Systemy Operacyjne 2

Szeregowanie procesów, część 2

Arkadiusz Chrobot

Katedra Systemów Informatycznych

21 marca 2024

Plan

Wady planisty O(1)

Planista O(1) posiada pewne wady, wynikające z tego, że bazuje na schemacie szeregowania opartym na wielopoziomowych kolejkach ze sprzężeniem zwrotnym:

- kwanty czasu związane z poziomami uprzejmości są niezmienne, co oznacza, że jeśli w systemie są jedynie np. dwa procesy gotowe do wykonania, ale o bardzo niskim priorytecie, to mogą one nieprzerwanie korzystać z procesora tylko przez bardzo krótki czas i są często wywłaszczane,
- ziarnistość kwantów czasu może być zbyt mała: długość kwantów czasu przydzielonych dwóm procesom o wysokich priorytetach (np. -20 i -19) jest zbliżona, ale w przypadku procesów o niskich priorytetach (np. 18 i 19) znacząco się różni,
- pomiar zużycia kwantu czasu nie jest precyzyjny,
- heurystyki używane do oceny poziomu interaktywności procesów są podatne na manipulacje, co pozwala procesom uzyskiwać dłuższe kwanty czasu, niż w rzeczywistości potrzebują.

Wady planisty O(1)

Wpływ niektórych z tych wad udało się złagodzić w planiście O(1), ale całkowite ich usunięcie okazało się niemożliwe. To dlatego programiści jądra Linuksa postanowili przebudować mechanizm szeregowania w wersji 2.6.23.

Klasy szeregowania

Jednym z najważniejszych elementów nowego mechanizmu są struktury typu `struct sched_class` nazywane *klasami szeregowania* i reprezentujące politykę szeregowania dla określonej grupy procesów. Każda taka struktura zawiera wskaźniki na funkcje, które realizują, zgodnie z daną polityką, następujące operacje:

`enqueue_task()` dodaje proces do kolejki procesów gotowych,

`dequeue_task()` usuwa proces z kolejki procesów gotowych,

`yield_task()` pozwala procesowi zrzec się CPU,

`check_preempt_curr()` sprawdza, czy bieżący proces powinien być wywłaszczony na rzecz procesu, który właśnie został wybudzony,

`pick_next_task()` wybiera następny proces do wykonania,

`put_prev_task()` funkcja związana ze zmianą kontekstu,

`set_curr_task()` funkcja wywoływana, kiedy polityka szeregowania bieżącego procesu ulega zmianie,

`new_task()` funkcja odpowiedzialna za przydział procesora dla nowego procesu.

Klasy szeregowania

Nowy mechanizm szeregowania udostępnia następujące polityki:

SCHED_FIFO procesy czasu rzeczywistego szeregowane algorytmem FCFS,

SCHED_RR procesy czasu rzeczywistego szeregowane algorytmem rotacyjnym,

SCHED_DEADLINE procesy czasu rzeczywistego szeregowane algorytmem EDF (ang. *Earliest Deadline First*); ta polityka została dodana w wersji 3.14 jądra,

SCHED_NORMAL zwykłe procesy szeregowane z użyciem algorytmu CFS; ta polityka odpowiada polityce SCHED_OTHER w standardzie POSIX,

SCHED_BATCH polityka dla procesów o niskim priorytecie, zorientowanych na przetwarzanie; są one szeregowane przez planistę CFS,

SCHED_IDLE polityka dla procesów o niskim priorytecie, które są wykonywane, gdy nie ma innych procesów do wykonania; są one również szeregowane przez planistę CFS.

Klasy szeregowania

Klasy szeregowania są połączone w listę, zaczynającą się od klas dla procesów o najwyższym priorytecie (czasu rzeczywistego), a kończącą klasami związanymi z procesami o niskim priorytecie (procesy wsadowe i bezczynności). Funkcja `schedule()` iteruje po tej liście, wywołując funkcję (metodę) `pick_next_task()` dla każdej z tych klas. Ta metoda, która zwróci wartość różną od `NULL` określa kolejny proces, któremu zostanie przydzielony procesor. Warto odnotować, że klasy szeregowania są jednym z kilku przykładów zastosowania w kodzie jądra paradygmatu programowania zorientowanego obiektowo, choć jest ono napisane w języku C, nie zaś w C++.

