkich osób interesujących się tworzeniem jądra: podział na moduły. Ingo określił tą konstrukcję jako „Elastyczną hierarchię modułów schedulera", choć jest to hierarchia bez gałęzi. Jest to lista modułów ustawiona w pewnym porządku. Pierwszy moduł schedulera, który może zajmować się działającymi procesami, decyduje o właściwej kolejności wykonywania zadań. Obecnie występują dwa moduły: CFS i uproszczona wersja schedulera czasu rzeczywistego. Ten moduł pojawia się na pierwszym miejscu na liście, wobec czego każdy proces czasu rzeczywistego zostanie obsłużony przed normalnymi procesami.

6.6 Najważniejsze funkcje

Każdy moduł schedulera zawiera stosunkowo niewielki zbiór zaimplementowanych metod. Zaczynając od funkcji kolejkowych:

- 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);

Kiedy proces rozpoczyna swój kolejny okres – zmienia swój stan na gotowy do wykonania, scheduler (główny moduł) obsługuje tę zmianę poprzez enqueue_task(); (dodanie do kolejki); proces, który przestaje być „runnable", zostaje wyjęty z kolejki (dequeue_task()). Funkcja requeue_task() umieszcza proces za wszystkimi innymi procesami z tym samym priorytetem. Istnieje też kilka funkcji, które służą schedulerowi do śledzenia procesu:

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

Główny scheduler wywołuje funkcję task_new(), kiedy procesy są tworzone. Funkcja task_init() wykonuje wszystkie niezbędne kalkulacje dotyczące priorytetów. Może być wywoływana np., kiedy proces wylicza wartość nice. Funkcja task_tick() jest wywoływana z każdym cyklem zegara, aby odświeżyć obliczenia i być może zmienić kontekst (wywłaszczyć proces).

Główny moduł może zapytać moduł schedulera, kiedy obecnie wykonywany proces powinien być zastąpiony innym procesem poprzez funkcję:

- void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);

W schedulerze CFS, ta funkcja służy do porównania priorytetów z właśnie wykonywanym procesem. Dalej następuje wybór kolejnego procesu. Przy wyborze procesu pod uwagę bierze się również granularność, możliwe jest np. to, by jakiś proces wykonywał się nieco dłużej niż ścisłe zasady sprawiedliwości to określiły. Kiedy główny scheduler wybiera kolejny proces do uruchomienia, używa funkcji:

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

Funkcja pick_next_task() prosi moduł, aby wybrał proces (z procesów obsługiwanych przez dany moduł), który powinien być teraz uruchomiony. Po wywłaszczeniu każdego procesu, odpowiedni moduł zostaje o tym poinformowany (put_prev_task()) Wreszcie, mamy jeszcze dwie funkcje pomocne przy obsłudze wyważania obciążenia procesorów:

- struct task_struct * (*load_balance_start) (struct rq *rq);
- void struct task_struct * (*load_balance_next) (struct rq *rq);

Te funkcje implementują prosty iterator, który może zostać wykorzystany przez schedulery do pracy ze wszystkimi procesami obsługiwanymi przez dany moduł. Podział schedulera na moduły pozwala na stosunkowo łatwe wprowadzanie zmian. W przyszłości taka konstrukcja może być podstawą do stworzenia bardziej zaawansowanych algorytmów szeregowania procesów. Oczywiście, należy pamiętać, że jest to dopiero początek pracy. Co warto zapamiętać? Zarówno RSDL jak i CFS są lepszymi schedulerami niż te z głównej linii jądra, a Con jest pionierem koncepcji równego traktowania wszystkich procesów. Ingo Molnar podczas pracy nad CFS wykorzystał pomysły twórcy RSDL-a.

