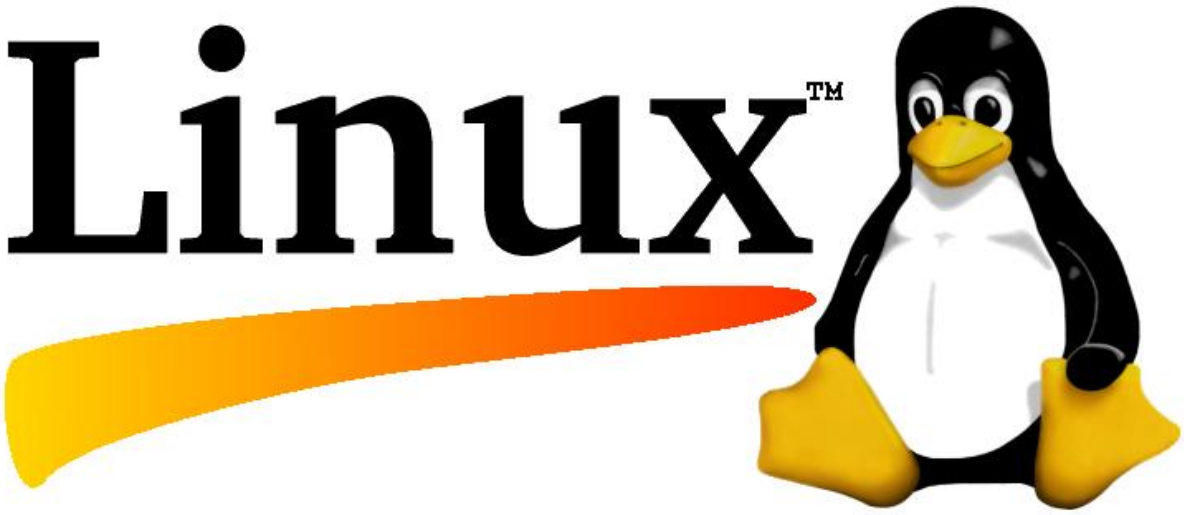


Co nowego w jądrze 2.6?

Autor:	Ryszard Beczek, Cezary Tkaczyk, Michał Wójcikowski
Data ostatniej zmiany:	21 stycznia 2005
Info:	W dokumencie została wymieniona większość najważniejszych nowości wprowadzonych w wersji 2.6 jądra Linuxa. Częściowo opisuje też aspekty, których zmiany te dotyczą.

Podstawowe informacje o dokumencie

Autor:	Ryszard Beczek, Cezary Tkaczyk, Michał Wójcikowski
Status:	roboczy
Tytuł:	Co nowego w jądrze 2.6?
Projekt:	SO
Utworzenie:	5 stycznia 2005 roku
Ostatnie zmiany:	21 stycznia 2005
Liczba stron:	52+2

Spis treści

1 Wprowadzenie

Minęły już cztery lata od kiedy ukazała się wersja 2.4 jądra linuxa. Najwyższy czas zatem na zmiany. Już od roku mamy nową wersję 2.6. Co nowego przyniosły prace nad obecną wersją? Krótki zarys nowości został zawarty właśnie w tej pracy.

1.1 Historia

Projekt jądra linuxa zapoczątkował Linux Torvalds w 1991 roku jako system operacyjny Minix dla komputera 386. Pierwsza oficjalna wersja Linuxa 1.0 była w marcu 1994, ale obsługiwała jedynie jednoprocesorowe maszyny i386. Rok później powstała wersja 1.2, która obsługiwała wiele platform (w szczególności: Alpha, Sparc,and Mips). Kolejnym krokiem była obsługa wieloprocesorowości wprowadzona w Linux 2.0 w czerwcu 1996.

Po wersji 2.0 kolejne podwersje powstawały już nieco wolniej (2.2 w styczniu 1999, 2.4 w styczniu 2001). Obie obsługiwały coraz to większą ilość sprzętu i platform. W szczególności wersja 2.4 jest uważana za przełomową dla komputerów typu desktop, ze względu na obsługę ISA Plug-and-Play, USB, PC Card i wiele innych.

