# Co nowego w jądrze 2.6

Adam Bielański
Michał Kistowski
Tomek Zdunowski

12 stycznia 2005

# Nowy scheduler

Linux 2.6 używa nowego algorytmu schedulera, zaprojektowanego przez Ingo Molnar'a. Jest to algorytm działający w czasie O(1) i powinien sprawować się odczuwalnie lepiej podczas dużego obciążenia. Jego zaletą jest również skalowalność ze względu na liczbę procesorów.

W scheduler'ze używanym w jądrze 2.4 algorytm przydzielający odcinek czasu procesora dla procesów, wymaga by wszystkie procesy już wykorzystały swoje odcinki czasu, zanim nowy przydział będzie dokonany. Konsekwencją tego jest fakt, że na systemach wieloprocesorowych większość procesorów jest bezczynna w czasie gdy procesy na nich się obliczające wykorzystają swoje odcinki czasu i czekają na wyliczenie nowych - to obniża wydajność systemów SMP. Ponadto, bezczynne procesory zaczynają wykonywanie oczekujących (gotowych) procesów, również gdy te procesy są przypisane do innego procesora, co sprawia, że procesy często zmieniają procesory na których są wykonywane. Kiedy takie zjawisko zdarza się procesowi o wysokim priorytecie, lub interaktywnemu - wydajność całego systemu spada...

Nowy scheduler rozwiązuje te problemy rozdzielając czas dla procesów dla każdego CPU osobno, eliminując globalną pętlę, w której przeliczane na nowo były odcinki czasu dla procesów i w konsekwencji następowała synchronizacja procesorów.

## Procesy klasy RT

Ta klasa procesów nie doczekała się wielkich zmian w nowym scheduler'ze. Nadal procesy w tej klasie mają priorytety od 0 do 99, im niższa wartość tym większy priorytet procesu. I nadal mają bezwzględne pierwszeństwo przed zwykłymi procesami. Mogą być szeregowane w ramach polityki SCHED_RR lub SCHED_FIFO.

## Zwykłe procesy

Zwykłe procesy doczekały się znacznie większych zmian. I głównie tym będziemy się teraz zajmować.

Główne zalety nowego scheduler'a w porównaniu do wersji 2.4 to:

1. **Wydajność systemów SMP:** Jeśli są procesy oczekujące, wszystkie procesory powinny pracować
2. **Procesy oczekujące:** Żaden proces nie będzie zagłodzony w oczekiwaniu na procesor, również żaden proces nie może zawłaszczać procesora na zbyt długi okres.
3. **Przypisywanie do procesorów w arch. SMP:** Procesory powinny być przypisane do jednego procesora i nie zmieniać go zbyt często.
4. **Priorytety:** Mniej ważne zadania powinny zaczynać wykonywanie z mniejszym priorytetem i vice versa.
5. **Zrównoważenie obciążenia:** Schedule powinien zmniejszać priorytet każdego procesu, który generuje więcej obciążenia niż procesor może przetworzyć.

6. **Wydajność procesów interaktywnych:** Z nowym scheduler'em użytkownik nie powinien zaobserwować opóźnionych reakcji systemu na zdarzenia typu naciśnięcie klawisza klawiatury, czy kliknięcie myszą, nawet wtedy, gdy system jest bardzo obciążony. Scheduler rozpoznaje zadania "interaktywne" i trochę inaczej przydziela im odcinki czasu.

Jak je osiągnięto:

- Priorytet zwykłego procesu określony przez nice jest tylko podstawą do wyliczenia dynamicznego priorytetu procesu. Priorytet dynamiczny wyliczany jest tak, by promować procesy interaktywne. O tym jak scheduler rozpoznaje interaktywne procesy (a raczej jakie procesy uznaje za interaktywne) więcej za chwilę.
- Algorytm używa dwóch tablic (na każdy procesor) do wyboru procesu o najwyższym priorytecie przy przełączaniu kontekstu, są to: tablica “active” i tablica “expired”. Dostęp do nich jest poprzez wskaźniki. Tablica active zawiera (posortowane) wszystkie procesy, które są przypisane do tego CPU i mogą jeszcze się wykonywać. Tablica “expired” zawiera posortowaną listę zadań, które wyczerpały swoje przydziały czasu. Jeśli tablica “active” stanie się pusta, wtedy następuje zamiana wskaźników do

tablic, przez co tablica “expired” (zawierająca procesy gotowe do wykonania) staje się nową tablicą “active” i vice versa. Przechowywana jest również 140-bitowa bitmapa, w której bity zapalane są jeśli są gotowe procesy do wykonania na tym CPU o priorytecie równym numerowi zapalanego bitu. Tablice active i expired, nieprzypadkowo również mają rozmiar 140 i w każdym ich polu znajduje się lista procesów o danym priorytecie. Pola od 0 do 99 przeznaczone są dla procesów klasy RT, a pola od 100 do 139 dla zwykłych procesów. Znakomicie przyspiesza to znajdowanie zadania o najwyższym priorytecie.

- Wyliczanie nowego odcinka czasu procesora dla procesu odbywa się od razu gdy bieżący odcinek się kończy. Pozwoliło to na uniknięcie pętli wyliczającej goodnes na końcu epoki.