Rozdział 7

Wyniki testów porównujących działanie planistów z wersji jądra 2.4 oraz 2.6

7.1 Znacząca poprawa działania planisty w wersji 2.6

Na stonie internetowej;

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

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

7.2 Testy przy użyciu hackbench

Przeprowadzono testy za pomocą programu hackbench. Porównywano działanie schelurera z wersji jądra Linux 2.4.18 oraz 2.6.0-test9. Testy przeprowadzono następujących systemach:

- system jednoprocesorowy:

  CPU: 1GHz Pentium III w/ 256k L2 cache

  Memory: 512MB

- system z dwoma procesorami:

  CPU: 850MHz Pentium III w/ 256k L2 cache x2 (Coppermine)

  Memory: 1GB

- system z czterema procesorami:

  CPU: 700 MHz Pentium III w/ 1024k L2 cache x4 (Cascades)

  Memory: 4GB

- system z ośmioma procesorami

  CPU 700 MHz Pentium III w/ 1024k L2 cache x8 (Cascades)

  Memory: 8GB

7.3 Co to jest hackbench?

Hackbench jest benchmarkiem badającym wydajność schedulera. Został stworzony przez Rusty Russell'a. Tworzymy n procesów pisarzy oraz n procesów czytelników. Każdy pisarz pise 100 wiadomości do wszystkich czytelników. Każdy pisarz pisze zatem $100 * n$ wiadomości, a czytelnik odbiera $100 * n$ wiadomości. Kod programu hackbench.c jest dostępny na stronie.

Test tworzy grupę procesów ustawioną przez użytkownika (dla podanego testu: 25). Każdy pisarz pisze 100 wiadomości, a odbiorca je odbiera. Całkowita liczba wysłanych wiadomości wynosi 100 razy liczba procesów.

7.4 Wyniki uruchamianych testów

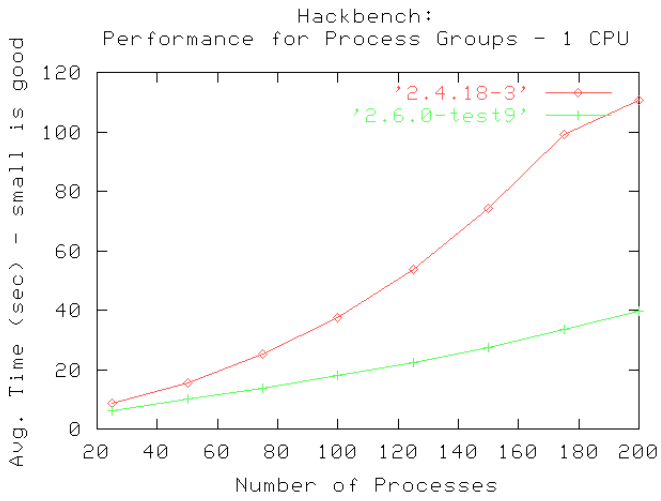
Początkowo każdy test uruchamia się dla 25 procesów, zwiększając ich liczbę o kolejną wielokrotność 25 w każdym kroku, aż do maksymalnej wielkości ustawionej przez testera. Każdy zbiór procesów jest wykonywany 5 razy. Dla każdego zbioru procesów podaje się średni czas do zakończenia testu. Wyniki przedstawia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

7.5 Obserwacja 1: 2.6.0 is bardziej wydajny niż 2.4.18

Pierwszy zbiór testów zastał przeprowadzony z maksymalną liczą 200 pisarzy i czytelników. Testy przeprowadzono dla systemów o różnej liczbie procesorów. Na wykresach przedstawiono średni czas, jaki był potrzebny do zakończenia testu hachbench. Wykresy 7.1, 7.2, 7.3 oraz 7.4 przedstawiają zależność czasu wykonania od liczby procesów.

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

Na wykresach 7.5 oraz 7.6 zestawiono wyniki poprzednich testów, tak aby można było zaobserować wzrost wydajności obu planistów w miarę zwiększania liczby planistów.



Rysunek 7.1: Zależność czasu wykonania od liczby procesów

7.7 Obserwacja 3: 2.6.0 pozwala na bardziej wydajne zużycie zasobów

Następna grupa testów miała na celu określenie maksymalnej liczby procesów, które mogą być wykonywane do momentu w którym zasoby systemu zostaną wyczerpane. Nie czyniono żadnych modyfikacji jądra. Testy były przeprowadzone na domyślnych ustawnieniach. Testy uruchamiano dla coraz większej maksymalnej liczby grup procesów do momentu, w którym system zgłosił błąd związany z brakiem zasobów. Wtedy test był przerywany i przedstawiany był ostatni poprawny wynik.

