Linux 2.6.8.1

Adam Bułak
Ha Nhat Viet
Damian Klata

1. Wstęp

- Wieloprocesowe systemy:

  - z możliwością wydzierżawiania (ang. preemption) – Unix, Linux też oczywiście

  - bez takiej możliwości – proces sam oddaje procesor (ang. yielding) – wszystkie Mac OS do 9 włącznie
- Nowy scheduler – zaimplementowany przez Ingo Molnara, który prace nad nim rozpoczął już w grudniu 2001.

2. Po co komu nowy scheduler? Jakie postawiono przed nim zadania?

- Uniezależnienie czasu działania schedulera od liczby procesów – tzw. O(1) scheduling
- Osobne kolejki zadań dla każdego procesora
- Równomierny rozdział zadań pomiędzy wszystkie procesory
- Szybkie reagowanie dla procesów interaktywnych
- Sprawiedliwość w podziale czasu procesora i zapobieganie głodzeniu
- Lepsze mechanizmy szeregowania zadań czasu rzeczywistego
- Sprawniejsze szeregowanie dla architektury NUMA (Non Uniform Memory Access)

3. Mała powtórka

- W Linuxie mamy dwa rodzaje procesów:
- I/O-Bound – takie, które większą część czasu spędzają na zlecaniu/oczekiwaniu operacji we/wy
- Processor-Bound – te, które przez większość czasu korzystają z procesora

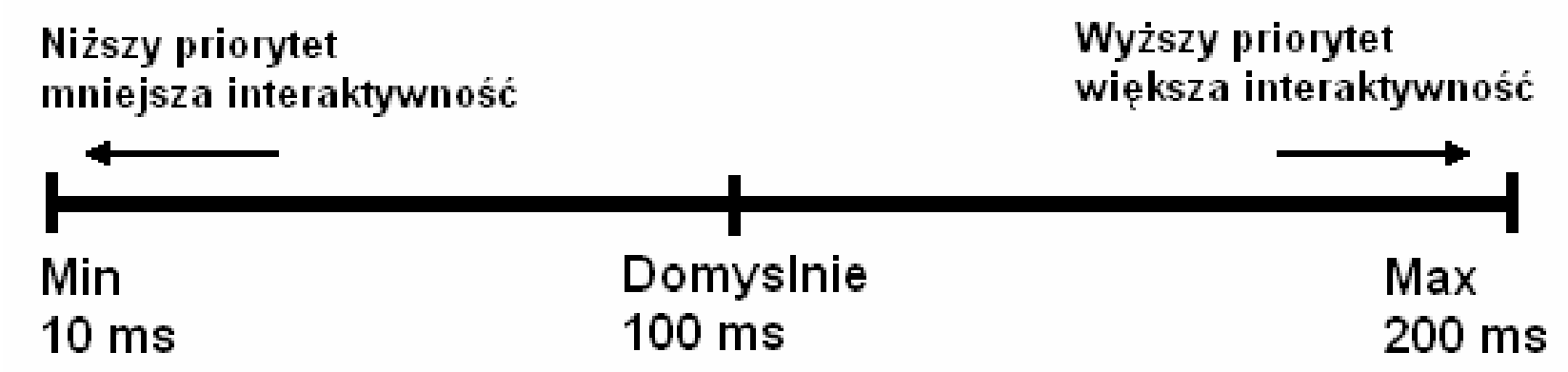
4. Priorytety:

- Dwa rodzaje priorytetów:
- nice priority – od -20 do 19 (domyślnie 0) – im mniejsza wartość tym zadanie bardziej uprzywilejowane
- real-time priority – od 0 do 99 – większa wartość – mniej ważne zadanie

5. Kwanty czasu (ang. timeslice)

- Kwant czasu – w milisekundach określa, ile czasu procesora ma do dyspozycji zadanie.
- Kwanty przyznawane są rundami – oznacza to, że zadanie które wykorzystało swój przydział może rozpocząć korzystanie z następnego przydzielonego mu kwantu czasu dopiero, gdy wszystkie inne zadania wyczerpią swoje przydziały.

