Wykład trzeci

Szeregowanie procesów w Linuksie (od wersji 2.6.0 do wersji 2.6.22)

Istnieją dwa typy systemów wielozadaniowych: systemy wielozadaniowe z kooperacją (bez wywłaszczania) i systemy wielozadaniowe z wywłaszczaniem. W systemach pierwszego typu proces zawsze dobrowolnie zrzeka się procesora i oddaje sterowanie do systemu operacyjnego. Drugi typ systemów kontroluje wykorzystanie procesora przez realizowane zadania i może tymczasowo przerywać ich działanie, odbierając im procesor. Zapobiega to sytuacji, w której pojedynczy proces mógłby zmonopolizować dostęp do procesora. Linux jak większość współczesnych systemów operacyjnych realizuje wielozadaniowość z wywłaszczaniem.

Schemat szeregowania jaki zastosowano w tym systemie opiera się na schemacie wielopoziomowych kolejek ze sprzężeniem zwrotnym. Linux, zgodnie ze standardem POSIX, wyróżnia trzy grupy procesów: SCHED_OTHER – zwykłe procesy, które mogą być wykonywane w imieniu wszystkich użytkowników oraz SCHED_FIFO i SCHED_RR, które są procesami czasu rzeczywistego wykonywanymi z uprawnieniami użytkowników uprzywilejowanych. Najpierw zostanie opisane szeregowanie dla pierwszej grupy. Podobnie, jak w innych systemach uniksowych faworyzowane są procesy ograniczone wejściem-wyjściem, gdyż do tej kategorii należą procesy interaktywne. Planista nie może jednak ignorować procesów ograniczonych procesorem i nie powinien prowadzić do ich zagłodzenia. Kryterium decydującym o kolejności wykonania procesów jest ich priorytet. W Linuksie zadania o wyższym priorytecie trafiają do kolejek o dłuższym kwancie czasu, podczas gdy procesy o niższym priorytecie umieszczane są w kolejkach o krótszym kwancie. W obrębie pojedynczej kolejki procesy są szeregowane według algorytmu rotacyjnego, który każdemu z nich przydziela jednakowy kwant czasu. Ponadto zadania o wyższym priorytecie są uruchamiane przed procesami o niższym priorytecie. Wartość priorytetu zadania jest ustalana zarówno przez użytkownika, jak i może być modyfikowana przez system operacyjny. Może ona ulegać zmianie podczas istnienia procesu w systemie, a więc jest to priorytet dynamiczny. Każdy nowy proces otrzymuje domyślny priorytet, który z czasem może ulec zmianie, zależnie od tego, czy ów proces częściej wykonuje operacje wejścia – wyjścia, czy częściej korzysta z procesora. Wartości priorytetów rozdzielone są na dwa zakresy: pierwszy to *poziom uprzejmości (ang. nice)*, do którego należą wartości od -20 do 19 (im mniejsza wartość, tym większy priorytet), drugi zakres, to wartości od 1 do 99 (im większa wartość, tym wyższy priorytet). Ten ostatni zakres to priorytety dla zadań *czasu rzeczywistego*. Linux nie jest rygorystycznym systemem czasu rzeczywistego, ale istnieją odmiany jego jądra, które spełniają wymogi nakładane na takie systemy. Procesy czasu rzeczywistego podzielone są na dwie grupy. Te, które należą do pierwszej z nich (SCHED_FIFO) szeregowane są według algorytmu FCFS, natomiast te należące do drugiej (SCHED_RR) z użyciem algorytmu rotacyjnego. Dla procesów czasu rzeczywistego priorytet jest priorytetem statycznym. Zwykłe procesy w Uniksie otrzymują domyślny kwant czasu większy niż 20 ms (w Linuksie jest to 100 ms), który w zależności od ich „zachowania" może się zwiększać lub zmniejszać. Proces nie musi od razu wykorzystywać całego dostępnego mu czasu, ale jeśli wyczerpie cały kwant to ulega przeterminowaniu i nie może zostać ponownie uruchomiony do czasu, aż pozostałe procesy nie wyczerpią swoich kwantów czasu i nie zostanie uruchomiona procedura ich przeliczania. Proces, który wyczerpie swój kwant czasu podlega jednocześnie wywłaszczaniu. Może on również zostać wywłaszczony, jeśli w systemie pojawi się nowy proces o wyższym priorytecie. Mechanizm szeregujący w jądrze 2.6 Linuksa posiada kilka godnych uwagi cech:

1. implementuje szeregowanie w czasie O(1),
2. implementuje idealne skalowanie SMP,
3. implementuje powiązania wątków z procesorami w trybie SMP,
4. daje dobrą wydajność procesów interaktywnych,
5. równomiernie rozkłada czas procesora,
6. stosuje optymalizację dla często spotykanego przypadku, kiedy w systemie jest kilka procesów gotowych do wykonania.