Wykres przedstawia maksymalną liczbę grup procesów, która poprawnie zakończyła wykonywanie hackbench

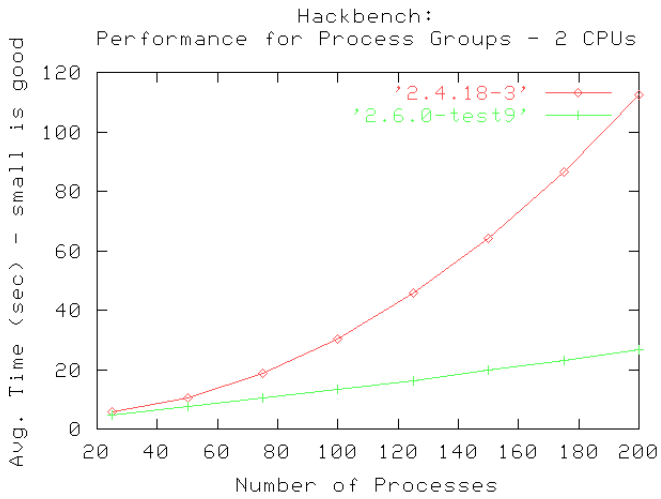
system jednoprocesorowy (wykres 7.7)

Ustawiono maksymalną liczbę procesów na 600. Załamanie dla obu wersji jądra nastąpiło dla 200 procesów.

system dwuprocesorowy (wykres 7.8)

Dla planisty z jądra 2.4.18 maksymalna liczba procesów, które mogą być uruchomione wynosi 225, podczas gdy dla 2.6.0-test9 ta liczba procesów jest większa: wynosi 350

system czteroprocesorowy (wykres 7.9)



Rysunek 7.2: Zależność czasu wykonania od liczby procesów

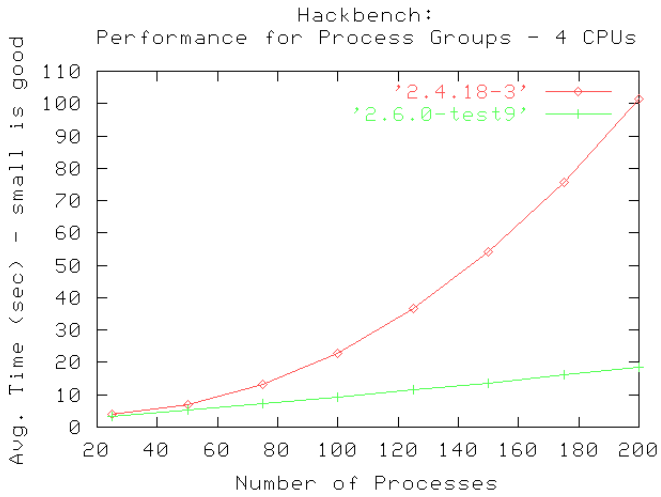
Maksymalna liczba procesów dla 2.4.18 wynosi 225 (jak w przypadku systemu dwuprocesoweg), podczas gdy liczba procesów, które mogą być uruchomione w systemie czteroprocesorowym z wersją planisty z 2.6.0-test9, wzrasta do 525.

system ośmioprocesorowy (wykres 7.10)