Ostatecznie, 17 grudnia 2003 powstała wersja 2.6. Znów traktowana jako wielki skok, tym razem w kierunku zarówno dużo większych systemów, jak i tych mniejszych (takich jak np. PDA).

Dziś, czyli 14.I.2005, mamy stabilną wersję 2.6.10.

W skrócie historia wersji jądra i ich rozmiarów wygląda tak:

nr wersji	data	linie kodu
0.01	09/1991	7.5K
1.0	03/1994	158K
1.2	03/1995	277K
2.0	07/1996	649K
2.2	01/1991	1536K
2.4	01/2001	2888K
2.6	12/2003	5200K

1.2 Mapa pracy

2 Obsługiwane platformy sprzętowe -
Opis nowych architektur sprzętowych i rozwiązań hardwarowych obsługiwanych w nowym jądrze.

3 Procesy -
Nowości w obsłudze procesów przez jądro - Scheduler O(1), Wywłaszczanie jądra.

4 Odwrotne odwzorowanie stron -
Opis implementacji odwrotnego odwzorowania stron (reverse mapping).

5 Urządzenia -
Nastąpiły duże zmiany w obsłudze urządzeń w jądrze 2.6. Rozdział jest najobszerniejszym w całej pracy. Opisano w nim m.in. Unified Device Model, Asynchroniczne we/wy, Blokowe we/wy, nowy sposób konstruowania i kompilowania modułów.

6 Nowe Obsługiwane urządzenia i funkcje -
W rozdziale wymieniono niektóre urządzenia i funkcje, których obsługa pojawiła się dopiero w jądrze 2.6.

7 Systemy plików -
Rozdział traktuje o nowo wprowadzonych systemach plików, o ulepszeniach już istniejących (w szczególności o EA i ACL).

8 Inne nowości -
Inne aspekty nowego jądra.

9 Porady praktyczne -

2 Obsługiwane platformy sprzętowe

Największą siłą linuxa jest jego elastyczność i wsparcie dla obsługi wielu platform. W jądrze 2.6 można zanotować kilka nowości.

2.1 Embeded systems

Czym są systemy wbudowane (Embeded systems)? Niektórzy mówią tak o systemach, które mają ograniczoną interakcję z człowiekiem, ale nie jest to do końca prawdziwe. Można powiedzieć, że są to architektury, którym brakuje pewnych aspektów (np. jednostki zarządzającej pamięcią - MMU), albo mają inne rozwiązania sprzętowe (np. ARM, MIPS, PPC), albo mają inne wymagania stawiane systemowi operacyjnemu (np. potrzebują systemu czasu rzeczywistego). Przykłady embeded systems:

Linux robi znaczące postępy w obsłudze takich urządzeń. Dzięki licencji GPL, każdy może przystosować Linuxa do swojego PDA, palmtopa, czy przenośnego urządzenia, ściągając jądro i potrzebne do tego aplikacje z internetu. Pojawiło się więc bardzo wiele zarówno komercyjnych, jak i darmowych dystrybucji:

RTLinux - Real-Time Linux, komercyjny;

uCLinux - Linux dla urządzeń bez MMU;

Montavista Linux - Linux dla ARM, MIPS, PPC, komercyjny;

ARM-Linux

FREESCO i Linux Router Project - Linuxy na jednej dyskietce, nadają się do robienia prostych serwerów, mostów, routerów.
`http://www.freesco.org`,
`http://www.linuxrouter.org`

Linux On A Floppy (LOAF) - linux na jednej dyskietce dla słabszych komputerów, np 386.
`http://loaf.ecks.org`

Do oficjalnej wersji jądra wprowadza się rozwiązania z takich dystrybucji. Akurat w jądrze 2.6 najwięcej zapożyczono z **uCLinux**.