Jednostki szeregowania

W wersji 2.6.23 jądra dodano także inną ważną strukturę, typu `struct sched_entity`, która pozwala jądru szeregować nie tylko pojedyncze procesy, ale również ich grupy. To co zatem podlega szeregowaniu w nowym mechanizmie nazywa się ogólnie *jednostką szeregowania* (ang. *scheduling entity*). Takie struktury zostały dołączone jako nowa składowa do każdego deskryptora procesu. Przykładem takiej jednostki szeregowania jest `rt_bandwidth`, czyli grupa zawierająca wszystkie procesy czasu rzeczywistego. Linux przyznaje 95% każdej sekundy czasu procesora dla tych procesów, a pozostałe 5% dla zwykłych procesów. Ta proporcja może być zmieniona przez uprzywilejowanego użytkownika. Jednostka `rt_bandwidth` została wprowadzona w wersji 2.6.23 jądra, aby zapobiec możliwości zmonopolizowania procesora przez procesy podlegające polityce szeregowania SCHED_FIFO.

Priorytety

Począwszy od wersji 2.6.23 jądra Linuksa, priorytety **wszystkich** procesów są statyczne, z jednym wyjątkiem. Priorytet zwykłego procesu może być tymczasowo podniesiony do priorytetu czasu rzeczywistego, jeśli zainicjował on wywołanie systemowe korzystające z tak zwanego muteksa czasu rzeczywistego (ang. RT-mutex). Ta operacja jest wykonywana, aby zapobiec problemowi *inwersji priorytetów*.

Szeregowanie sprawiedliwe — Wprowadzenie

Nadrzędnym celem szeregowania sprawiedliwego (ang. *fair scheduling*) jest zapewnienie każdemu procesowi *równego udziału* (ang. *fair share*) w mocy obliczeniowej CPU. Aby zrozumieć jak ono działa rozważmy idealnie wielozadaniowy procesor (ang. *perfectly multitasking processor*). Kiedy takie urządzenie wykonuje tylko jeden proces, to daje mu do dyspozycji 100% swojej mocy obliczeniowej, ale kiedy musi wykonać n identycznych procesów, to każdemu z nich oddaje $\frac{1}{n}$ tej mocy. W związku z tym każdy z nich wykonuje się n razy wolniej, ale za to wszystkie działają równocześnie, bez żadnych przerw. Niestety, to rozwiązanie nie może być zrealizowane w praktyce. Zamiast tego można przydzielać CPU procesowi na podstawie tego, *jak długo oczekiwał on na wykonanie*. Jeśli w systemie jest tylko jeden proces gotowy do wykonania, to może on otrzymać CPU na tak długo, jak potrzebuje, ale jeśli po pewnym czasie pojawi się inny proces gotów do działania, który jeszcze się nie wykonywał, to ten pierwszy zostanie niezwłocznie wywłaszczony z procesora, bo korzystał z niego dłużej niż ten drugi.

Szeregowanie sprawiedliwe — Wprowadzenie

Rozważmy jeszcze jeden scenariusz, w którym dwa nowe, identyczne procesy pojawiają się w tym samym czasie w systemie. Planista może wyliczyć czas, przez który mogą się one wykonywać (*czas wykonania*) przyjmując pewne *opóźnienie docelowe* (ang. *targeted latency*) i przydzielając każdemu z nich pewien jego udział. Opóźnienie docelowe to krótki odcinek czasu, zazwyczaj rzędu dziesiątek milisekund, ale dłuższy niż czas przełączania kontekstu. Jeśli rozszerzymy ten scenariusz na n procesów, to pojawi się problem. Przy n dążącym do nieskończoności, czas przez który pojedynczy proces może korzystać z CPU spada od zera. W związku z tym należy wprowadzić dolną granicę dla długości tego czasu, która jest nazwana *minimalną ziarnistością* (ang. *minimum granulity*). W rzeczywistości niektóre procesy są ważniejsze niż inne, co wyrażone jest przez ich priorytety. W szeregowaniu sprawiedliwym te priorytety są zamieniane na *wagi* używane przez planistę do wyznaczenia porcji opóźnienia docelowego dla każdego z procesów.