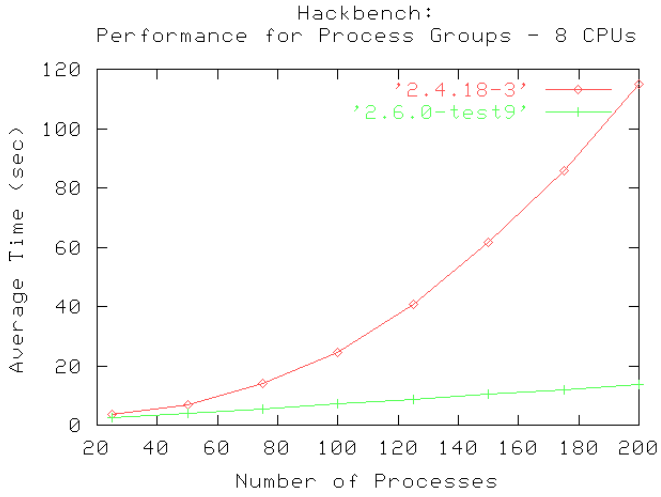
Dla planisty z wersji jądra 2.6.0-test9 inny powód załamania, otrzymano komunikat o następującym błędzi:

SENDER: write (error: No buffer space available)

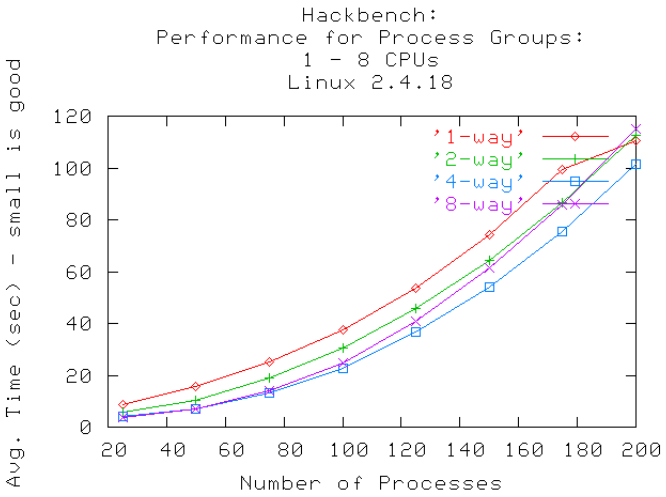
Dla systemu 8 procesorowego zaalokowano bowiem mniejszy bufor, który był wykorzystywany do przesyłania wiadomości i to doporowadziło do jego przepełnienia.



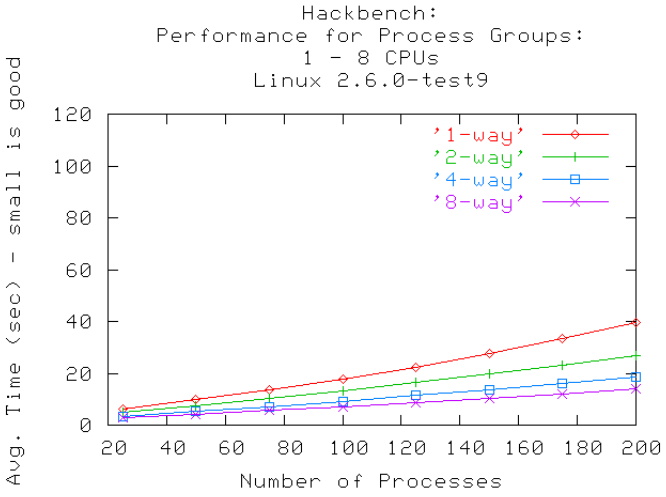
Rysunek 7.3: Zależność czasu wykonania od liczby procesów



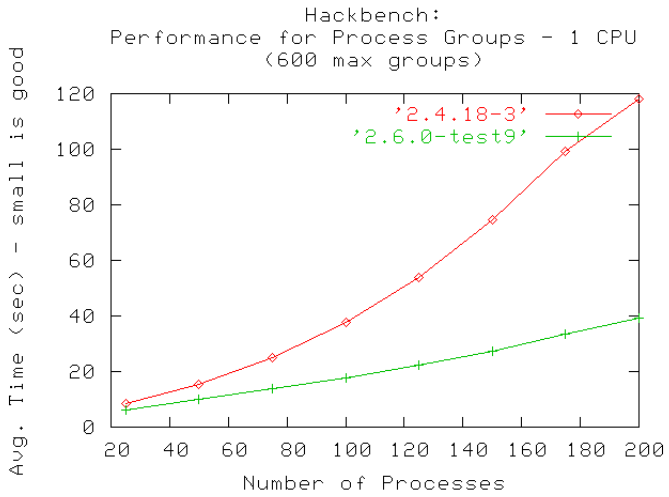
Rysunek 7.4: Zależność czasu wykonania od liczby procesów