- Wiele zmian doczekała się realizacja scheduler'a na architekturę SMP. Wersja z jądra 2.4 działała na SMP zbyt wolno (niewykorzystywanie wszystkich procesorów i przerzucanie zadań z jednego procesora na drugi), teraz spory fragment kodu w pliku sched.c poświęcony jest wyłącznie SMP. Rozbudowano procedury zarządzające równoważeniem obciążenia procesorów. Scheduler pamięta średnie obciążenie procesorów i przenosząc procesy pomiędzy procesorami dba o to, by nie przekroczyć tej średniej. Aby jednak przenoszenie procesów w ogóle miało miejsce równowaga musi zostać "istotnie" zachwiana, albo CPU wykonujący `schedule()` wywłaszcza właśnie ostatnie zadanie i zaraz będzie bezczynny - wtedy stara się "przyciągnąć" do siebie procesy oczekujące na innych CPU. Przenoszenie procesów wymaga synchronizacji procesorów, starano się

więc by było dokonywane jak najrzadziej.

- Zlikwidowano globalny `runqueue_lock` . W zamian utrzymywany jest mechanizm blokowania kolejki procesów na każdym procesorze, dzięki czemu procesy działające na różnych procesorach mogą być usypiane, budzone i zmieniane w pełni równolegle. Pętla przeliczająca “goodnes” i pętla wyznaczająca odcinki czasu dla procesów zostały wyeliminowane. Do znajdowania procesu o najwyższym priorytecie, żeby go obudzić, użyto algorytmu działającego w czasie O(1).
- Zadbano także o obsługę fork’a. W szczególności o to, by procesy o wysokim priorytecie nie tworzyły potomków również o wysokim priorytecie zbyt łatwo. Jeśli proces wykona fork’a zostaje “ukarany”, w ten sposób, że część czasu CPU przeznaczonego dla tego procesu jest oddawana potomkowi. Pociągnęło to za sobą konieczność

obsługi sytuacji, gdy potomek nie wykorzysta tego “daru” i np. się zakończy. Wówczas rodzic ma szansę odzyskać ten fragment czasu CPU. Dzięki temu poprawiono wydajność systemu w sytuacjach gdy jeden proces tworzy wielu potomków, z których część prawie natychmiast się kończy. Trzeba przyznać jednak, że nie ułatwia to czytania kodu scheduler’a :)

- Scheduler został oczywiście przystosowany również do wywłaszczalnego jądra. Jedynie niewielkie jego fragmenty są oznaczone jako niewywłaszczalne.
- Scheduler dostał również wiele poprawek kosmetycznych, pewne fragmenty zostały trochę uproszczone, inne są rozmieszczone w lepszych (zdaniem p. Molnara) obszarach kernel’a (to również nie ułatwia czytania kodu). Efektem tych wszystkich zabiegów jest zmniejszenie rozmiaru kodu nowego scheduler’a w odniesieniu do starej

wersji.

Wprowadzono termin “zadanie interaktywne”. Oznacza to takie zadanie, które nie wykorzystuje CPU do intensywnych obliczeń, tylko dobrowolnie “zasypia” po dostaniu procesora (np oczekując na zdarzenie I/O). Scheduler liczy czas tego “dobrowolnego spania” dla każdego zadania i premiuje te procesy, które śpią najwięcej przyznając im bonus do dynamicznego priorytetu (funkcja `recalc_task_prio )`i status zadania “interaktywnego”.
Jeśli zadanie jest “interaktywne”, wtedy wstawiamy je do tablicy “active” po wyczerpaniu przez nie aktualnego przydziału czasu CPU i dajemy mu nowy przydział czasu. Nie będzie wykonywane natychmiast ponownie, natomiast zostanie zastosowana strategia round-robin do niego i pozostałych zadań w tablicy active. Aby zapewnić, że nie zagłodzimy zadań znajdujących się w tablicy “expired” nie będziemy zadania

interaktywnego wstawiać do tablicy “active” jeśli pierwsze zadanie z tablicy “expired” miałoby czekać na CPU więcej niż “rozsądny” okres. Wielkość tego “rozsądnego” okresu jest zmienna w funkcji obciążenia CPU w taki sposób, że częstotliwość przełączania tablic maleje ze wzrostem ilości działających procesów. Zignorujemy również “interaktywność” zadania jeśli wcześniej wstawiliśmy do tablicy “expired” zadanie o większym statycznym priorytecie. W przeciwnym przypadku złamalibyśmy zasadę uprzywilejowującą zadania o wysokim statycznym priorytecie.
Parametr nice procesu ma wpływ na szanse by zadanie stało się interaktywne i tak np. proces mający nice +19 nigdy nie będzie “interaktywny”, z kolei tylko bardzo duże obciążenie procesora sprawi by zadanie o nice -20 zostało przeniesione do tablicy “expired”. Domyślny poziom nice 0 sprawia, że zadanie może, w niektórych sytuacjach, stać się “interaktywne”.