Planista CFS

Planista CFS (ang. *Completely Fair Scheduler*) zastąpił w Linuksie planistę O(1). Jego twórcą jest Ingo Molnár, który bazował na pomysłach Cona Kolivasa, australijskiego programisty jądra Linuksa. Ta zmiana została wprowadzona, aby rozwiązać pewne problemy z szeregowaniem procesów interaktywnych w komputerach osobistych (ang. *desktop computers*). Zgodnie z nazwą planista ten implementuje szeregowanie sprawiedliwe, choć *nie jest* ono całkowicie sprawiedliwe, jeśli liczba gotowych do wykonania procesów jest bardzo duża. Na szczęście taki przypadek występuje rzadko.

Planista CFS

Kod planisty CFS znajduje się w pliku `kernel/sched/fair.c`. Ten mechanizm używa dwóch tablicy, o 40 elementach każda, do zamiany priorytetów na wagi i odwrotnie. Pierwsza z nich nazywa się `sched_prio_to_weight` i przechowuje wagi. Waga dla domyślnego priorytetu (poziom uprzejmości 0) wynosi 1024. Wagi procesów o wyższych priorytetach są wyliczone poprzez mnożenie tej wartości przez kolejne potęgi liczby $1,25$. Wagi procesów o niższych priorytetach są wyliczone poprzez dzielenie tej wartości przez kolejne potęgi liczby $1,25$. Druga tablica nazywa się `sched_prio_to_wmul` i przechowuje odwrotności tych wag.

Planista CFS

Procesy są szeregowane zgodnie z ich *wirtualnym czasem wykonania* (ang. *virtual runtime*). Dla pojedynczego procesu wartość tego czasu jest wyliczana na podstawie faktycznego czasu jego działania, liczby procesów gotowych do działania i wagi związanej z jego priorytetem. Procesor jest przydzielany dla procesu z najkrótszym wirtualnym czasem wykonania, który jest mierzony w nanosekundach. Ta wartość jest przechowywana w polu `vruntime` struktury `se`, typu `struct sched_entity`, która z kolei jest polem deskryptora procesu. Czas wirtualny procesu jest aktualizowany przez funkcję `update_curr()`, która wywoływana jest okresowo i w następstwie określonych zdarzeń. Wartość opóźnienia docelowego jest przechowywana w zmiennej `sched_latency_ns` i domyślnie wynosi $20ms$. Może być ona zmieniona przez uprzywilejowanego użytkownika. Zmienna `sched_nr_latency` przechowuje maksymalną liczbę procesów, które muszą być przeszeregowane w tym okresie czasu. Podlega ona okresowej aktualizacji. Minimalna ziarnistość ma wartość $1ms$.

Planista CFS

Kolejka procesów gotowych, w przypadku planisty CFS, jest zaimplementowana w postaci drzewa czerwono-czarnego. Jest to rodzaj wyważonego drzewa BST, w którym każdy węzeł ma dodatkową cechę nazywaną *kolorem*, która podlega następującym regułom:

1. Korzeń drzewa jest zawsze czarny.
2. Każdy węzeł jest czarny lub czerwony.
3. Potomkowie czerwonego węzła są zawsze czarni.
4. Liście są zawsze czarne.
5. Wszystkie proste ścieżki, prowadzące od określonego węzła do związanych z nim liści, mają taką samą liczbę czarnych węzłów.

Jeśli te warunki są spełnione, to drzewo jest wyważone. Kiedy co najmniej jeden z nich nie jest spełniony, na skutek dodania nowego lub usunięcia istniejącego węzła, to należy to drzewo na nowo wyważyć poprzez wykonanie operacji rotacji określonych poddrzew i/lub zmianę kolorów odpowiednich węzłów.