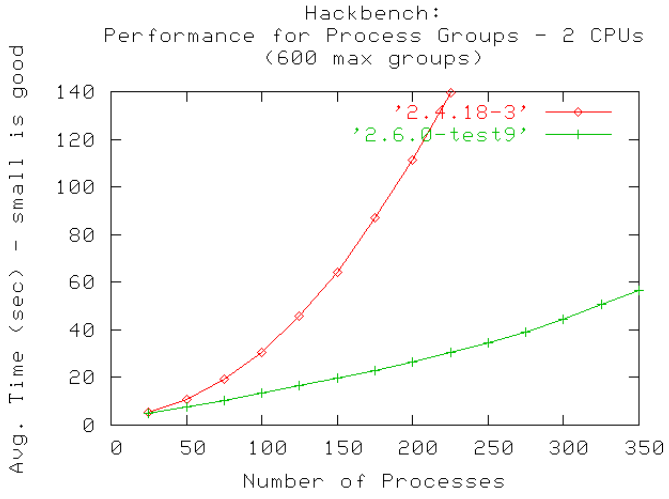
Rysunek 7.5: Skalowalność na systemy wieloprocesorowe



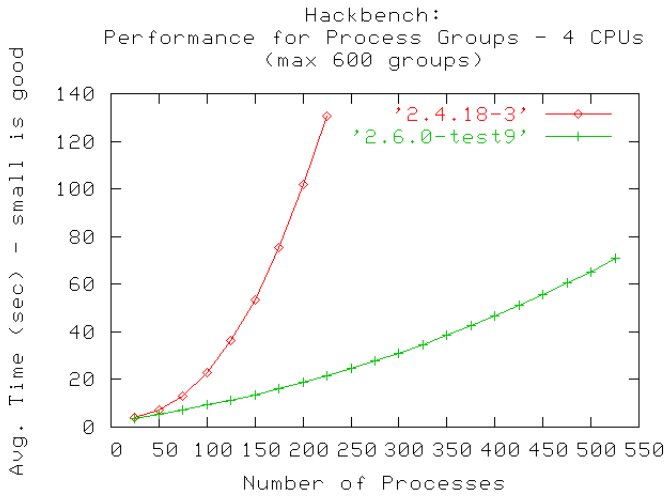
Rysunek 7.6: Skalowalność na systemy wieloprocesorowe