6.Wykres 1


Niższy priorytet
mniejsza interaktywność
Wyższy priorytet
większa interaktywność
Min
10 ms
Domyslnie
100 ms
Max
200 ms

7. Kolejki procesów działających (runqueues)

- Najważniejsze składowe

```
struct runqueue {
  unsigned long         nr_running; // liczba zadań
  w kolejce
  struct task_struct    *curr       // obecnie
  wykonywane zadanie
  struct prio_array     *active           //
  wskaźnik do tablicy zadań, które nie wyczerpały
  jeszcze przyznanego im czasu
  struct prio_array     *expired    // wskaźnik do
  zadań z wyczerpanym kwantem czasu
}
```

8. Tablice priorytetów

```
struct prio_array {
  int               nr_active;
  unsigned long          bitmap[BITMAP_SIZE];
  struct list_head       queue[MAX_PRIO];
}
```

- nr_active – ilość wszystkich zadań
- bitmap – mapa bitowa z ustawionym bitem x jeżeli wśród wszystkich zadań jest przynajmniej jedno zadanie o priorytecie x. Ponieważ liczba priorytetów jest równa 140, a int ma 32 bity, potrzebujemy 5 słów, stąd mapa bitowa = 160 bitów
- queue[x] – lista procesów o priorytecie x
- MAX_PRIO – liczba wszystkich możliwych priorytetów (100 dla zadań czasu rzeczywistego + 40 dla pozostałych = 140)

9. Do czego służą prio_array?

- Każde zadanie, które wyczerpało swój kwant czasu usuwane jest z kolejki **active** i umieszczane w kolejce **expired** z nowo przyznanym priorytetem i kwantem czasu (jak są przyznawane – o tym za chwilę). Do czasu, aż w kolejce **active** są jeszcze jakieś zadania – scheduler przydziela im odpowiednio (jak - o tym również za chwilę) procesor do momentu, aż ostatnie zadanie wyczerpie swój kwant czasu. Wtedy następuje zamiana kolejki **active** z kolejka **expired** i już.

10. Dynamiczne priorytety

- prio - dynamiczny priorytet
- static_prio – podstawowy, statyczny priorytet
- sleep_avg – określa interaktywność zadania
- effective_prio() - ustala wartość prio na podst. static_prio i sleep_avg
- prio = static_prio +/- max. 5

11. Jak scheduler ocenia interaktywność zadań?

- sleep_avg – stosunek czasu spędzonego na spaniu do czasu wykorzystania procesora
- przyjmuje wartości od 0 do MAX_SLEEP_AVG (domyślnie 10ms)
- Kiedy zadanie przechodzi w stan gotowe do wykonania sleep_avg ustawiane jest na ilość czasu, jaką zadanie pozostawało w czasie oczekiwania zanim stało się gotowe.
- Każde tyknięcie zegara systemowego, w czasie gdy zadanie korzysta z procesora powoduje zmniejszenie wartości sleep_avg.
- Okazuje się, że ten sposób określania interaktywności jest bardzo miarodajny.

12. Jak wyliczany jest nowy kwant czasu?

- Skalowanie priorytetów na ilości przyznawanego czasu.
- Funkcja timeslice

Stan	**Kwant czasu**	**Interaktywność**	**Priorytet (wartość nice)**
początkowy	połowa ilości rodzica	nie wiadomo	taki jak rodzica
minimalny	10 ms	niska	wysoki
domyślny	100 ms	Średnia	0
maksymalny	200 ms	wysoka	niski

13. O procesach wybitnie interaktywnych

- Procesy bardzo interaktywne z chwilą wyczerpania swojego kwantu czasu nie zawsze są przenoszone do kolejki expired.
- TASK_INTERACTIVE() - określa, czy dane zadanie jest wystarczająco interaktywne
- EXPIRED_STARVING() - określa, czy przypadkiem czas od ostatniego przełączenia między kolejkami nie jest za długi
- Jeżeli pierwsze makro true a drugie false wtedy zadanie jest z powrotem umieszczane w kolejce active, w przeciwnym razie w expired.
- Dlaczego takie rozwiązanie – żeby nie niepokoić użytkownika zbyt długim czasem na oczekiwanie na odpowiedź systemu na naciśnięcie klawisza.

14. Usypianie i budzenie.

- Wszystkie zadanie, które są w stanie uśpienia (zablokowania) umieszczane są w osobnej kolejce.
- Po co?
- Żeby nie zajmowały procesora. Zadania, które są oznaczone jako SLEEPING nie są brane pod uwagę przez schedulera.

15. Jak?

```
DECLARE_WAITQUEUE(wait, current);
add_wait_queue(q, &wait);
set_current_state(TASK_INTERRUPTIBLE;
  /* lub UNINTERRUPTIBLE */
while (!warunek_pobudki)
  schedule();
set_current_state(TASK_RUNNING);
remowe_wait_queue(q, &wait);
```

16. Równoważenie międzyprocesorowe

- load_balance() - zdefiniowana w sched.c funkcja odpowiedzialna za równownomierne obciążanie procesorów
- Kiedy wywoływana:
- co 1 ms jeśli procesor pozostaje w stanie bezczynności
- co 200 ms w przeciwnym wypadku
- w systemach z jednym procesorem nigdy
- Zawsze z zablokowaną obsługiwaną kolejką i z wyłączonymi wszystkimi przerwaniami

