Linux Scheduler

Systemy operacyjne sa projektowane pod kątem optymalnej pracy na dostępnych zasobach. Jest to osiągane poprzez wirtualizację procesora oraz pamięci. Wielozadaniowość polega na przypisaniu do każdego zadania osobnego virtualnego procesora przy czym zadania nie są świadome braku wyłączności na fizyczną jednostkę.

Wirtualizacja pamięci polega na przypisania zadaniom osobnej wirtualnej przestrzeni adresowej, która jest później mapowana na prawdziwą pamięć systemu.

Wirtualizacja procesora jest uzyskana dzięki podziałowi czasu pracy procesora na poszczególne zadania. Na wykonanie procesu zostaje przeznaczony określony przedział czasu. Algorytmem stosowanym do zarządzania wykonywanymi zadaniami jest Scheduler – planista.

Planista jest częścią każdego systemu operacyjnego. Algorytm wybiera zadania z najwyższym priorytetem (najczęściej programy wymagające interakcji). Jednocześnie scheduler musi zapobiegać zbyt długiemu oczekiwaniu na wykonanie przez procesy „proces starvation”. Oznacza to, że procesy o niskim prioryciecie muszą mieć możliwość wykonania niezależnie od ilości konkurujących procesów z priorytetem wysokim. Dodatkowo procesy powinny sprawiać wrażenie wykonywanych równocześnie.

W przypadku procesów interaktywnych (GUI) idealny scheduler przeznaczyłby każdemu procesowi bardzo mały przedział czasu pracy na CPU i często je zmieniał. Czas oczekiwania na odpowiedź powinien być nie zauważalny dla człowieka (50 and 150ms).

Procesy nie interakrywne wymagają odwrotnego mechanizmu. Operacja przeskakiwania pomiędzy procesorami jest relatywnie kosztowna. W takiej sytuacji korzystniejszym scenariuszem jest dłuższe wykonywanie pojedyńczych zadań.

# O(1) Scheduler

Jądro 2.6 przyniosło wiele zmian odnośnie schedulera. Nazwa została wybrana ze względu na to że algorytm Schedulera wymaga stałego czasu na podjęcie decyzji, niezależnie od ilości zadań. Algorytm wykorzystany przez O(1) scheduler polega na aktywnych i nieaktywnych tablicach procesów. Każdy proces otrzymuje określoną cząstkę czasu po której jest zastępowany i przenoszony do nieaktywnej tablicy. Po wykorzystaniu swoich przydziałów czasu przez wszystkie procesy z tablicy, następuje zamiana tablic. Przy zamianie tablica aktywna staje się nieaktywną, a inna nieaktywna aktywną.

Główną wadą tego algorytmu jest złożona heurystyka wykorzystana do określania czy proces jest interaktywny czy nie. Algorytm stara się zidentyfikować procesy interaktywne poprzez analizowanie średniego czasu uśpienia. Procesy śpiące dłużej najprowdopodobniej oczekują na interakcję użytkownika. Takie procesy oznaczane są jako interaktywne i otrzymują wyższy priorytet. Obliczenia te są skomplikowane i dają możliwość do błędów spowodowanych nieinteraktywnym zachowaniemm interaktywnych procesów.

Rozwiązaniem tego problemu jest zastosowanie w nowszych wersajch jądra systemu algorytmu CFS.

# Completely Fair Scheduler

Idealnym, precyzyjnym procesor jest rozwiązaniem sprzętowym które może wykonywać wiele procesów równolegle, dając każdemu procesowi równą część mocy obliczeniowej.

Rysunek 1

Ze względu na to, że taki idealny processor istnieje pozostaje emulowanie jego pracy poprzez oprogramowanie. W prawdziwym procesorze jedynie jedno zadanie może być wykonywane w danej chwili. W takiej sytuacji jeden proces dostaje 100% udziału a inne 0%.