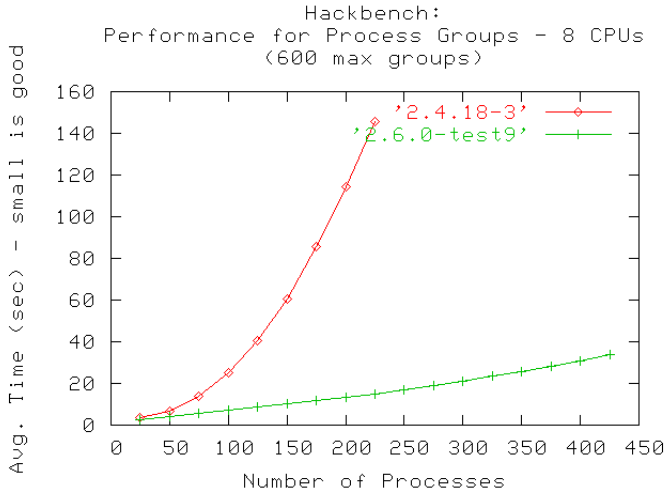
Rysunek 7.7: Zużycie zasobów



Rysunek 7.8: Zużycie zasobów



Rysunek 7.9: Zużycie zasobów



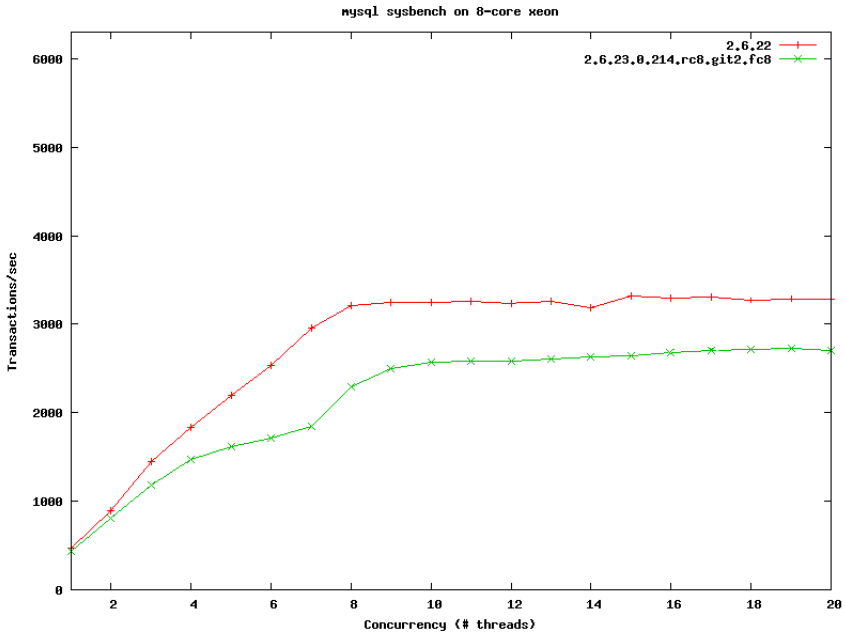
Rysunek 7.10: Zużycie zasobów

Rozdział 8

Wyniki testów dla CFS

Dyskusja o tym, który z planistów: RSDL czy CFS jest lepszy pojawiała się na wielu listach dyskusyjnych użytkowników linuksa. Między innymi na stronie internetowej: www.kerneltrap.org przedstawiano różne wyniki świadczące o wyższości jednego schedulera nad drugim.

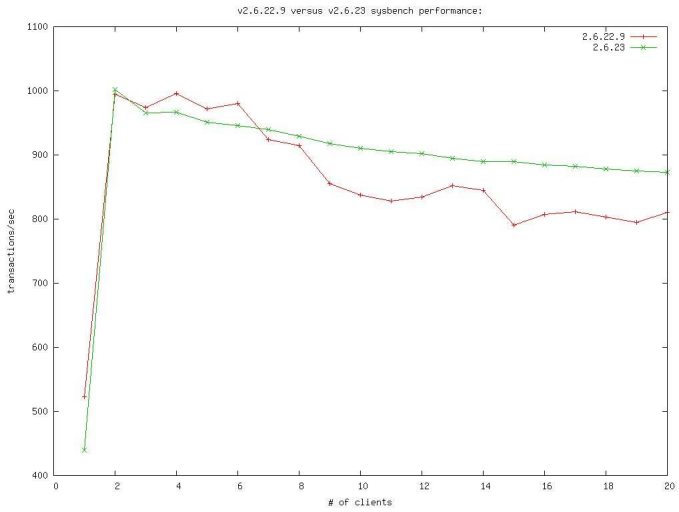
Wyniki testów na serwerze MySQL Nicholas Miel przedstawia wynik przeprowadzonych przez siebie testów. Na wykresie możemy obserwować zależność pomiędzy liczbą wątków a liczbą transakcji na sekundę w przygotowanej przez autora bazie (wykres 8).