2.1.1 Źródła

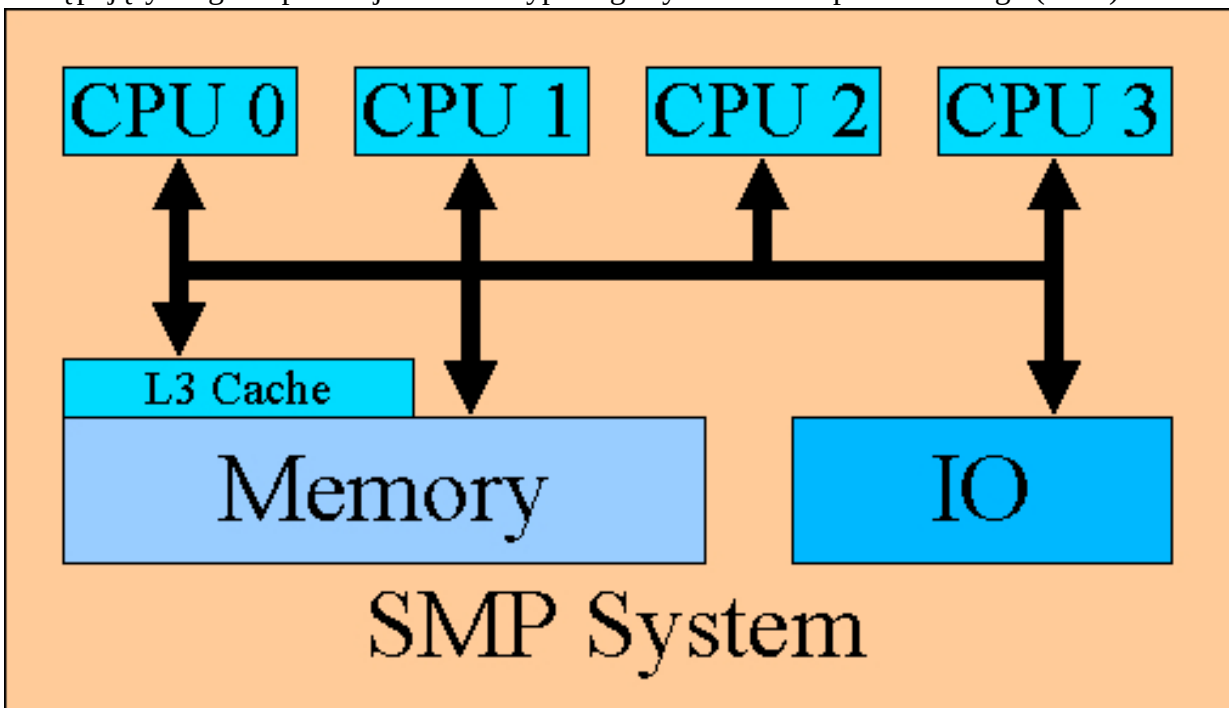
`http://www.linuxdevices.com/articles/AT2760742655.html`
`http://www.linuxdevices.com/articles/AT4936596231.html`

2.2 NUMA

Z drugiej strony, Linux przystosowuje się do dużych architektur serwerowych. Taką jest np. **NUMA** (Non-Uniform Memory Access).

2.2.1 Typowe SMP

Następujący diagram pokazuje schemat typowego systemu wieloprocesorowego (SMP):