## Deklaracja tablicy priorytetowej:

```
#define MAX_PRIO    (MAX_RT_PRIO + 40) /* == 140 */
#define BITMAP_SIZE ((((MAX_PRIO+1+7)/8)+ \
                    sizeof(long)-1)/sizeof(long))
struct prio_array {
   int nr_active;
        /* liczba procesów oczekujących na CPU*/
   unsigned long bitmap[BITMAP_SIZE];
        /* mapa bitowa określająca czy jest oczekujący
           proces o danym priorytecie*/
   struct list_head queue[MAX_PRIO];
        /* tablica list procesów o
   odpowiednich priorytetach */
};
```

# Funkcja obliczająca dynamiczny priorytet zadania:

```
#define MAX_BONUS       (MAX_USER_PRIO * PRIO_BONUS_RATIO / 100)
#define MAX_RT_PRIO     100
static int effective_prio(task_t *p)
 {
        int bonus, prio;

        if (rt_task(p))
                return p->prio;

        bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;

        prio = p->static_prio - bonus;
        if (prio < MAX_RT_PRIO)
                prio = MAX_RT_PRIO;
        if (prio > MAX_PRIO-1)
                prio = MAX_PRIO-1;
        return prio;
 }
```

Widać, że priorytet statyczny zadania klasy RT się nie zmienia, a zwykłego zadania może się zmienić, ale nie może wyjść z przedziału [MAX_RT_PRIO..MAX_PRIO-1]

# Zmiany w modułach kernela

- Zmiana rozszerzenia modułu z “.o” na “.ko” ( *kernel object* )
- Można wyłączyć możliwość odłączania (unload) modułów, jeśli istotniejsza dla nas jest stabilność systemu od jego wydajności.
- Praktycznie obowiązkowe stało się korzystanie z makr module_init(funkcja_inicjująca_moduł) i module_exit(funkcja_czyszczaca_moduł). Wprawdzie te makra istniały w wersji jądra 2.4, ale często programiści definiowali funkcje inicjującą i czyszczącą o standardowych nazwach (inid_module() i cleanup_module()) nie korzystając z makr. Teraz można wprawdzie też uniknąć stosowania tych makr, ale natkniemy się na ostrzeżenia kompilatora.

- Większy nacisk położono na zamieszczanie informacji o typie licencji modułu, za pomocą makra MODULE_LICENSE(nazwa_licencji). Znowu - to makro było już w wersji jądra 2.4, ale teraz nie stosowanie go powoduje pojawienie się komunikatu: “module license ’unspecified’ taints kernel.” w logu systemowym. Oprócz tego, bez licencji GPL moduł nie będzie miał dostępu do niektórych symboli jądra.
- Zmieniono domyślne zachowanie modułu dotyczące eksportowania symboli. Do tej pory domyślnym zachowaniem było eksportowanie wszystkich symboli z modułu, w jądrze 2.6 aby wyeksportować symbol, trzeba explicite użyć makra EXPORT_SYMBOL(nazwa). Z kolei każdy użytkownik symbolu danego modułu musi przed pierwszym użyciem tego symbolu użyć try_module_get(&moduł).

- Napisano od nowa podsystem sprawdzania wersji modułu przy instalacji. Numer wersji jądra pod którą był tworzony moduł jest teraz dołączany tylko raz do modułu, w sekcji ELF przy kompilacji i nie jest widoczny dla modułu jako zmienna. Dołączana jest także informacja o statusie SMP, wywłaszczania kernela i wersji kompilatora, aby wychwycić więcej sytuacji gdy moduł mógłby się okazać niekompatybilny.
- Pojawiło się makro do tworzenia aliasów modułów. Wystarczy w kodzie modułu zamieścić linię: MODULE_ALIAS(”nazwa_aliasu”) ;
- Poprawiono zarządzanie licznikami odwołań do modułów. Teraz każdy, kto odwołuje się do modułu musi najpierw użyć funkcji `int try_module_get(&module)` . Może to również dotyczyć samego modułu, w pewnych rzadkich przypadkach. Wartość zwracana try_module_get jest

równa zero gdy próba się nie powiedzie (np. gdy moduł jest właśnie wyrzucany z systemu). Funkcja module_put() zwalnia moduł. Aby umożliwić “siłowe” wyrzucenie modułu, trzeba w nim włączyć opcję CONFIG_MODULE_FORCE_UNLOAD. Oczywiście ta opcja powinna być włączona jedynie w wersjach testowych modułów.

- Usunięto makro MODULE_PARM(). W jądrze od wersji 2.6 aby moduł mógł przyjmować parametry, musi być do niego dołączony plik linux/moduleparam.h . Same parametry deklaruje się makremmodule_param(nazwa, typ, perm) gdzie ’nazwa’ jest nazwą parametru (i zmiennej przechowującej jego wartość), ’typ’ jego typem, a ’perm’ - uprawnieniami, które będą się odnosiły do pozycji w sysfs odpowiadającej parametrowi. Jest też kilka wyspecjalizowanych funkcji do konkretnych rodzajów parametrów:

- Jeśli parametr ma inną nazwę niż zmienna, która będzie przechowywać jego wartość, możemy skorzystać z makra module_param_named(nazwa, wartość, typ, perm) gdzie 'nazwa' jest nazwą zewnętrzną, a 'wartość' nazwą zmiennej wewnętrznej.