Rysunek 8.1: Wyniki testów na serwerze MySQL

Jednak brakuje dokładnej specyfikacji przeprowadzonych testów (konfiguracja systemu, sprzęt komputetowy). W odpowiedzi na tak przedstawioną tezę

Ingo Molnar (twórca CFS) przeprowadził swoje własne testy. Ich wyniki znacząco odbiegają od poprzednich (wykres 8):

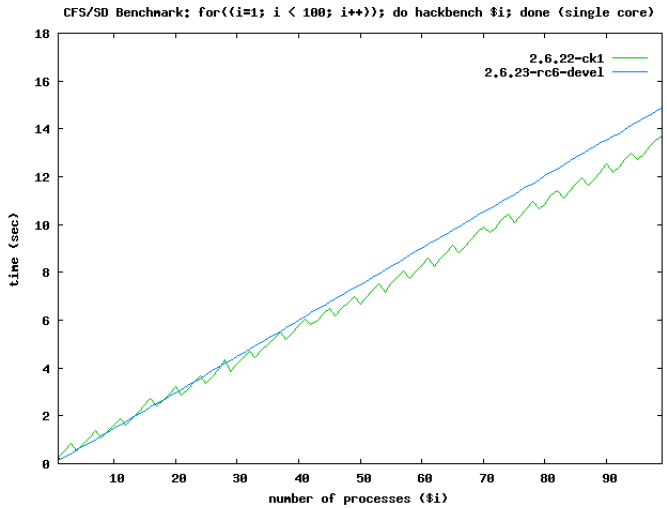


Rysunek 8.2: Wyniki testów na serwerze MySQL

Autor szczegółowo przedstawił konfigurację systemu (załączony plik konfiguracyjny). Testy zostały przeprowadzone na maszynie: Core2Duo 1.83 GHz. Test przeprowadzono na bazie MySQL maksymalnie obciążając system (duża liczba klientów oraz mocno obciążany serwer). Na podstawie testu można wywnioskować przewagę schedulera z wersji: 2.6.23 (CFS) nad 2.6.22.9 (RSDL). Widać co prawda lepsze rezultaty osiągane przez RSDL dla pewnego zakresu liczby klientów, ale zakres ten nie jest zbyt duży. Widać także większą stabilność wyników w przypadku CFS.

Testy przy użyciu hackbench Obserwując wykres 8.3 Igno Molnar zwrócił uwagę na fakt, iż niebieska linia (reperzentująca CFS) ma znacznie mniejsze wachania w stosunku do lini zielonej (reperezentującej SD). Widać jednak iż SD jest lepszy od CFS. Ingo szuka przyczyny tej sytuacji w tym, iż pomiędzy jądrem 2.6.22-ck1 oraz 2.6.23-cfs-devel zmieniło się wiele innych rzeczy i to któraś z nich może mieć wpływ na wynik. Ingo przypuszcza, iż może to mieć związek z sched_clock(). W pomocniczych testach skonfigurował system (m.in.sched_clock()), aby ustawienia były bardziej zbliżone do domyślnych ustawień w jądrze 2.6.22-ck1. W ten sposób osiągnięto lepsze rezultaty. Stąd wyciągnął wniosek, iż różnice w wydajności pomiędzy poszczególnymi jądrami są spowodowane bardziej precyzyjnym (ale wolniejszym) sched_clock() wprowadzonym do wersji 2.6.23-cfs-devel oraz początkowym opóźnieniem swieżo tworzonych procesów. Kiedy został zapytany, czy poprzednia wersja sched_clock() zostanie przywrócona w nowszych wersjach, Ingo odparł, iż dopóki różnice w wydajności nie będą zbyt duże, lepszym rozwiązaniem jest nowsza wersja, z uwagi na precyzję wykonywanych działań.

Testy mierzące czas przełączania kontekstu lat_ctx
Mierzy czas przełączania kontekstu dla podanej liczby procesów i wielkości



Rysunek 8.3: hacbench

czasu. Tworzy n proxesów i łączymy je za pomocą łącz nienazwanych (pipe) w pierścień. Processy przekaziją sobie żeton. Proces odbiera żeton i wykonuje pewnien kod (parametr testu), następnie przekazuje żeton następnemu procesowi w pierścieniu. (wykres 8.4)

CFS a gry 3D Niektóre zgłaszane obawy na temat CFS dotyczyły tego, iż może on radzić sobie gorzej z grami 3D niż SD. Ingo Molnar stwiedził, iż ludzie regularnie testujący płynność gier 3D oceniają wydajność CFS jako wystarczającą. Te oceny Ingo popiera własnymi odczuciami, twierdząc, iż CFS i SD są praktycznie tak samo wydajne, jeśli chodzi o płynność gier 3D. Podkreśla, iż powody pojawiających się wcześniej informacji o spadku wydajności CFS, zostały poprawione. Przedstawił wyniki testów, świadczące o tym, iż poprawiony CFS działa nie gorzej lub nawet lepiej niż SD.