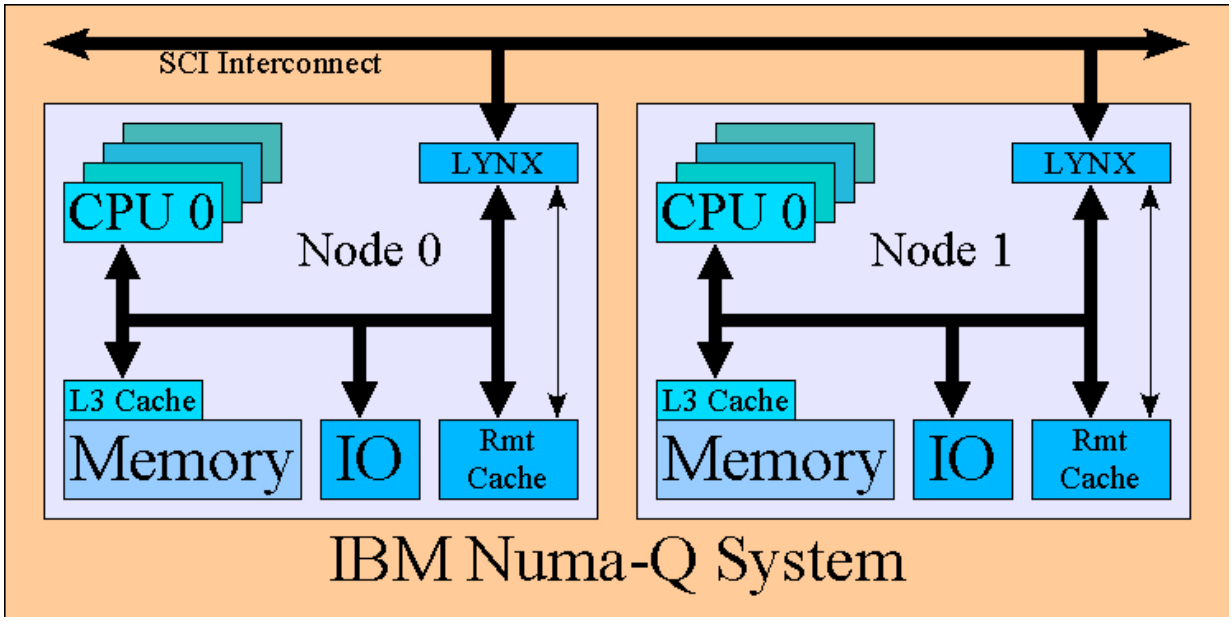


Typowe SMP zawiera kilka procesorów. Każdy z nich posiada cache L1 (zintegrowaną) i L2 (zewnętrzną, ale odpowiadającą jednemu procesorowi). Architektura może też mieć pamięć L3 (zewnętrzną), najczęściej współdzieloną przez kilka procesorów. Opóźnienia w korzystaniu z pamięci można uszeregować od najmniejszych:

- cache L1, L2, L3, pamięć

2.2.2 NUMA

Następujący diagram pokazuje architekturę NUMA. W tym wypadku jest to konkretny system - IBM Numa-Q:



Każdy węzeł jest prostym czteroprocesorowym SMP. Każdy procesor w węźle posiada własny cache L1 i L2. Węzeł posiada cache L3 wspólny dla jego procesorów. Węzły połączone są przez urządzenie Lynxer, które zawiera interfejs SCI. Procesory z jednego węzła mogą korzystać z pamięci innego węzła. Do przyspieszenia tej operacji używa się dodatkowej pamięci - zdalnego cache. Opóźnienia w korzystaniu z pamięci można uszeregować od najmniejszych:

- cache L1, L2, L3, pamięć lokalna, cache zdalny, pamięć zdalna

2.2.3 Wnioski

W typowym SMP dostęp do całej pamięci odbywa się przez jedną magistralę. Takie rozwiązanie działa dobrze dopóki w systemie jest niewiele CPU. Przy większej ich ilości (np. 16) magistrala staje się wąskim gardłem.

Architektura NUMA została stworzona z myślą o pozbyciu się problemów przeciążonej magistrali. Zachowując dostęp do pamięci innych węzłów znacznie szybszy niż w przypadku klastrów.

Dla architektury NUMA w jądrze 2.6 planowanie wykonania procesów jest wykonywane przez ten sam Scheduler O(1). Potrzebna jest jednak znajomość topologii systemu. Poza tym, planista musi dbać żeby procesy wykonywały się w ramach jednego węzła (pole `mempolicy` w `task_struct`).