Rysunek 2

CFS dąży do wyeliminowania tej niesprawiedliowści. Trzyma w pamięci informację o niesprawiedliwym balansowaniu dla każdego procesu. W trakcie oczekiwania na procesor scheduler określa ile czasu byłby on wykonywany na idealnym procesorze. Wartość ta jest reprezentowana przez: „wait\_runtime”. Jest to czas (w nanosekundach) wykorzystywany do ustalenia ile dany proces powinien dostać przydziału procesora zanim zostanie zamieniony. Miarą wzrostu sprawiedliwego zegara jest czas standardowy podzielony przez ilość oczekujących procesów.

Do wykonania wybierany jest proces z najdłuższym czasem oczekiwania. Wykorzystując tę zasadę CFS dąży do stanu w którym każdy proces posiada zerowy czas oczekiwania. Każdy proces posiada równy przydział CPU, co każdy idealny procesor by zapewniał.

Mechanizmy zapweniające emulowanie idealnego procesora:

1. Mechanizm do obliczania sprawiedliwego przydziału dla każdego procesu. Jest to oddzielny zegar działający z prędkością standardowego czsu podzielonego przez ilość oczekujących procesów
2. Mechanizm przechowujący informację o czasie oczekiwania przez proces na wykonanie.

# Red-Black Tree

Jest to typ drzewa binarnego samobalanjującego. Zaletami tej struktury są: dobry najgorszy czas wykonania 0(lon n) (n to ilość elementów w drzewie)oraz wydajność w praktyce.

In red-black trees, the leaf nodes are not relevant and do not contain data. These leaves need not be explicit in computer memory—a null child pointer can encode the fact that this child is a leaf—but it simplifies some algorithms for operating on red-black trees if the leaves really are explicit nodes. To save memory, sometimes a single sentinel node performs the role of all leaf nodes; all references from internal nodes to leaf nodes then point to the sentinel node. (Source: Wikipedia.)

CFS uses the fair clock and wait runtime to keep all the runnable tasks sorted by the rq->fair\_clock - p->wait\_runtime key in the rbtree (see the Red-Black Tree sidebar). So, the leftmost task in the tree is the one with the “gravest CPU need”, and CFS picks the leftmost task and sticks to it. As the system progresses forward, newly awakened tasks are put into the tree farther and farther to the right—slowly but surely giving every task a chance to become the leftmost task and, thus, get on the CPU within a deterministic amount of time.

Because of this simple design, CFS no longer uses active and expired arrays and dispensed with sophisticated heuristics to mark tasks as interactive versus non-interactive.

CFS implements priorities by using weighted tasks—each task is assigned a weight based on its static priority. So, while running, the task with lower weight (lower-priority) will see time elapse at a faster rate than that of a higher-priority task. This means its wait\_runtime will exhaust more quickly than that of a higher-priority task, so lower-priority tasks will get less CPU time compared to higher-priority tasks.

Kernel 2.6.24

CFS has been modified a bit further in 2.6.24. Although the basic concept of fairness remains, a few implementation details have changed. Now, instead of chasing the global fair clock (rq->fair\_clock), tasks chase each other. A clock per task, vruntime, is introduced, and an approximated average is used to initialize this clock for new tasks. Each task tracks its runtime and is queued in the RBTree using this parameter. So, the task that has run least (the one that has the gravest CPU need) is the leftmost node of the RBTree and will be picked up by the scheduler. See Resources for more details about this implementation.

In kernel 2.6.24, another major addition to CFS is group scheduling. Plain CFS tries to be fair to all the tasks running in the system. For example, let's say there is a total of 25 runnable processes in the system. CFS tries to be fair by allocating 4% of the CPU to all of them. However, let's say that out of these 25 processes, 20 belong to user A while 5 belong to user B. User B is at an inherent disadvantage, as A is getting more CPU power than B. Group scheduling tries to eliminate this problem. It first tries to be fair to a group and then to individual tasks within that group. So CFS, with group scheduling enabled, will allocate 50% of the CPU to each user A and B. The allocated 50% share of A will be divided fairly among A's 20 tasks, while the other 50% of the CPU time will be distributed fairly among B's 5 tasks.