Daniel Górski
Przemysław Jakubowski

Szeregowanie zadań we współczesnych systemach operacyjnych.

Szeregowanie zadań w Linux Kernel 2.6

Slajd 3: **Wstęp do szeregowania.**

Z szeregowaniem mamy do czynienia w momencie gdy wielu “klientów” chce skorzystać z wspólnego zasobu(zasobów). Szeregowanie powinno być sprawiedliwe, co w szczególności oznacza, iż każdy oczekujący w skończonym czasie ten zasób dostanie (brak zagłodzenia). Ważny jest koszt szeregowania czyli czas pomiędzy dostępem do zasobu przez kolejnych klientów i strategia dokonywania wyboru kolejnego klienta.

Slajd 4: **Scheduler.**

Scheduler jest to proces mający za zadanie podział czasu procesora pomiędzy procesy. Dobrze napisany scheduler powinien zapewniać dużą interaktywność podczas obciążenia systemu to jest nawet gdy jest uruchomione wiele procesów i pracujemy w edytorze tekstu naciskane klawisze powinny wyświetlać się bez opóźnienia. Sprawiedliwość: żadnego procesu nie możemy zagłodzić, co więcej każdy w rozsądnych odstępach czasu powinien dostęp do procesora otrzymywać. Scheduler powinien w pewien sposób rozróżniać procesy mniej i bardziej uprzywilejowane i tu standardowym rozwiązaniem jest przyznanie każdemu procesowi priorytetu oznaczające jego ważność.

Slajd 5: **Cele schedulera.**

Scheduler powinien umożliwiać poprawną pracę procesom czasu rzeczywistego. Idealnie by było gdyby czas działania schedulera był stały, niezależny od liczby procesów pracujących w systemie, rozwiązanie działające liniowo względem liczby procesów było by znacznie gorsze przy dużym obciążeniu systemu. Scheduler powinien w pełni wykorzystywać architekturę wieloprocesorową, w związku z czym nie powinien mieć żadnych globalnych struktur, gdyż dostęp do nich będzie wąskim gardłem systemu.

Slajd 6: **Projekt (design) schedulera w jądrze 2.6.**

Posiada on 140 poziomów priorytetów, mniejsza wartość oznacza większy priorytet. Z czego priorytety 1-100 odpowiadając procesom czasu rzeczywistego, a pozostałe zwykłym procesom. Mamy dwie tablice list priorytetów dla każdego procesora: tablice procesów, którym pozostały jeszcze jakieś kwanty czasu i te, które już swoje kwanty czasu wykorzystały. Na listach mamy procesy o tym samym priorytecie, mamy liczniki elementów na danej liście i mapę bitową zajętości. 1 pod i-tym bitem oznacza, że lista o i-tym priorytecie jest nie pusta. Tablice te są dostępne poprzez wskaźniki, co czyni operację zmiany epoki niezwykle szybką: po prostu zmieniamy wskaźniki.

Slajd 8, 9: Algorytm O(1).

Teraz dokładnie wyjaśnię w jaki sposób wszystkie operacje w schedulerze działają w czasie stałym.

Wybór procesu z największym priorytetem polega na wybraniu pierwszego elementu z listy o niepustej listy o największym priorytecie. Realizowane jest to poprzez wyszukanie pierwszej jedynki w bitmapie, ta operacja jest mocno zoptymalizowana. Następnie ten proces jest uruchamiany co również wykonywane jest w czasie stałym (zmiana kontekstu), niezależnie od liczby procesów czas działania algorytmu jest stały, ponadto w tym algorytmie nie ma żadnych elementów losowych, czyli jest on deterministyczny.

W schedulerze 2.6 zastosowano inne rozłożenie pracy przy obliczaniu nowych kwantów czasu, teraz w odróżnieniu od 2.4 nowy kwant obliczany i priorytet dynamiczny jest od razu po tym jak proces wykorzysta bieżący. W 2.4 przy zmianie epoki następowało ponowne wyliczenie wszystkich kwantów. Zatem zamiast raz wykonywać pracę O(n), wykonujemy n razy pracę O(1). Jest to bardzo duży postęp gdyż zastosowanie wielu procesorów istotnie poprawia czas pracy, a ponadto zapewnia to większą interaktywność podczas obciążenia systemu, gdyż nie ma przestojów na wyliczenie wszystkich priorytetów w jednym momencie. Zrozumienie tego jest istotne, gdyż w tej kwestii ilość pracy względem schedulera 2.4 się nie zmienia, zmienia się jedynie jej podział w czasie.

Slajd 10: Strategia szeregowania w schedulerze 2.6.

Teraz dokładniej omówię strategie szeregowania w 2.6. Mamy 140 priorytetów, przy czym 1-100 są priorytetami procesów czasu rzeczywistego, a

101-140 priorytetami procesów pozostałych. Mamy 3 różne strategie szeregowania, jedną dla zwykłych procesów i dwie dla procesów czasu rzeczywistego.

Slajd 11: Strategia szeregowania procesów zwykłych.