- Parametry w postaci łańcucha znaków można deklarować przez typ charp, albo makrem module_param_string(nazwa,string,długość,perm) . Długość zwykle najlepiej określić jako sizeof(string).

- Parametry postaci listy oddzielonej przecinkami można wczytać jako tablicę, używając makra module_param_array(nazwa,typ,ilość,perm) , gdzie 'ilość' zostanie przez makro ustawiona na ilość elementów parametru aktualnego. Makro sprawdza rozmiar tablicy 'nazwa' i kopiuje nie więcej elementów niż tyle ile może się zmieścić w tablicy.

# Więcej...

Wewnętrzne ograniczenia Linux'a zostały również powiększone, tam gdzie to możliwe. Zniesiono ograniczenie maksymalnej ilości równocześnie otwartych plików. Teraz ta liczba dynamicznie zmienia się w czasie działania systemu. Pozostałe ograniczenia pozostały, ale zwiększono:

- Ilość użytkowników i grup w systemie - z ok. 65 000 do ponad 4 mld (z 16-bitów do 32)
- Ilość PIDów - z ok. 32 000 do ponad miliarda.
- Maksymalną objętość systemów plików do 16 TB
- Maksymalną ilość typów urządzeń (MAJORDEV) - z 255 do 4096
- Maksymalną ilość urządzeń poszczególnego typu (MINORDEV) - z 255 do ponad miliona

Nowe ograniczenia nie powinny być krępujące przynajmniej do czasu pojawienia się wersji 3.0 :-)

# Trochę kosmetyki

Zmieniono obsługę nagrywarek CD-RW podłączanych do magistrali IDE. Teraz zapis jest dokonywany bezpośrednio przez sterownik IDE, zamiast przez pośredniczący i emulowany sterownik SCSI. Pomijając kwestię wydajności, jest to znacznie bardziej elegancka implementacja.
Dołączono również wsparcie dla standardu Serial ATA (S-ATA) o transferze przekraczającym 150 MB/s

# Zmiany w systemach plików

**Systemy lokalne**

W naturalnych dla Linux'a systemach plików - ext2 i ext3, wprowadzono wsparcie dla rozszerzonych atrybutów plików, które (atrybuty) mogą być przechowywane wewnątrz systemu plików, ale na zewnątrz pliku. Oznacza to na początek konieczność update'u takich narzędzi jak 'tar' zanim będzie można skorzystać z tych nowinek :) . Oprócz tego, pozwoliło to na zaimplementowanie kontroli dostępu do plików zgodnie ze standardem POSIX (ACL), który rozszerza standard UNIX'owy i pozwala na bardziej subtelną kontrolę. W ext3 dodano również opcję dostosowania 'journal commit time', co ma znaczenie dla użytkowników laptopów.

Również domyślne opcje montowania systemu plików mogą być przechowywane w systemie plików, dzięki czemu nie trzeba ich przekazywać przy każdym montowaniu. Można również oznaczyć katalog jako “indeksowany”, co pozwoli przyspieszyć wyszukiwanie w nim plików. Poprawiono również wsparcie dla

systemów plików mniej “linuksowych”. Na przykład dla XFS. Teraz dysk sformatowany jako XFS może być root-disk’iem. Jest w nim również wsparcie dla rozszerzonych atrybutów i ACL.

Zadbano również o miłośników dominującego systemu i wprowadzono również poprawki zwiększające kompatybilność z systemami plików Windows’a. Nowy kernel obsługuje nowy schemat tablicy partycji Windowsa, tzw. “Woluminy dynamiczne”, które pojawiły się w wersji 2000. Jest jednak zupełnie nieprawdopodobne, by ten schemat stał się

standardem Linux'a :)

Napisano także nowy moduł, stopniowo ulepszany, obsługujący system plików NTFS, który teraz można zamontować w trybie rw. Pamiętać należy, że tryb zapisu na NTFS jest jeszcze wciąż trochę... eksperymentalny.

Pojawiło się również wsparcie dla rzadko używanego systemu FAT12, używanego przez DOS i to nie w ostatnich wersjach, który ostatnio wrócił do użytku w niektórych odtwarzaczach MP3.

Również dla systemu plików OS/2, czyli HPFS, poprawiono kompatybilność. Wprowadzono także wiele drobnych

poprawek, na przykład obsługa Quota'y została napisana od nowa, by obsługiwała zwiększoną liczbę użytkowników systemu, można zaznaczyć katalogi jako synchroniczne, dzięki czemu zmiany wewnątrz nich będą natychmiastowe.

Wprowadzono “przezroczystą kompresję” dla systemu plików ISO9660.