Planista wiąże z każdym procesorem w systemie kolejkę procesów gotowych do wykonania. Jest to struktura danych o nazwie *struct runqueue*, która została zdefiniowana w pliku *kernel/sched.c* Dodatkowo zostały zdefiniowane odpowiednie makrodefinicje pozwalające na łatwy dostęp do tej struktury. Dostęp do kolejek procesów gotowych jest synchronizowany przy pomocy semaforów z aktywnych oczekiwaniem, zwanych krócej ryglami pętlowymi. Jeśli istnieje konieczność modyfikacji kilku kolejek równocześnie, to muszą one być zajmowane według ściśle określonej kolejności. Zapobiega to powstawaniu zakleszczeń.

Każda z kolejek procesów gotowych zawiera dwa wskaźniki na tablice priorytetów: aktywną i przeterminowaną (*struct prio_array*). W strukturze opisującej taką tablicę znajduje się pole, określające liczbę gotowych do wykonania procesów w tej tablicy oraz tablica wskaźników do list procesów. Indeksy w tej tablicy są wartościami priorytetów zadań. Priorytety czasu rzeczywistego są odwracane, czyli w zakresie od 1 do 99, im większa wartość, tym niższy priorytet, a priorytety zwykłych procesów są przeliczne na zakres od 100 do 139 (tak samo jak poprzednio). W obrębie list procesy są szeregowane według algorytmu rotacyjnego, z wyjątkiem procesów czasu rzeczywistego należących do grupy SCHED_FIFO. Dodatkowo, każda tablica priorytetów wyposażona jest w mapę bitową (jeszcze jedno pole struktury *struct prio_array*), której poszczególne bity określają, czy są w tablicy zadania o danym priorytecie do wykonania. Aby znaleźć zadanie o najwyższym priorytecie gotowe do wykonania należy jedynie znaleźć pozycję pierwszego ustawionego bitu w tej mapie.

Kwanty czasu zadań są przeliczane zaraz po ich wyczerpaniu przez zadanie i zanim zadanie trafi do tablicy przeterminowanej. Wymiana „starych" priorytetów na „nowe" sprowadza się wyłącznie do wymiany wskaźników na tablicę aktywną i przeterminowaną.

Wyboru następnego do wykonania zadania dokonuje funkcja *schedule*(), wywoływana zawsze wtedy, kiedy trzeba zawiesić wątek jądra lub wywłaszczyć zdanie. Oprócz wyboru zadania do uruchomienia wywołuje ona także funkcję *context_switch*(), która odpowiedzialna jest za przełączenie kontekstu. Kod funkcji *schedule*() jest bardzo prosty i czas jego wykonania nie zależy od liczby procesów w systemie. Bardziej krytyczną pod względem czasu działania jest wywoływana przez *schedule*() funkcja *context_switch*(), której implementacja jest zależna od architektury platformy sprzętowej na której działa system.

Priorytet każdego zwykłego zadania jest ustalany na podstawie priorytetu statycznego, jakim jest poziom uprzejmości oraz na podstawie stopnia interaktywności procesu. Interaktywność jest wyznaczana heurystycznie jako stosunek długości okresów oczekiwania i aktywności. W zależności od tak wyliczonego stopnia interaktywności priorytet dynamiczny jest zwiększany lub zmniejszany o 5. Każdy nowo powstały proces w systemie otrzymuje połowę kwantu czasu, jaki w chwili jego tworzenia miał proces macierzysty. Ten czas jest także odliczany od kwantu rodzica. Po wyczerpaniu kwant jest na nowo wyliczany dla obydwu procesów. Zadania o najwyższym priorytecie otrzymują kwanty o wielkości 200 ms, zdania o najniższym priorytecie kwanty o wartości około 10 ms. Podstawowa długość kwantu czasu wynosi 100 ms. Jądro Linuksa promuje zadania o dużym stopniu interaktywności. Takie zadania po wykorzystaniu swojego kwantu czasu nie trafiają od razu do tablicy przeterminowanej, ale otrzymują drugą szansę i są umieszczane w tablicy aktywnej, na końcu tej samej listy, na której były poprzednio (dostają ten sam kwant czasu, co przed wykonaniem). Aby nie doszło do głodzenia procesów jądro wykonuje makrodefinicję *EXPIRED_STARVING()*, która sprawdza, czy w tablicy przeterminowanej są procesy, które zbyt długo czekają na realizację.