Każdy proces ma dwa priorytety – statyczny i dynamiczny. Statyczny priorytet zwany "nice" jest z zakresu -20..19, domyślnie jest on ustawiany na 0, ale można go zmienić poprzez wywołanie funkcji systemowej nice(). Procesy są wykonywane od najwyższych priorytetów, (pierwsza jedynka w bitmapie, najwyższy priorytet oznacza najmniejszą wartość!!!), dokładnie jest wykonywany pierwszy z listy procesów o najwyższych priorytecie. Gdy wszystkie procesy wykorzystają kwanty czasu rozpoczyna się nowa epoka.

Slajd 12: Interakcyjność. W zależności od tego czy proces korzysta głównie z procesora, czy też głównie z operacji I/O jest przydzielany mu priorytet dynamiczny. Procesy często korzystające z I/O są nieco uprzywilejowane względem procesów korzystających głównie z procesora. Ten bonus wyliczany jest na podstawie czasu jaki proces pozostawał w uśpieniu(prawdopodobnie czekał na I/O) względem maksymalnego czasu oczekiwania. Priorytet dynamiczny jest obliczany w momencie budzenia procesu i umieszczania go w Runqueue.

Nie dotyczy procesów czasu rzeczywistego. Dokładnie priorytet dla procesów użytkownika wygląda następująco:

Prio = p->static_prio - bonus

Bonus = CURRENT_BONUS(p) – MAX_BONUS/2

CURRENT_BONUS(p) = NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS / MAX_SLEEP_AVG)

Slajd 13: Interakcyjność...

Bonus był obliczany "na krótką metę", dlatego są dodatkowo wprowadzone inreactivity_credit, które pamiętają na dłuższą metę zachowanie procesu w przeszłości. Bez tego w sytuacji gdy proces często korzystający z procesora zaśnie na dłużej raz na pewien czas byłby wynagradzany dużym bonusem mimo iż nie powinien być, gdyż procesem interakcyjnym nie jest. Odwrotnie: proces głównie czekający na wejściu/wyjściu w sytuacji gdy raz na jakiś czas będzie chciał skorzystać dłużej z procesora zostanie mocno ukarany, mimo iż nie

powinien być, gdyż jest procesem interakcyjnym. Procesy zyskują interaktviwy_credit podczas czekania, tracą go korzystając z procesora. Wartość interactivity_credit jest brana pod uwagę w następujących sytuacjach:

1) Proces o niskiej wartości interactivity_credit (mniej niż -100) jest budzony z długiego uśpienia – nie zyskuje on czasu oczekiwania na I/O liczonego w zmiennej sleep_avg (na jej podstawie oblicza się priorytet dynamiczny)
2) Proces silnie interakcyjny (więcej niż +100) długo korzystał z procesora – nie zostaje on ukarany zmniejszeniem priorytetu.

Slajd 14: Ponowne umieszczanie procesów w kolejce aktywnych.

Raz na 1ms wywoływany jest scheduler_tick(). Jeśli bieżący proces wykorzystał już swój kwant czasu, ma przydzielany nowy i jest umieszczany w kolejce zużytych. Aby uniknąć przestojów procesy interakcyjne mogą być włożone ponownie do kolejki aktywnych po wykorzystaniu kwantu czasu, pod warunkiem, że nie głodzimy procesów z kolejki zużytych.

Slajd 15: Obliczanie kwantów czasu.

Wielkość kwantu czasu jaka jest przyznawana procesowi jest określona wzorem:

BASE_TIMESLICE(p) = MIN_TIMESLICE + (MAX_TIMESLICE – MIN_TIMESLICE) *(MAX_PRIO-1 – (p->static_prio) / (MAX_USER_PRIO-1))

Jak widać zależy ona jedynie od wartości priorytetu statycznego (nice). Jednak aby procesy o wysokich priorytetach nie mogły zajmować procesora rzadko, ale na długo (pogorszyło by to czas reakcji systemu) wprowadzone zostały TIMESLICE_GRANULARITY, które oznaczają ile czasu proces może jednorazowo wykorzystać ze swojego kwantu czasu, o ile w kolejce gotowych są inne procesy o tym samym priorytecie. W tej sytuacji proces po wykorzystaniu TIMESLICE_GRANULARITY ms jest umieszczany na końcu kolejki gotowych z kwantem czasu pomniejszonym o daną wartość.

Slajd 16: Fork ().

Nowy proces utworzony przez fork(), jest umieszczany w kolejce gotowych, ale: bieżąca ilość czasu jaką posiada rodzic jest dzielony między obydwa procesy. Podobnie zostaje zmniejszona wartość sleep_avg rodzica i dziecka, na podstawie której jest obliczany priorytet dynamiczny procesu. Uniemożliwia to

zajęcie całego czasu procesora przez grupę procesów, gdy inne procesy również ten czas chcą uzyskać.

Slajd 17: Strategie szeregowania dla procesów czasy rzeczywistego.