## Systemy sieciowe

Pojawiło się wsparcie dla nowego NFSv4 zarówno dla klientów jak i dla serwerów. NFSv4 obsługuje bezpieczniejszą weryfikację tożsamości i szyfrowanie, inteligentniejsze blokowanie plików, wsparcie dla pseudo-systemów plików. Jedynym mankamentem NFSv4 jest to, że wsparcie dla niego jest eksperymentalne... Pocieszające jest zapewnienie, że jest ono również “relatywnie stabilne”. Dodatkowo, serwer NFS dla Linux’a został poprawiony, aby być bardziej skalowalnym (64 razy więcej użytkowników i dłuższe kolejki żądań), obsługiwać również protokół TCP oprócz UDP i łatwiej zarządzany (przez nowy system plików ’nfsd’, zamiast przez wywołania systemowe).

Wprowadzono również wiele poprawek do sieciowych systemów plików używanych w dominującym systemie operacyjnym. Standardowy współdzielony system plików

Windows, SMB (wspierany przez Linux'a), został zastąpiony przez CIFS (Common Internet File System), używany przez Windows 2000 i późniejsze. CIFS jest zestandaryzowaną i poszerzoną wersją SMB. Nowa wersja kernela zawiera zupełnie nowy moduł do obsługi systemów plików CIFS, pozwalający również na dostęp do typów plików nie występujących w Windows'ach (takich jak pliki urządzeń i dowiązania symboliczne) na takich serwerach jak Samba. Możliwy jest również dostęp do rozszerzeń SMB takich jak SMB-UNIX.

Pojawiło się również wsparcie dla stosunkowo nowych sieciowych systemów plików takich jak CODA, AFS i InterMezzo, w których pojawia się lokalne cache'owanie plików z systemów zdalnych, rozpraszanie pliku pomiędzy wieloma węzłami itp.

# Linux dla mikrourządzeń

Do głównego nurtu Linux'a dołączono w wersji 2.6 projekt uClinux, czyli Linux dla mikrokontrolerów. Główną różnicą między 'normalnym' komputerem, a mikrourządzeniem jest to, że procesory mikrourządzeń nie mają jednostki zarządzania pamięcią (MMU), która pozwala na dostęp do pamięci w różnych trybach. Są to zatem systemy wielozadaniowe bez ochrony pamięci i innych cech związanych z tym. Można zatem w takich systemach ingerować w pamięć innych procesów :) Jest to generalnie rozwiązanie niewskazane dla systemów z wieloma użytkownikami, ale świetnie się sprawdzające w urządzeniach typu PDA. Dodatkowo pojawiła się możliwość postawienia systemu zupełnie bez partycji wymiany (swap). Przykładowe procesory tego typu to Motorola m68k, NEC v850 i Hitachi H8/300.

Można powiedzieć, że wersja 2.6 jądra zrywa (przynajmniej częściowo) z ograniczeniami dziedziczonymi przez poprzednie wersje z pierwszego Linux'a chodzącego na i80386.

# Wywłaszczalne jądro

RTOS (Real Time Operating System) oznacza system operacyjny czasu rzeczywistego gdzie każde przerwanie zostaje obsłużone w pewnym krótkim czasie, zazwyczaj ograniczonym przez pewną stałą (zazwyczaj <10ms). Są to systemy w których liczy się czas reakcji na zdarzenie. Wywłaszczanie jądra dostępne w jądrze 2.6 jest zdecydowanym krokiem w kierunku systemów RTOS.

Wywłaszczalne jądro oznacza możliwość wywłaszczenia procesu działającego w trybie jądra, tzn. przerwanie wykonywania danego procesu i wznowienie w przyszłości, a w między czasie może zostać obsłużony inny proces ( Schedule sie tym zajmuje). Dotychczas możliwe było wywłaszczanie procesów tylko w trybie użytkownika, a proces działający w trybie jądra działał dopóki sam się nie zakończył lub dobrowolnie zrzekł sie procesora.
Zmiany dokonane w jądrze 2.6 w kierunku RTOS'ow:

- opisane tu Wywłaszczalne jądro
- stały czas działania algorytmu Schedule
- drobnoziarnistość sekcji krytycznej procesów, dodanie preemption point'ów w których proces może zostać wywłaszczony

Istnieje możliwość zabronienia wywłaszczenia danego kodu, w tym celu należy użyć funkcji: `preempt_disable()` oraz `preempt_enable()`. Kod zawarty między tymi instrukcjami nie zostanie wywłaszczony. Każdy proces posiada licznik zagnieżdżonych wywołań `preempt_disable()` - `preempt_count`. Wywołanie `preempt_disable()` zwiększa ten licznik, a `preempt_enable()` zmniejsza o `1`. Wywłaszczenie może nastąpić tylko wtedy gdy `preempt_count == 0`.

# Co nie zostanie wywłaszczone

- Ze względów logicznych nigdy nie zostanie wywłaszczony Scheduler oraz min. handlery przerwań.
- Nie zostanie również wywłaszczony kod z zablokowaną obsługą przerwań.
- Sekcja krytyczna spin-locków gdyż jest zintegrowana z `preempt_disable` i `preempt_enable`.

# Problemy związane z wywłaszczaniem

- w architekturze SMP wywłaszczanie kodu który używa danych prywatnych procesora nie jest możliwe ponieważ proces może zostać wznowiony na innym procesorze niż był dotychczas wykonywany, kod takiego procesu powinien być zawarty pomiędzy preempt_disable() i preempt_enable().
- rejestry FPU nie są zachowywane gdy proces w trybie jądra zostanie wywłaszczony, kod używający FPU również nie może zostać wywłaszczony.