17. Jak działa load_balance?

- find_busiest_queue() znajduje procesor z największą ilością zadań – jeśli żaden procesor nie ma co najmniej 25% więcej zadań niż bieżący zwracany jest NULL
- wybiera kolejką, z której zostaną przeciągnięte zadania – preferowana jest expired
- następnie load_balance() znajduje listę zadań o największym priorytecie
- każde zadanie ze znalezionej kolejki jest sprawdzane – tj. czy może zostać przeciągnięte na bieżący procesor – jeśli tak wywoływane jest dla niego pull_task()
- ostatnie dwa kroki są powtarzane tak długo, aż kolejki procesorów zostaną zrównoważone

18. Przełączanie kontekstu

- Obsługiwane przez funkcję context_switch() zdefiniowaną w kernel/sched.c. Wywoływana przez schedule() za każdym razem, gdy nowe zadanie zostało wybrane do wykonania. Składa się z dwóch podstawowych kroków:
- wywołanie switch_mm() (def. W include/asm/mmu_context.h) – do przełączania mapowania pamięci
- wywołanie switch_to() (include/asm/system.h) – przełączenie stanu procesora

19. Kiedy wywoływany jest schedule()

- Jeśli tylko zadania mogłyby jawnie wywoływać schedule() - mogłyby działać w nieskończoność.
- need_resched – flaga sygnalizująca konieczność wywołania schedule()
- Kto może ustawić:

  - scheduler_tick() - funkcja wywoływana, gdy zadanie wyczerpie swój kwant czasu

  - try_to_wake_up() - wywoływana, gdy proces o wyższym priorytecie niż obecnie wykonywany otrzymał sygnał pobudki
- Jądro sprawdza, że bit ustawiony i wywołuje schedule()
- Kiedy sprawdza?

  - powrót z funkcji systemowej

  - powrót z funkcji obsługi przerwania.

20. Wywłaszczanie procesów użytkownika

- Może nastąpić w czasie:
  - powrotu z funkcji systemowej
  - powrotu z funkcji obsługi przerwania
- Dlaczego?
- Bo wtedy właśnie jądro sprawdza, czy ustawiony jest bit need_resched i jeśli tak wywłaszcza proces

21. Wywłaszczanie jądra.

- Możliwość wywłaszczania jądra – jedna z najważniejszych cech jądra 2.6 – niespotykana w większości Unixów
- Jak to działa?
- Żeby wywłaszczyć zadanie jądra trzeba najpierw sprawdzić, czy nie ma ustawionych żadnych blokad – służy do tego specjalna zmienna **preempt_count**, którego wartość odpowiada ilości blokad założonych przez obecny wątek jądra (zmienna ta znajduje się w strukturze thread-info).
- Jeżeli ustawiona jest flaga need_resched i preempt_count jest równe zero wtedy można wywłaszczać.
- Za każdym razem, gdy zwolnienie przez jądro blokady powoduje wyzerowanie preept_count sprawdzana jest flaga need_resched.

22. Kiedy może dojść do wywłaszczania jądra?

- w czasie powrotu z obsługi przerwania do kodu jądra
- kiedy jądro zwalnia ostatnią blokadę
- jeśli jądro jawnie wywoła schedule()
- kiedy wątek jądra blokuje się w oczekiwaniu na jakieś zdarzenie

23. Szeregowanie zadań czasu rzeczywistego

- Zadania czasu rzeczywistego otrzymują priorytety od 0 do 99
- Priorytety pozostałych zadań skalowane są na przedział 100 – 140 (-20 -> 100, 18 -> 140)
- Dwie polityki szeregowania:
- SCHED_FIFO – procesy są obsługiwane zgodnie z zasadą first-in-first-out – procesy o wyższym priorytecie są obsługiwane przed tymi o niższych priorytetach – może dojść do głodzenia procesów o niskich priorytetach, brak przydziału kwantów czasu
- SCHED_RR – podobnie jak wyżej, z tym że procesy otrzymują kwanty czasu – po wykorzystaniu swojego przydziału zadanie trafia na koniec kolejki zadań o danym priorytecie
- SCHED_OTHER – starndardowa polityka obsługi pozostałych zadań
- Jądro w żaden sposób nie zmienia początkowego priorytetu zadań czasu rzeczywistego.

24. Jak można wpływać na zachowanie schedulera?

- MIN_TIMESLICE
- MAX_TIMESLICE
- PRIO_BONUS_RATIO – określa w procentach, o ile może się zmienić bazowy priorytet przy wyliczaniu dynamicznego priorytetu
- MAX_SLEEP_AVG – jak długo proces musi spać, żeby być uznanym za interaktywny
- STARVATION_LIMIT –