Linux zapewnia jedynie soft RT. Oznacza to iż proces czasu rzeczywistego nie ma gwarancji, że konkretną pracę wykona w konkretnym czasie, czy też że w konkretnym momencie będzie miał on dostęp do procesora. Oznacza to jedynie, że w sytuacji gdy jakiś proces czasu rzeczywistego chce pracować, procesora nie dostaną pozostałe procesy. Brak możliwości zapewnienia pełnego RT wynika z wielu rzeczy: między innymi z tego iż nie wiemy ile czasu zajmie nam odczytanie strony pamięci (może ona być w cache'u procesora, może być w pamięci, lub trzeba ją będzie ściągnąć z dysku). Linux daje nam 2 strategie szeregowania procesów czasu rzeczywistego:

Strategia fifo działa następująco: proces czasu rzeczywistego, który otrzymał procesor, działa dotąd aż zrzeknie się go dobrowolnie, lub pojawi się proces z wyższych priorytetem, który go wydziedziczy. Oczywisty jest przypadek zagłodzenia, gdy dwa procesy czasu rzeczywistego o tym samym priorytecie chcą uzyskać czas procesora, a działa tylko jeden z nich.

W strategii SCHED_RR procesy mają przydzielane kwanty czasu, działają aż do ich wykorzystania, po czym mają je przyznane ponownie i umieszczane są na końcu kolejki gotowych o danym priorytecie. Oznacza to, że jedynie procesy RT o najwyższym priorytecie dzielą między siebie procesor, podczas gdy pozostałe będą głodzone.

Slajdy 18,19,20: Struktura RunQueue

spinlock_t lock

- zapewnia, że tylko jeden proces może modyfikować kolejkę w danym czasie

unsigned long nr_running

- liczba gotowych procesów w kolejce

unsigned long long nr_switches

- liczba zmian kontekstu od momentu powstania kolejki (nigdzie nie używane)

atomic_t nr_iowait

- liczba procesów czekających na I/O

unsigned long nr_uninterruptible

- liczba zadań w kolejce, które nie obsługują przerwań

unsigned long expired_timestamp

- stempel czasowy ostatniej zmiany epoki

int best_expired_prio

- najwyższy priorytet procesu czekającego w kolejce zużytych

task_t *curr

- wskaźnik do aktualnie działającego procesu

task_t *idle

- wskaźnik do procesu który działa wtedy, gdy nic innego nie działa

prio_array_t *active

prio_array_t *expired

- wskaźniki do tabel

unsigned long long timestamp_last_tick

- stempel czasowy ostatniego wywołania scheduler_tick()

Slajd 21: Strukura Priority array

unsigned int nr_active

- liczba gotowych zadań w kolejce

unsigned long bitmap[BITMAP_SIZE]

- do szukania najwyższego priorytetu

struct list_head quene[MAX_PRIO]

- tablica list procesów o danych priorytetach

Slajdy 22,23,24,25: Wieloprocesorowość

SMP – symetryczna wieloprocesorowość: mamy wiele procesorów mających wspólną pamięć (cache oddzielne) i dzielących również inne zasoby. Scheduler 2.6 nie zawiera globalnych struktur, które stanowiłyby „wąskie gardło” systemu. Każdy z procesorów ma swoją oddzielną RunQueue. Raz na 1ms, gdy procesor

jest bezczynny (pracuje tylko proces idle), a na 200ms gdy procesor pusty nie jest, następuje wywołanie funkcji load_balance(). Ma to zapewnić zrównoważenie podziału zadań pomiędzy procesory. Działa ona następująco: jest wyszukiwany najbardziej obciążony procesor i jeśli różnica w ilości zadań jest większa niż 25% następuje równoważenie. Z obciążonego procesora są wybierane procesy do przeniesienia na bieżący: brane pod uwagę są preferencje procesu co do procesorów na których może być uruchamiany, kiedy ostatnio działał on na procesorze z którym jest związany (zbyt „świeżych procesów nie przenosimy – może być coś jeszcze w cache'u).

Slajdy 26,27,28,29: Tuning schedulera 2.6.

Jako, że Linux jest systemem open source, sami mamy możliwość zmiany systemu i tak możemy sami zmienić działanie schedulera. Można to osiągnąć zmieniając stałe z których on korzysta i po przekompilowaniu jądra zostaną wprowadzone nowe ustawienia. I tak możemy zmienić:

MIN_TIMESLICE, MAX_TIMESLICE czyli przedział z jakiego procesy mają przydzielane kwanty czasu. (Kwant czasu jest liczony jedynie na podstawie priorytetu statycznego nice). Zwiększenie tych zmiennych przekłada się na zwiększenie ogólnej wydajności systemu, ale pogorszeniu ulega jego czas reakcji na działania użytkownika. Jest to przydatne gdy zadaniem danego komputera nie jest głównie interakcja z użytkownikiem, czyli np. na serwerach.

PRIO_BONUS_RATIO – zakres przedziału z jakiego wartości przyjmuje priorytet dynamiczny, wyrażony w % przedziału procesu statycznego, domyślnie ustawiony na 25% zatem jest to przedział [-5,5], (priorytet statyczny nice przyjmuje wartości z przedziału[-20,19]. Współczynnik ten oznacza na ile ważny jest priorytet dynamiczny względem statycznego, manipulując nim zmieniamy swój wpływ na szeregowanie procesów (maleje lub rośnie rola priorytetu statycznego, który możemy zmieniać poprzez wywołanie funkcji systemowej nice()).