W jakich zastosowaniach ważną rolę odgrywają systemy RTOS

- systemy monitorujące gdzie czas reakcji na nieprawidłowość odgrywa ważną rolę (elektrownie/fabryki)
- obsługa urządzeń I/O (np. modemy DSL które do poprawnej pracy wymagają procesora)

# Prawdziwe RTOS-y

- VxWorks
- QNX
- KURT-Linux
- Windows CE
- Palm OS

# Zarys historyczny

Już w jądrze 2.4 z łatka RML preempt było możliwe wywłaszczanie procesów w trybie jądra, później wersja testowa 2.5.4 i na końcu włączenie do oficjalnego jądra 2.6, ale nadal jest to opcjonalne (tzn. można wyłączyć podczas kompilacji jądra).

# Dokumentacja

`http://www.kernel.org/` - The Linux Kernel Archives
`http://www.kniggit.net/wwol26.html` - The Wonderful World of Linux 2.6 (Joseph Pranevich)
`http://www.linuxdevices.com/articles/AT5152980814.h` - Can Linux be a real-time operating system?
`http://lwn.net/Articles/driver-porting/` - Driver porting: the preemptible kernel
`http://www.tekla.neostrada.pl/prog/rtos/` - Systemy operacyjne czasu rzeczywistego

# Obsługa NUMA i Hyperthreading

Jedna z fundamentalnych zmian w nowej wersji jądra 2.6 prowadząca do pełnego wykorzystania Linuxa w zastosowaniach serwerowych jest obsługa NUMA. NUMA (“Non-Uniform Memory Access”) architektura wieloprocesorowa która powstała z połączenia SMP (Symmetric MultiProcessing) i MMP (Massively Parallel Processors). W technologii SMP wszystkie procesory korzystają z zasobów systemu, czyli pamięci operacyjnej i przestrzeni I/O, w jednakowym trybie i na równych prawach. Skalowalność tej architektury jest mocno ograniczona w głównej mierze przez dwa czynniki:

- przepustowość magistrali (każdy procesor z niej korzysta)
- problem spójności pamięci podręcznej (cache) (aktualność danych w pamięci systemowej)

Rozwiązaniem tych problemów jest architektura NUMA. W tej technologii każdy procesor (lub 2-4-procesorowy węzeł SMP) dysponuje własną, fizyczną pamięcią RAM, mając równocześnie możliwość dostępu do całej przestrzeni adresowej systemu (ale z istotnym opóźnieniem). Problem przepustowości magistrali został wyeliminowany, ponieważ odwołania do wspólnej pamięci nie są już tak częste. Dzięki tej technologii system jest znacznie bardziej skalowalny, dla technologii SMP wąskim gardłem jest 8 procesorów, systemy oparte o większą liczbę procesorów są nieefektywne. Tymczasem w architekturze NUMA buduję się systemy oparte o 16, 32 i więcej procesorów i są to bardzo efektywne systemy. Implementacja: Optymalna topologia rozmieszczenia pamięci

względem procesorów jest dostarczana przed podsystem VM (Virtual Memory). Optymalnym rozwiązaniem dla procesu jest przydzielenie mu pamięci w jednym węźle oraz procesora z tego węzła (patrz Scheduler).
`http://lse.sourceforge.net/numa/` - Linux Support for NUMA Hardware
`http://www.kniggit.net/wwol26.html` - The Wonderful World of Linux 2.6 (Joseph Pranevich)

# Hyperthreading

HypertTreading (HT) (hiperwątkowść) została wprowadzana przez firme Intel i tylko przez nią jest wykorzystywana w procesorach Pentium IV. Technologia HT umożliwia wykonywanie wielu wątków danego procesu równolegle na jednym procesorze. Procesor zostaje “podzielony” na 2, 4 (zależnie od architektury) wirtualne procesory i system “wygląda” jakby miał zamiast wirtualnych prawdziwe procesory. Obsługa HT odbywa się na poziomie sprzętowym systemu. Mimo, że wirtualne procesory dzielą jeden fizyczny procesor, to takie rozwiązanie przynosi często efekt lepszej wydajności systemu (nawet do 40% dla niektórych zastosowań). Obsługa HT wiąże się z dodatkową komplikacją algorytmu Schedulera. Obsługa HT przez jądro jest możliwa dzięki temu, że Scheduler prawidłowo rozpoznaje obciążenie zarówno procesora rzeczywistego jak i wirtualnego.