Planista CFS

Jądro Linuksa posiada implementację drzewa czerwono-czarnego ogólnego przeznaczenia (jej szczegóły są wyjaśnione w trzeciej instrukcji laboratoryjnej, a opis samego drzewa znajduje się np. w książce „Wprowadzenie do algorytmów" autorstwa T. H. Cormena i innych). Planista CFS używa tej implementacji do sortowania procesów według ich wirtualnego czasu wykonania. Proces z najkrótszym wirtualnym czasem wykonania jest reprezentowany przez skrajnie lewy węzeł tego drzewa. Jeśli ten czas jest krótszy niż wirtualny czas wykonania bieżącego procesu, to ten proces jest wywłaszczany. Czas potrzebny na znalezienie skrajnie lewego węzła drzewa czerwono-czarnego wynosi $O(log_2(n))$, gdzie n jest liczbą procesów gotowych do wykonania. Aby skrócić czas trwania tej operacji, funkcja która jest odpowiedzialna za wstawienie nowego węzła do drzewa zapisuje w specjalnym wskaźniku jego adres, jeśli jest to skrajnie lewy węzeł. Wykrycie takiego przypadku nie jest skomplikowane: jeśli ta funkcja przeszukując drzewo celem wstawienia nowego elementu, zawsze wybiera lewą gałąź, to ten nowy węzeł będzie skrajnie lewy.

Planista CFS

Jeśli wspomniany wskaźnik ma wartość NULL, to oznacza to, że klasa związana z szeregowaniem algorytmem CFS jest pusta i planista powinien wybierać procesy z innej klasy.
Podobnie jak planista O(1) CFS stara się, aby nowo powstały proces otrzymał CPU przed swoim rodzicem. Aby osiągnąć ten cel musi on czasami zamienić miejscami wirtualne czasy wykonania tych procesów.
Planista CFS w porównaniu do O(1) zazwyczaj dłużej wykonuje operacje szeregowania z uwagi na konstrukcję jego kolejki procesów gotowych. Jednakże jest bardziej sprawiedliwy, jeśli chodzi o szeregowanie procesów interaktywnych i właśnie dlatego zastąpił swojego poprzednika.

Planista EEVDF

W wersji 6.6 jądra planista CFS został zastąpiony przez planistę EEVDF (ang. *Earliest Eligible Virtual Deadline First*). Nowy algorytm szeregowania radzi sobie lepiej z szeregowaniem procesów, które wymagają krótkich opóźnień (ang. *latency*) i obsługą nowoczesnych procesorów. Opóźnienie jest czasem, przez który proces oczekuje na przydział procesora. Procesy interaktywne korzystają z CPU przez krótki czas, ale potrzebują go dostać najszybciej jak się da. Z kolei procesy zorientowane na przetwarzanie korzystają długo z procesora, ale mogą poczekać na jego przydział.

Nowoczesne procesory posiadają rdzenie, które funkcyjnie są równoważne, ale różnią się wydajnością. Intel nazywa je (wprowadzając zamieszanie) P-rdzeniami i E-rdzeniami. Te pierwsze zapewniają dużą szybkość przetwarzania (ang. P —*performance*), a te drugie dużą efektywność (ang. E — *efficiency*) energetyczną. Szeregowanie procesów dla procesorów zbudowanych z takich rdzeni wymaga innego podejścia niż to używane w algorytmie CFS.

Planista EEVDF

Opis algorytm EEVDF został opublikowany w artykule z 1995 roku autorstwa Iona Stoicy i Husseina Abdel-Wahaba. *Nie jest* to algorytm szeregowania dla systemów czasu rzeczywistego i w dużym stopniu przypomina CFS. Podobnie jak on, EEVDF przydziela każdemu procesowi udział (ang. *fair share*) w czasie procesora, uwzględniając priorytet tego procesu. Jednakże, po wykorzystaniu tego udziału przez proces, planista oblicza różnicę między czasem CPU, który proces otrzymał, a tym przez który rzeczywiście z procesora korzystał. Ta wielkość nazywa się *spóźnieniem* (ang. *lag*). Procesy ze spóźnieniem większym lub równym zero są oznaczane jako *uprawnione* (ang. *eligible*) do wykonania. Planista powinien im przydzielić CPU w pierwszej kolejności, ponieważ nie uzyskały one w całości swojego przydziału procesora.

Planista EEVDF

Procesy z negatywnym spóźnieniem muszą poczekać na przydział CPU. Ten czas oczekiwania nazwany jest *czasem uprawnionym* (ang. *eligible time*). Jest on dodawany do wirtualnego czasu wykonania (ang. *virtual runtime*) procesu. Suma tych dwóch wielkości nazwana jest *wirtualnym terminem* (ang. *virtual deadline*) i oznacza czas, przed którego upłynięciem proces *nie powinien* otrzymać procesora.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!