MAX_SLEEP_AVG – ograniczenie na parametr SLEEP_AVG. Ta wartość oznacza ile najwyżej czasu może uzbierać proces czekający, jest ona w mianowniku wzoru na priorytet dynamiczny. Zwiększenie tej wartości bardziej „wyrówna”, różnice między procesami interakcyjnymi i nie. Trudniej będzie uzyskać procesowi maksymalny bonus i maksymalną karę.

STARVATION_LIMIT – jeśli procesy w kolejce zużytych czekają mniej

czasu, to możliwe jest ponowne umieszczanie procesów interakcyjnych w kolejce gotowych. Sterując tą wartością ustalamy ważność procesów działających w tle, jeśli zależy nam przede wszystkim na tym aby praca z systemem była sprawna, a poza tym dużo rzeczy w tle ma się wyliczać w momentach gdy nic nie robimy, możemy tą wartość zwiększyć, jeśli chcemy aby procesy w tle wykonywały się równie szybko co procesy interakcyjne z którymi pracujemy wartość tą zmniejszamy.

Slajd 30: Porównanie 2.4 z 2.6.

Co można by poprawić w obecnym schedulerze?

Ciekawą opcją wydaje się stworzenie profili ustawień schedulera do różnych zadań, umożliwiłoby to dynamiczne zmienianie ustawień w zależności od tego, do czego akurat komputer nam służy.

Gdy mamy do czynienia z systemem na którym pracuje wielu użytkowników naraz, można byłoby zredukować rolę głównego schedulera do podziału czasu procesora pomiędzy poszczególnych użytkowników, a każdy z użytkowników miałby własnego schedulera do podziału jego czasu między jego procesy. Takie rozwiązanie doprawdy dodatkowo obciążałoby system, ale byłoby bardziej sprawiedliwe ze względu na podział czasu między użytkowników.

Slajd 31: Test porównawczy 2.4 z 2.6.

Zaczerpnięte ze strony: http://developer.osdl.org/craiger/hackbench/

Scheduler w jądrze 2.6 zdecydowanie różni się od tego w 2.4. Nie jest jego bezpośrednią kontynuacją, gdyż cały był napisany od nowa. Dużo rzeczy ulepszono i tak: czas zmiany kontekstu i zmiany epoki jest stały w 2.6, wobec liniowego czasu w 2.4. Szczególnie ma to znaczenie przy obciążonym systemie, gdyż wtedy obsługa zmiana kontekstu i epoki jest zdecydowanie dłuższa w 2.4. Kolejną wadą 2.4 jest jedna globalna kolejka procesów z której korzystały wszystkie procesory, wprowadzenie wielu kolejek w 2.6 znacznie poprawiło wydajność na systemach wieloprocesorowych. Kolejną wadą 2.4 jest brak możliwości wywłaszczania procesów czasu rzeczywistego, co zmniejsza czas reakcji systemu.

Slajd 32: Opis testu.

Omówię teraz test porównawczy schedulera w jądrze 2.4 i 2.6.

Zostało uruchomionych kolejno 25, 50, 75...200 par procesów klient –

serwer, każdy klient wysyłał do każdego serwera 100 komunikatów, test dla każdej wartości był wykonywany wielokrotnie i była wyciągana średnia z czasów wykonania.

Slajdy 33,34,35,36: Test wydajnościowy.

Testy wydajnościowe schedulera 2.4 i 2.6 kolejno na 1,2,4,8 procesorach. W każdym teście zdecydowanie wygrywa 2.6, przy czym dla większej liczby procesorów różnica jest większa.

\Slajdy 37,38: Skalowalność.

Testy skalowalności. Wyraźnie widać, że linux 2.4 przy dużej liczbie procesów nie jest skalowalny, znaczy to, że większa liczba procesorów nie przekłada się (lub przekłada się bardzo słabo) na wydajność systemu. Linux 2.6 nie posiadający globalnych struktur skalowalny jest, gdyż zwiększanie liczby procesorów zwiększa znacząco wynik uzyskany w teście.

Slajdy 39,40,41: Gospodarowanie zasobami.

Testy gospodarowania zasobami, było wykonywane testy dla kolejnych wartości do czasu, gdy system testy poprawnie wykonywał. Linux 2.6 wraz ze zwiększeniem liczby procesorów i co za tym dostępnych dla systemu zasobów umożliwiał działanie większej liczbie procesów, natomiast 2.4 mimo zwiększenia liczby zasobów nie wykorzystywał ich lepiej.

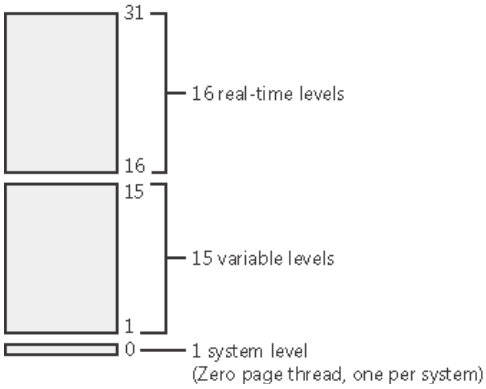
Szeregowanie zadań w Windows XP

Szeregowanie wątków w Windowsie jest zaimplementowane wewnątrz jądra systemu. Nie ma żadnego pojedynczego modułu czy algorytmu odpowiadającego za szeregowanie. Aczkolwiek kod rozprzestrzeniony jest w jądrze w zdarzeniach związanych z szeregowaniem. Ogólnie wszystkie te metody nazywane są „kernel dispatcher”. Następujące zdarzenia mogą wywołać „dispatching” :

- Wątek jest gotowy do uruchomienia, np. został utworzony, zakończył status czekania.
- Wątek oddaje procesor, np. zakończył się, skończył mu się kwant czasu, wszedł w stan czekania.
- Zmienił się priorytet wątku.
- Wątek nie może wykonywać się już na danym procesorze, ponieważ zmienił mu się atrybut processor affinity.