Dokumentacja:
`http://www.kernel.org/` - The Linux Kernel Archives
`http://www.kniggit.net/wwol26.html` - The Wonderful World of Linux 2.6 (Joseph Pranevich)

# Unified Device Model

Mnogość dostępnego sprzętu komputerowego oraz nieustanny rozwój tej branży wymusiło na nowym jądrze ujednolicenie i ustandaryzowanie zarządzania urządzeniami podłączonymi do komputera. Powstał ujednolicony model urządzenia (Unified Device Model) dostępnego w systemie. Unified Device Model jest bezpośrednio związany z SysFs,

Unified Device Model jest odpowiedzialny:

- Jakie urządzenia są podłączone do komputera i w jakim stanie aktualnie się znajdują, do jakiej magistrali są podłączone oraz które sterowniki są odpowiedzialne za obsługę tych urządzeń.
- Struktura dostępnych w systemie szyn, jakie są zależności pomiędzy szynami (np. USB controler może być podłączony do szyny PCI).
- Specyfikacja sterowników: z jakimi urządzeniami dany sterownik może współpracować oraz jakie szyny obsługuje
- Hierarchie urządzeń dostępnych w systemie. Każde urządzenie jest obiektem pewnej klasy (np. bus, usb). Dodatkowo można odpowiedzieć na pytanie czy dane urządzenie jest podłączone do komputera, np. myszka. Nie interesuje nas techniczna strona (czy ps/2, usb) tylko

sam fakt istnienia myszki w systemie. To właśnie zapewnia Unified Device Model.

# SysFs

Wirtualny system plików przedstawiający w postaci hierarchii katalogów wszystkie urządzenia dostępne w systemie. SysFs jest zazwyczaj zamontowany jako /sys . Podfoldery w /sys odpowiadają głównym podsystemom w systemie. Przykładowo:
`/sys/block/hda/device` odpowiada pierwszemu dyskowi twardemu;
`/sys/bus/usb/devices` odpowiada urządzeniu podłączonemu do USB.

# Kobject (Kernel Object)

Wypada wspomnieć o tej strukturze danych odpowiadającej za reprezentacje danych każdego obiektu w systemie. Kobject'y są wzajemnie powiązane tworząc hierarchie klas obiektów dostępnych w systemie. Jest to podstawowa struktura danych z której korzysta Unified Device Model i SysFs.

Implementacja:

```
struct kobject {
        char                    *k_name;
        char                    name[KOBJ_NAME_LEN];
        struct kref             kref;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct dentry           *dentry;
};
```

Każdy obiekt posiada nazwę char *kname, licznik odwołań kref, rodzica i należy do pewnego zbioru kset.

# Hotplug

Natychmiastowa obsługa urządzeń podłączanych do komputera. Jądro zajmuje się wykryciem typu urządzenia i odpowiednich sterowników oraz zainstalowaniem urządzenia w systemie. Takimi urządzeniami mogą być: USB klawiatura, modem. Obsługa tej technologii byla już zaimplementowana w jądrze 2.4 ale w 2.6 została poprawiona i rozszerzona klasa urządzeń obsługiwanych.

Dokumentacja:
`http://www.win.tue.nl/ aeb/linux/lk/lk-12.html` - The Linux Kernel (Andries Brouwer)
`http://lwn.net/Articles/51437/` - Porting device drivers to the 2.6 kernel
`http://linux-hotplug.sourceforge.net/` - Linux Hotplugging

# Audio i Multimedia

Zostały wprowadzone sterowniki ALSA (Advanced Linux Sound Architecture) jako domyślny standard dzwięku w miejsce przestarzałego, choć ciągle stosowanego jeszcze podsystemu OSS (Open Sound System). Nowy standard został wprowadzony w celu wyeliminowania częstych problemów ze starymi sterownikami kart dźwiękowych.

- Obsługa wielu kart dźwiękowych różnego typu w jednym komputerze.
- Full-Duplex odtwarzania i zapisu dźwięku.
- Łączenie strumieni audio.

Została poprawiona obsługa strumieni wideo za co jest odpowiedzialny podsystem Video4Linux (V4L) w nowym jądrze została poprawiona obsługa tego systemu.
Została dodana bądź ulepszona obsługa urządzeń:

- karty radiowe i telewizyjne
- karty wideo do obróbki sygnału wideo z różnych źródeł (kamery, itp)
- urządzenia audio/wideo podłączone pod USB lub midi (min. kamery internetowe).

# Obsługa Wireless Devices

Pełna obsługa urządzeń do komunikacji bezprzewodowej:

- IrDA (Infrared Data Associates group), była już w jądrze 2.4 ale teraz jest znacznie lepsza obsługa.
- Bluetooth, protokół używany w nowych urządzeniach przenośnych (telefony, laptopy, drukarki, urządzenia PDA)
- Wireless LAN, sieć LAN oparta o komunikację bezprzewodową.

`http://www.kniggit.net/wwol26.html` - The Wonderful World of Linux 2.6 (Joseph Pranevich)

# Futeksy

Jądro 2.6 niesie ze sobą wiele usprawnień dotyczących obsługi programów wielowątkowych. Powstawanie biblioteki NPTL (Native POSIX Threads Library) było ściśle powiązane z powstaniem odpowiedniej funkcjonalności po stronie jądra.

Poprzednia implementacja wątków zgodnych ze standardem POSIX w Linuksie – LinuxThreads miała poważne ograniczenia, m. in.