Proces może zostać zawieszony w oczekiwaniu na jakieś zdarzenie, np.: realizację operacji wejścia–wyjścia lub w oczekiwaniu na sygnał. Taki proces nie może znajdować się w kolejce procesów gotowych do wykonania. Najczęściej dochodzi do tego po wywołaniu przez zadanie któregoś z wywołań systemowych, np.: *read()*, co powoduje dodanie tego zadania do kolejki procesów oczekujących (zdefiniowanej strukturą o typie *wait_queue_head_t*) i wywołanie funkcji *schedule()*, celem wyznaczenia innego procesu do wykonania. Budzenie zadań będących w określonej kolejce oczekiwania odbywa się za pomocą wywołania np. funkcji *wake_up()*. Jeśli

obudzone zadanie ma wyższy priorytet niż zadanie bieżąco wykonywane, ustawiany jest znacznik *need_resched*.

Wywłaszczenie procesu następuje wtedy, kiedy jest ustawiony znacznik *need_resched*, który ze względu na szybkość dostępu jest przechowywany w polu *flags* struktury *thread_info* zadania. Celem ustalenia nowego procesu do wykonania jądro wywołuje funkcję *schedule()*, która z kolej wywołuje *context_switch()*. Ta ostatnia dokonuje zmiany kontekstu, wykonując zamianę odwzorowania pamięci wirtualnej (za pomocą wywołania funkcji *switch_mm()*) oraz zmieniając stan procesora dla nowego zadania (za pomocą wywołania funkcji *switch_to()*) zachowując przy tym stos i wartości rejestrów dla poprzedniego zadania. Wywłaszczenie procesu użytkownika może zajść w ramach powrotu do przestrzeni użytkownika z wywołania systemowego, bądź z procedury obsługi przerwania. Wywłaszczenie wątku jądra natomiast może nastąpić w ramach powrotu do przestrzeni jądra z procedury obsługi przerwania, kiedy zostanie dla niego wyzerowany licznik *preempt_count*, kiedy wątek wykonywany w przestrzeni jądra jawnie wywoła funkcję *schedule()* lub kiedy wątek jądra ulegnie zablokowaniu (wejdzie w stan oczekiwania).

W systemach wieloprocesorowych zadania mogą być kojarzone z poszczególnymi procesorami, ale czasem zachodzi potrzeba zrównoważenia pracy systemu, wówczas część zadań z kolejki procesów gotowych procesora może zostać przeniesiona do kolejek innych procesorów lub odwrotnie. Mogą być dwie przyczyny takiego zdarzenia. Pierwsza zachodzi wtedy, kiedy w kolejce któregoś z procesorów nie ma żadnych zdań, wówczas mogą one być przeniesione z kolejek innych procesorów. Druga to wywołanie *load_balance()* za pomocą przerwania zegarowego. W tym przypadku zadanie równoważenia obciążenia jest bardziej skomplikowane. W skrócie polega ono na znalezieniu najbardziej obciążonej kolejki (ponad 25% obciążenia pozostałych kolejek w systemie) i rozłożeniu tego obciążenia na pozostałe procesory.

Istnieje szereg funkcji dostępnych dla aplikacji użytkownika, które mogą wpływać na sposób szeregowania zadań, oto niektóre z nich:

1. *nice()* - określa poziom uprzejmości procesu,
2. *sched_setscheduler()* - określa strategię szeregowania procesów,
3. *sched_getscheduler()* - pobiera strategię szeregowania procesów,
4. *sched_setparam()* - ustala parametry szeregowania procesu zgodne z określoną dla niego strategią,
5. *sched_getparam()* - pobiera parametry szeregowania procesu jakie określa jego strategia szeregowania,
6. *sched_get_priority_max()* - pobiera maksymalny priorytet dla określonej strategii szeregowania,
7. *sched_get_priority_min()* - pobiera minimalny priorytet dla określonej strategii szeregowania,
8. *sched_rr_get_interval()* - określa wartość kwantu czasu procesu z grupy SCHED_RR,
9. *sched_setaffinity()* - kojarzy proces/wątek z procesorem (procesorami),
10. *sched_getaffinity()* - pobiera maskę procesorów, na których zadanie może się wykonywać,
11. *sched_yield()* - pozwala procesowi dobrowolnie zrzec się czasu procesora.

Ostatnie wywołanie jest w jądrach serii 2.6 (do 2.6.22) zaimplementowane inaczej niż miało to miejsce w poprzednich wersjach. Dokonuje ono umieszczenia zrzekającego się zdania nie na końcu listy, ale w tablicy przeterminowanej. Z tego wywołania bezpośrednio powinny korzystać procesy użytkownika, natomiast wątki jądra powinno korzystać z *yield()*.

W wersji jądra 2.6.23 planista O(1) został zastąpiony planistą CFS (ang. *Completely Fair Scheduler*), który realizuje strategię sprawiedliwego szeregowania. Działanie tego planisty zostanie opisane na następnym wykładzie.