W każdym z tych przypadków, system musi zadecydować jaki wątek powinien być uruchomiony jako następny. Gdy już zostanie on wybrany, następuje zmiana kontekstu.

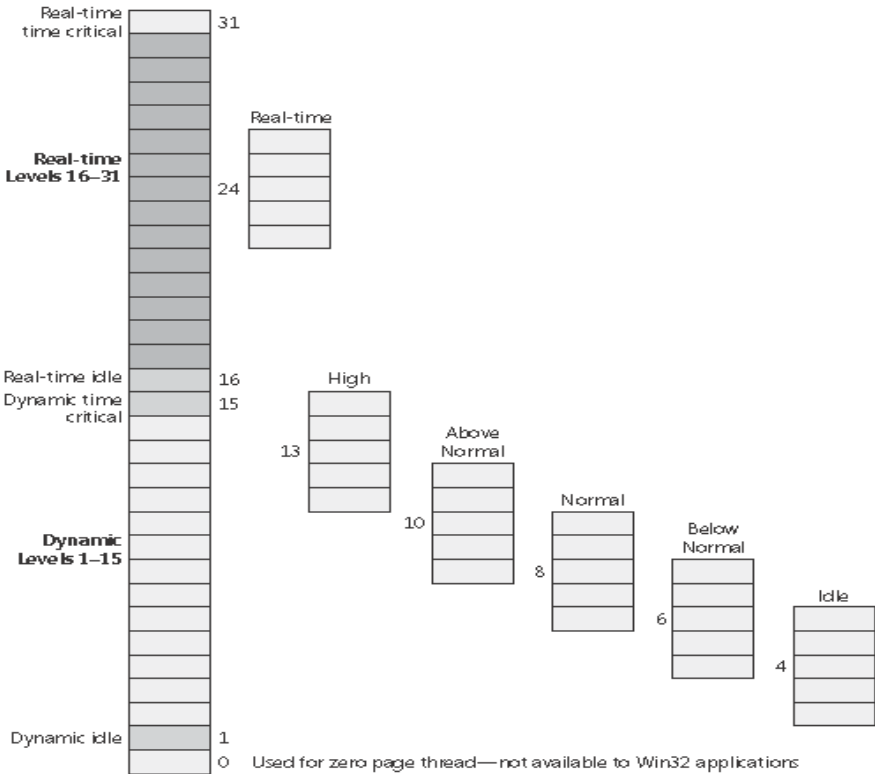
Aby zrozumieć algorytmy szeregowania, trzeba wpierw poznać priorytety których używa Windows aby je wykonywać. Do szeregowania wątków używa się 32 priorytetów, w zakresie od 0 do 31. Dzielą się one następująco:



- Szesnaście poziomów czasu rzeczywistego (16 – 31)
- Piętnaście dynamicznych poziomów (1 – 15)
- Jeden poziom systemowy (0)

Priorytety wątków są tworzone z dwóch różnych perspektyw, Windows API i jądra systemu. Najpierw API przyznaje klasę priorytetu procesom, podczas ich tworzenia (Czas rzeczywisty, Wysoki, Powyżej normalnego, Normalny, Poniżej normalnego, Niski), następnie wewnątrz tych klasy istnieją podklasy dla

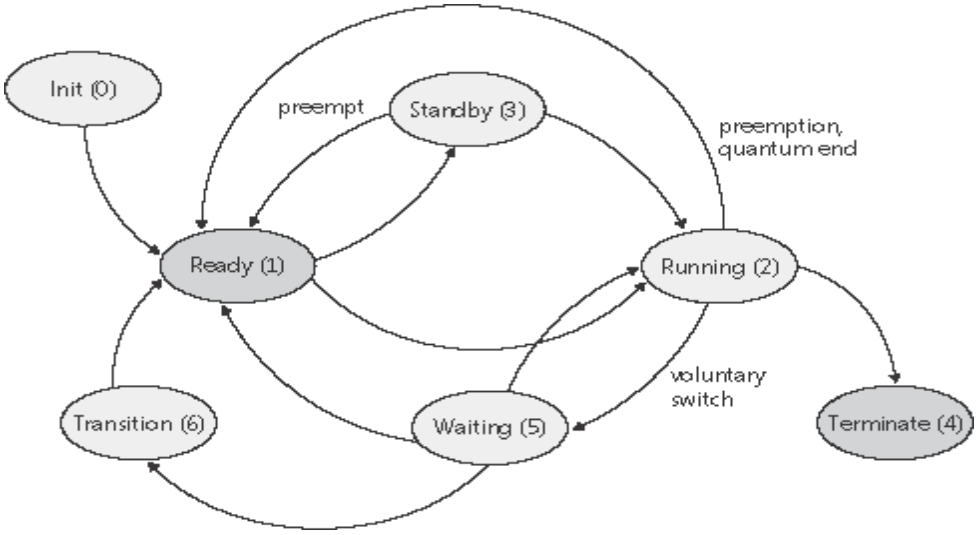
utworzonych przez te procesy wątków (Krytyczny, Wysoki, Powyżej normalnego, Normalny, Poniżej normalnego, Niski).