- Nie było dobrego mechanizmu synchronizacji wątków – do implementacji np. muteksów trzeba było wykorzystywać sygnały, co było bardzo kosztowne.
- Obsługa sygnałów była nieprawidłowa, np. sygnały **SIGSTOP**, **SIGCONT** były wysyłane tylko do jednego wątku.
- Liczba wątków w procesie była ograniczona do 8192.
- Niektórymi operacjami, np. tworzeniem nowych wątków zajmował się dodatkowy wątek zarządzający, który był wąskim gardłem na SMP.

Wynikało to z ograniczeń jądra 2.4. Okazało się, że nie da się ich ominąć w przestrzeni użytkownika.

Nowa implementacja wątków została oparta na modelu 1:1 – jednemu wątkowi użytkownika odpowiada jeden wątek jądra. Nie ma więc problemów z wątkiem zarządzającym.

Problem z dostarczaniem sygnałów do programów wielowątkowych został rozwiąznany dzięki przeniesieniu obsługi takich przypadków z biblioteki do jądra. W nowej implementacji obsługa sygnałów jest całkowicie zgodna ze standardem POSIX. Sygnały błędów oraz sygnały **SIGSTOP**, **SIGCONT** są dostarczane do całego procesu, a nie jednego z wątków.

W jądrze 2.6, w celu ułatwienia synchronizacji procesów, wprowadzono *futeksy*. Futex (skrót od **F**ast **U**serspace Mu**tex**) jest podstawowym mechanizmem służącym do budowania szybkich semaforów i podobnych mechanizmów.

Podstawową zaletą futeksów jest to, że w przypadku, gdy żaden proces nie czeka na semaforze, wszystkie operacje dzieją w przestrzeni użytkownika – nie potrzeba żadnych wywołań systemowych.

Futeksy są podstawowymi elementami, z których można budować bardziej złożone abstrakcje. Zostały zaprojektowane z myślą o wydajności, nie wygodzie użycia. W normalnych warunkach lepiej korzystać z opartych na nich gotowych blibliotek, np. właśnie NPTL.

# Implementacja

Tak naprawdę, futex jest atomowym licznikiem w obszarze pamięci dzielonym przez kilka procesów. W najprostszym przypadku wartości licznika mają następujące znaczenie:

- 1 nikt nie czeka, semafor otwarty
- 0 nikt nie czeka, semafor zamknięty
- $< 0$ ktoś czeka.

# Implementacja - *down*

Operacja *down* działa następująco:

- licznik jest atomowo zmniejszany,
- Jeśli wynik jest niezerowy to:
  - licznik jest ustawiany na -1
  - jest wykonywana operacja **FUTEX_WAIT**, która usypia bieżący proces. Proces może być później obudzony przy wywołaniu **FUTEX_WAKE**.

# Implementacja - *up*

Symetrycznie, operacja *up* polega na zwiększeniu licznika i w przypadku wyniku różnego od 1 ustawieniu licznika na 1 i wykonaniu operacji **FUTEX_WAKE**.

Możliwe jest też oczekiwanie z timeoutem i asynchroniczne oraz budzenie więcej niż jednego procesu.

W obu implementacjach jako rejestry wątku, służące do dostępu do TLS (Thread Local Storage), na IA-32 wykorzystywane były rejestry segmentu. Aby działało to prawidłowo, każdy wątek musiał mieć własny wpis w tablicy deskryptorów (w LinuxThreads w LDT, w NPTL w GDT). Ograniczało to ilość wątków procesu do 8192. W jądrze 2.6

wpisy w GDT są odpowiednio aktualizowane przez scheduler, co znosi to ograniczenie. Do przydzielania wpisów w GDT służy nowe wywołanie systemowe.

W jądrze 2.6 dodano nowe wywołanie `exit_group`, kończące wszystkie wątki procesu. Również `exec` powoduje zakończenie wszystkich wątków przed uruchomieniem nowego programu.

W katalogu `proc` w nowym jądrze jest widoczny tylko jeden wątek programu wielowątkowego, reprezentujący cały proces. Statystyki zwracane przez jądro, jak np. zużycie czasu procesora, również dotyczą wszystkich wątków w sumie.

Dzięki wszystkim tym usprawnieniom utworzenie i zakończenie 100 000 wątków na jądrze 2.6 z biblioteką NPTL trwa 2 sekundy w porównaniu z 15 minutami na starszych wersjach.

# Gdzie można przeczytać więcej

`http://people.redhat.com/drepper/futex.pdf` - “Futexes are tricky”
`http://people.redhat.com/drepper/nptl-design.pdf` - Native POSIX Threads Library

# Reverse mapping VM

Zanim jakaś strona z pamięci zostanie wymieniona na dysk, musi zostać oznaczona jako niedostępna w tablicy stron. Ponieważ ta sama strona może być widziana przez kilka procesów i to pod różnymi adresami, znalezienie odpowiednich wpisów wymagało w jądrze 2.4 przeszukania wszystkich tablic stron wszystkich procesów.

W jądrze 2.6 struktura `page`, opisująca stronę zawiera pole `chain` type `pte_chain`, które zawiera listę wszystkich wpisów w tablicach stron, dotyczących danej strony. Zwiększenie

wydajności wymieniania danych na dysk wiąże się jednak z kosztem w postaci dodatkowej pamięci potrzebnej do przechowywania łańcuchów PTE.