2.2.4 Źródło

http://lse.sourceforge.net/numa/faq/system_descriptions.html

2.3 HyperThreading

Hiperwątkowość jest obsługiwana np. przez procesor **Pentium IV**. Technologia pozwala fizycznie jednemu procesorowi wykonywać równolegle kilka wątków (dla **P IV** - dwa). Dzięki temu aplikacja wielowątkowa logicznie widzi dwa procesory.

Technologia HT może działać dzięki współużytkowaniu przez wątki zasobów procesora na 3 sposoby:

Replikacja - osobne kopie zasobów dla każdego wątku

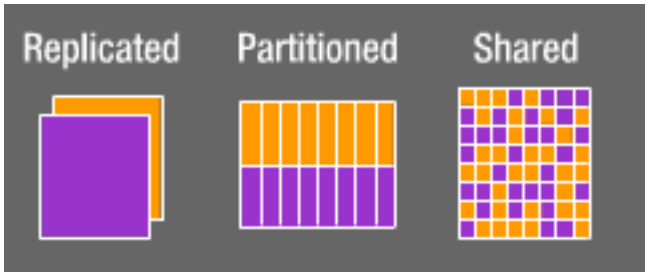
- wszystko związane ze stanem procesora (rejestry)
- wskaźniki instrukcji

Podział - podział zasobów między wątki

- bufory
- kolejki

Współdzielenie - współdzielnie fizycznie tych samych zasobów

- cache



2.3.1 Źródło

`http://or1cedar.intel.com/media/training/intro_ht_dt_v1/tutorial/index.htm`

2.4 Inne

Pojawiła się obsługa jeszcze wielu innych architektur, najważniejszymi z nich są:

- AMD 64-bit (x86-64)
- PowerPC 64-bit(ppc64)
- User Mode Linux (w przeciwieństwie do jądra 2.4 nie potrzeba już patchy)

3 Procesy

3.1 Nowy scheduler w jądrze 2.6.

3.1.1 Wprowadzenie.

Jedną z kluczowych zmian wprowadzonych w jądrze 2.6 jest nowy scheduler. Stary algorytm nie obsługiwał dostatecznie szybko systemów wieloprocesorowych, które stają się coraz bardziej popularne. Także rosnące obciążenie systemów miało wpływ na rozwój tego programu. Ingo Molnar, twórca schedulera do jądra 2.6 zaproponował algorytm działający w czasie stałym. Dodatkowo jego algorytm doskonale radzi sobie na maszynach wieloprocesorowych.

Co to w ogóle jest scheduler?
Scheduler to program zarządzający procesami. Zajmuje się przede wszystkim szeregowaniem procesów. Kieruje się pewną polityką przydzielania procesom procesora.

Wady schedulera 2.4.

- Przy dużych obciążeniach systemu, spada znacząco reakcja systemu.
- Poziom interaktywności procesu nie jest uwzględniany przy obliczaniu priorytetów.
- Brak bezwarunkowego postarzania śpiących procesów.
- Niski poziom współbieżności w przypadku pracy na maszynie wieloprocesorowej.

3.1.2 Cechy nowego schedulera w porównaniu ze starym.

O(1).
Nowy scheduler nie ma już drogiej funkcji `runqueue_lock`. Pętla przeliczająca (która przeliczała kwanty dla procesów) i pętla funkcji `goodness()` zostały wyeliminowane. Do funkcji `wakeup()` i `schedule()` zostały użyte algorytmy O(1).

Zalety tego algorytmu można docenić przede wszystkim gdy system pracuje pod dużym obciążeniem. Znacząco zmniejsza się wówczas narzut schedulera na działanie systemu operacyjnego.

Jak to było z współbieżnością w 2.4?
W starym schedulerze algorytm odświeżający kwanty czekał aż wszystkie procesy wyczerpały swoje kwanty czasu, czyli nim nastąpił koniec epoki. Zatem na maszynie wieloprocesorowej, większość procesorów wykonywała `idle`. Musiały biernie czekać na koniec epoki i wykonanie algorytmu odświeżającego kwanty. To obniżało wydajność SMP.