Proces posiada tylko jedną wartość priorytetu, ale jego wątki już nie. Każdy wątek posiada bazowy priorytet (dziedziczony z procesu) i aktualny priorytet. To właśnie według aktualnego priorytetu następuje szeregowanie. Z różnych powodów Windows może zmienić priorytet wątków w dynamicznym zakresie (1 – 15) przez pewien okres czasu. Natomiast nigdy nie zmieni priorytetów wątków w zakresie czasu rzeczywistego. Posiadają one zawsze ten sam priorytet bazowy, jak i aktualny.

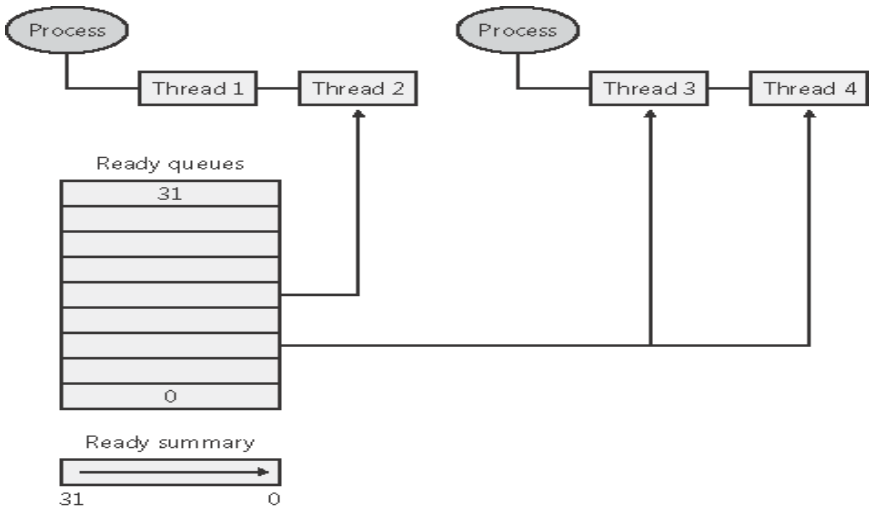
Priorytety w dynamicznym zakresie mogą być zmieniane przez użytkownika (o ile posiada prawa do zmiany priorytetów).

Aby przejść do algorytmu szeregowania, trzeba jeszcze zobaczyć stany w jakich może znajdować się wątek w Windowsie XP:



- Ready(Gotowy) – Wątek jest gotowy do uruchomienia
- Standby(Oczekiwanie) – Wątek został wybrany do uruchomienia, jeśli zajdą wszystkie potrzebne warunki , nastąpi zmiana dla kontekstu dla tego wątku. Zauważmy, że z tego stanu wątek może zostać wywłaszczony.
- Running(Uruchomiony) – Wątek właśnie jest uruchomiony.
- Initialized (Tworzenie) – Jest to stan podczas, którego wątek jest tworzony.
- Terminate(Zakończony) – W tym stanie wątek, znajduje się, gdy zakończy działanie.
- Waiting (Czekanie) – wątek jest w stanie czekania, np. na jakiś semafor.
- Transition – wątek jest w tym stanie, gdy jest gotowy do uruchomienia, lecz jądro systemu nie posiada wolnej pamięci. Gdy znajdzie się pamięć, wątek przechodzi do stanu Gotowy.

Aby podejmować decyzje na temat szeregowania wątków, system potrzebuje struktur danych, aby przechowywać informacje o gotowych do uruchomienia wątkach. Jest to tzw. „Dispatcher database”. W jego skład wchodzi KiDispatcherReadyListHead zawierający wątki gotowe do uruchomienia. Jest to lista 32 kolejek (każda dla jednego priorytetu). Aby przyspieszyć wyszukiwanie Windows trzyma też 32-bitową bitmapę nazwaną KiReadySummary. Każdy bit reprezentuje czy dla danego priorytetu, istnieje jakikolwiek gotowy do uruchomienia wątek.



Gdy, wątek jest gotowy do uruchomienia, dostaje procesor na okres kwantu czasu. Kwant czasu może być różnej długości, ze względu na kilka czynników:

- Konfiguracja systemu (długie czy krótkie)
- Status procesu(Pierwszoplanowy czy w tle)

Jednakże, proces może nie ukończyć swojego kwantu czasu, gdyż system Windows uwzględnia wywłaszczanie podczas szeregowania wątków. Ponadto, wątek może zostać wywłaszczony zanim zacznie wykorzystywać swój kwant czasu.

Kwant czasu może być różnej długości. I tak w Windowsie XP jest on domyślnie równy dwóm tyknięciom zegara. Natomiast w Windows Server 2003 wynosi on 12 tyknięć zegara. Związane to jest z tym aby na systemie serwerowym zminimalizować liczbę zmian kontekstu. Dzięki dłuższemu kwantowi czasu, aplikacja serwerowa ma większą szansę na zrealizowanie zapytania klienta.