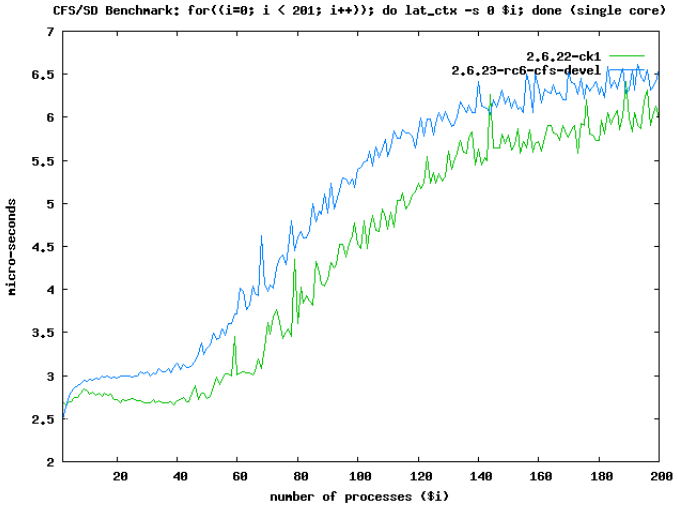
Porównanie: v2.6.22-ck1 (SD) v2.6.22-cfsv19 (CFS) Gra: Quake 3 Arena Demo under Wine (0.9.41) (wykres 8.5)

Zachowanie Quake3 uruchomionego pod wine'em w przypadku SD

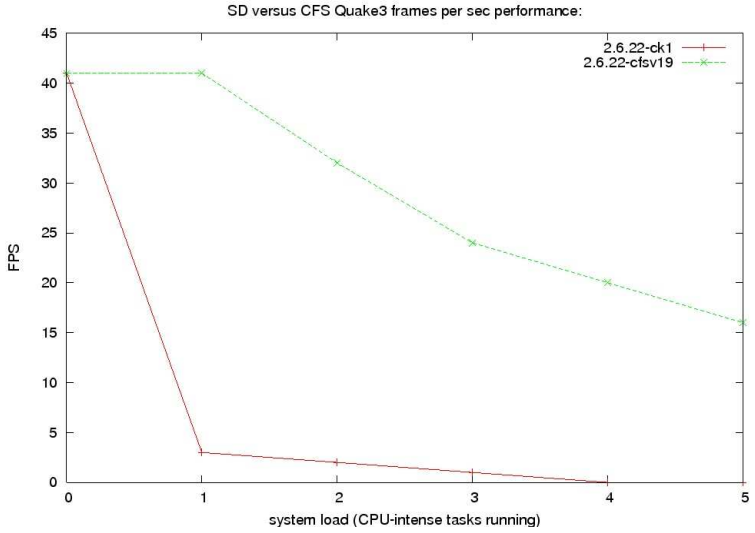
Ociążając CPU płynność gry spada drastycznie. Już przy jednym dodatkowym procesie, który potencjalnie może maksymalnie obciążyć procesor, płynna gra nie jest możliwa.

Zachowanie Quake3 uruchomionego pod wine'em w przypadku CFS

Liczba klatek na sekundę spada delikatnie wraz z większającym się obciążeniem. Liczba klatek na sekundę (16 fps) jest całkiem akceptowalna nawet przy 500% obciążeniu procesora, a grafika wygląda całkiem dobrze. Liczba klatek na sekundę w zależności od liczby procesów z grubsza jest odwrotnie proporcjonalna.



Rysunek 8.4: lat_ctx



Rysunek 8.5: Zależność między liczba zadań w tle a liczbą klatek na sekundę