Długość tyknięcia zegara zależy od platformy sprzętowej. Dla przykładu dla większości jednoprocesorowych maszyn architektury x86 wynosi on około 10 milisekund, natomiast dla wieloprocesorowych maszyn architektury x86 około 15 milisekund.

Każdy wątek, kiedy otrzymuje kwant czasu, dostaje zmienną quantum value ustawianą jako trzykrotność tyknięć zegara dla kwantu (np. dla Windows XP jest to 2*3 = 6). Każde tyknięcie zegara zmniejsza tą wartość o trzy. Wartość ta jest trzymana jako wielokrotność tyknięć zegara, aby można też ją zmniejszać pojedynczo. I tak właśnie, gdy wątek wykona funkcja czekania, która zakończy natychmiast, wartość quantum value zmniejsza się o jeden. Jeśli czekanie nie zakończy się od razu, to wątkom o priorytecie poniżej szesnastu też pomniejsza się o jeden wartość quantum value. Gdy nie było takiej kontroli, wątek który zacząłby działać, następnie przeszedł w stan czekania, następnie znowu się uruchomił i przeszedł w stan czekania, jeśli nie byłby aktywnym procesem podczas tyknięcia zegara, nigdy nie skończyłby swojego kwantu czasu.

Użytkownik ma pewną kontrolę nad kwantem czasu. Służy do tego wartość rejestru pod adresem HKLM\ SYSTEM\ CurrentControlSet\ Control\ PriorityControl\ Win32PrioritySeparation.

4	2	0
Short vs. Long	Variable vs. Fixed	Foreground Quantum Boost

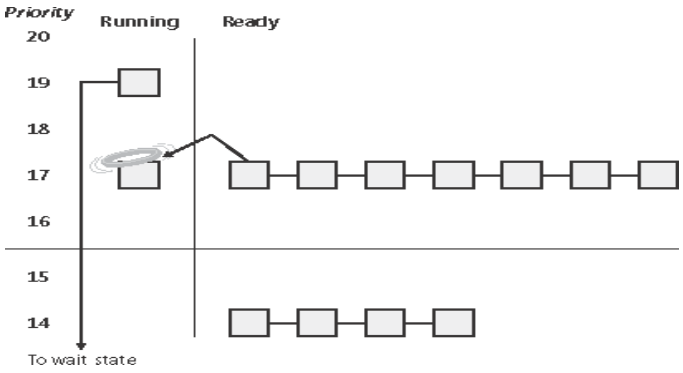
Jest to sześciobitowy rejestr opisujący :

- Czy kwant czasu ma być długi czy krótki.
- Czy kwant czasu ma być zmieniany dla procesów pierwszoplanowych
- Jeśli tak, to o ile ma być przyspieszany dla procesów pierwszoplanowych.

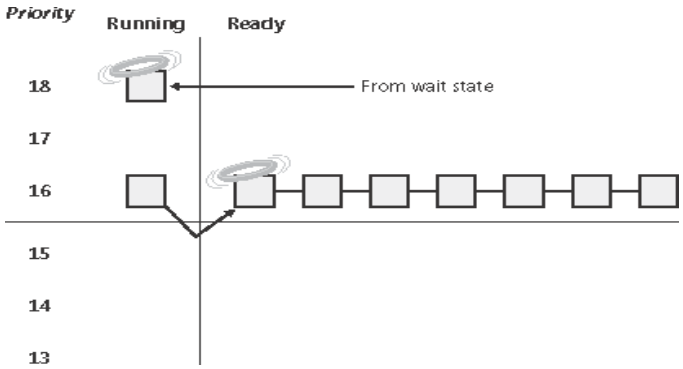
Jakie mogą być scenariusze szeregowania?

- Wątek oddaje procesor, np. zawisł na semaforze, operacji wejścia/wyjścia.

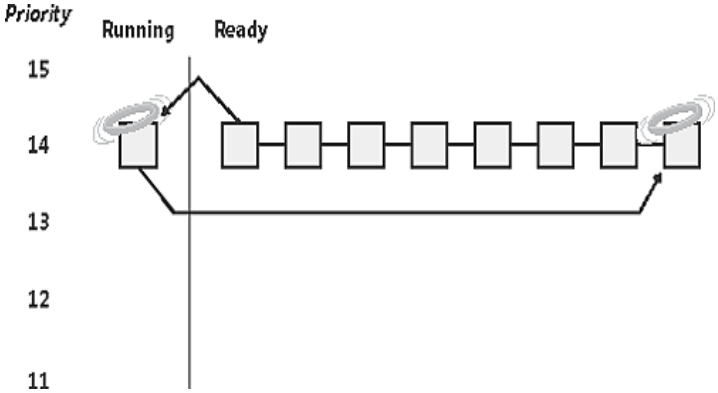
Na jego miejsce wchodzi pierwszy wątek z kolejki o największym priorytecie.



- Wątek zostaje wywłaszczony – wraca wtedy na początek kolejki o swoim priorytecie.



- Wątek oddaje procesor, np. skończył mu się kwant czasu, zakończył działanie. Wątek wraca na koniec kolejki o swoim priorytecie.



Priorytety wątków mogą się zmienić się ze względu na pewne okoliczności :

- Koniec czekania na operację wejścia/wyjścia

  Wątek może uzyskać następujące zwiększenie priorytetu za czekanie na określone urządzenia:
  - Dysk, CD – ROM - +1
  - Sieć - +2
  - Myszka, klawiatura - +6
  - Dźwięk - +8

  Aczkolwiek, jego priorytet może osiągnąć co najwyżej 15. Teraz po każdym zakończonym przez niego kwancie czasu, jego priorytet jest zmniejszany o jeden, aż powróci do bazowego.

- Koniec czekania na zdarzenie lub semafor

  Wątek dostaje przyspieszenie o 1, ale też nie może ono powodować przekroczenie 15. Po wykorzystanym kwancie czasu priorytet powraca do normy.

- Koniec czekania procesu pierwszoplanowego

  Wątek pierwszoplanowy zostaje przyspieszony o wartość zmiennej jądra systemu o nazwie PsPrioritySeparation. Wartość tej zmiennej może wynosić 0, 1 lub 2.
- Koniec czekania na zdarzenie związane z GUI

  Wątek kończący taki rodzaj oczekiwania dostaje przyspieszenie o 2.

- Przeciwdziałanie zagłodzeniu.

  Ponieważ w systemie działają wątki o różnych priorytetach istnieje możliwość, iż wątek o małym priorytecie może nie dostawać procesora, przez długi okres. Dlatego co sekundę wywoływany jest balance set manager, który szuka wątków gotowych do uruchomienia od co najmniej czterech sekund.
  Taki wątek dostaje priorytet równy 15 i podwójny kwant czasu. Po wykorzystaniu czasu, wraca do poprzedniego priorytetu.

A jak system Windows XP radzi sobie z wieloma procesorami?

Na początek potrzebne są zmiany w Dispatcher Database. I tak dodane są dwie nowe zmienne KeActiveProcessors. Jest to bitmapa procesorów, których można używać. Dostępna też jest bitmapa KiIdleSummary, która trzyma informację o procesorach, które nic nie robią. Mamy także dwa spinlocki KiDispatcherLock i KiContextSwapLock, odpowiedzialny za synchronizację dostępu.

Każdy wątek ma ustawioną zmienną o nazwie affinitymask, która mówi na jakich procesorach może być on odpalany. Pamiętajmy jednak o tym, że Windows nie będzie przestawiał wątków pomiędzy procesorami. Dla przykładu, jeśli na pierwszym procesorze jest odpalony wątek o priorytecie 8 (mógłby być odpalany na wszystkich procesorach), i przychodzi wątek o priorytecie 6, który nie może być odpalony na drugim procesorze, to system nie przestawi wątku z pierwszego procesora na drugi, aby mogły działać jednocześnie.

Każdy wątek ma zapisane dwa numery procesorów. Procesor idealny, ten na którym chciałby być odpalany, i procesor na którym był ostatnio odpalony.

Algorytmy szeregowania można podzielić na dwie części:

- Wybieranie procesora to wątku
  Jeśli są wolne procesory, to najpierw próbuje się na nich odpalić wątek. Zbiór wolnych procesorów redukuje się z affinitymask wątku. Następnie sprawdza się czy nie jest wolny idealny procesor wątku. Jeśli nie, to teraz jeśli procesor, na którym wykonuje się kod szeregowania jest wolny, to wątek odpalany jest na nim. Następnie bierze się wolny procesor o najmniejszym numerze.

  Jeśli nie ma wolnych procesorów, to system próbuje odpalić wątek na jego idealnym procesorze. Jeśli działający tam wątek ma większy priorytet, to wątek wrzucany jest do kolejki, a jeśli ma większy priorytet to wywłaszcza działający wątek.

- Wybieranie wątku do procesora
  Czasami trzeba wybrać do procesora wątek do odpalenia. Wybiera się niepustą kolejką o najwyższym priorytecie i na niej szuka wątków spełniających jeden z tych warunków
  - Był ostatnio odpalany na tym procesorze
  - Ten procesor jest dla niego idealny

- Nie był odpalany przez co najmniej trzy tyknięcia zegara
- Ma priorytet większy niż 24

Jeśli żaden wątek nie spełnia któregoś z warunków, wybierany jest po prostu pierwszy z kolejki.

Zestawienie schedulera w Linux 2.6 i Windows XP

Linux 2.6	Windows XP
Open source – można wprowadzić samemu dowolne zmiany	Brak możliwości wprowadzania samodzielnych zmian
Niedoskonały RT – proces RT może zagłodzić inne procesy RT	Niedoskonały RT – proces RT może zagłodzić inne procesy RT
Proces korzystający głównie z procesora, a raz na jakiś czas z I/O bonusu nie dostaje	Proces czekający na I/O dostaje bonus zawsze po obudzeniu
Aby wprowadzić zmiany trzeba rekompilować jądro	Część parametrów można zmienić poprzez aplikacje i rejestr. Reszty nie.
Oddzielne kolejki dla każdego procesora	Jedna globalna kolejka dla wszystkich procesorów