Ponadto w algorytmie tym procesory `idle` zaczynały wykonywać procesy oczekujące (jeśli ich własny procesor był zajęty) których kwant czasu jeszcze się nie wyczerpał, powodując że procesy zaczynały odbijać się między procesorami. Kiedy ”skaczący” proces miał przypadkiem wysoki priorytet, lub był interaktywny, spadała wydajność całego systemu.

Jak nowy scheduler rozwiązał problemy ze współbieżnością?
Nowy scheduler zwiększył stopień współbieżności systemu przez rozdzielenie kwantów na poszczególne procesory, eliminację globalnej synchronizacji i pętli przeliczającej.

Kwanty czasu są teraz odświeżane zaraz po ich wygaśnięciu, a proces przenoszony jest do innej tablicy. Dzięki zastosowaniu tego podejścia, koniec epoki oznacza jedynie zamianę wskaźników.

Dodatkowo nowy scheduler zawiera mechanizmy umożliwiające wykonywanie operacji `sleep`, `wakeup` i `context-switch` całkowicie równolegle.

Interaktywność.
Z nowym schedulerem użytkownik nie powinien zauważyć że system nie odpowiada na wydarzenia jak kliknięcie myszą czy naciśnięcie klawisza, nawet podczas dużego obciążenia systemu. Procesy interaktywne mają odpowiednio wyższy priorytet.

3.1.3 Schemat działania nowego algorytmu.

Najważniejsze struktury.

Dwie, posortowane priorytetami tablice priorytetów na każdy procesor. Pierwsza tablica zawiera wszystkie zadania przydzielone temu procesorowi, a które to procesy mają jeszcze kwant czasu (active). Druga (expired) zawiera wszystkie te, którym skończył się kwant (zadania są także posortowane). Tablice te nie są dostępne bezpośrednio, tylko poprzez dwa wskaźniki w strukturze `runqueue`(oddzielnej dla każdego CPU). Jeśli tablica active się opróżni, zamieniamy wskaźniki.

```
struct prio_array {
  int nr_active;
        /* liczba procesów */
  unsigned long bitmap[BITMAP_SIZE];
        /* mapa bitowa procesów o danym priorytecie */
  struct list_head queue[MAX_PRIO];
        /* tablica list procesów */
```

Struktura `runqueue`, zawiera informacje dla konkretnego procesora. Każdy CPU ma swoją kopię. Poprzez tą strukturę osiąga się dostęp do `prio_array`.

```
struct runqueue {
  spinlock t lock;
        /*sekcja krytyczna dostępu dla procesów*/
  unsigned long nr_running;
        /*liczba zadań gotowych do wykonania*/
  unsigned long nr_switches;
        /*liczba przełączeń kontekstu dla procesora*/
  unsigned long expired_timestamp;
        /*od kiedy jakiś proces znajduje się w expired*/
  unsigned long nr_uninterruptible;
        /*liczba procesów w stanie uninterruptible*/
  struct task_t *curr;
        /*proces wykonujący się na tym procesorze*/
  struct task_t *idle;
        /*proces idle dla tego procesora*/
  struct prio_array_t *active;
        /*procesy aktywne które mają czas*/
  struct prio_array_t *expired;
        /*procesy aktywne którym skończył się czas*/
  struct prio_array_t arrays[2];
        /*prio_array_t używane jako active i expired*/
 ...
};
```

inne struktury znane nam już ze starego schedulera, takie jak `task_struct`pozostały w prawie nie zmienionej formie.

Najważniejsze funkcje.

```
void load_balance(runqueue_t *this_rq, int idle)
```

Jeśli zajdzie taka potrzeba, przepycha proces z jednego procesora na drugi, aby wyrównać ich zużycie. Wywoływana explicite gdy kolejki procesów gotowych 2 procesorów są nie wyrównane, jak i okresowo zgodnie z taktami zegara. Przed wywołaniem bieżące kolejki muszą zostać zablokowane.

```
void schedule()
```

Główna funkcja szeregująca. Po powrocie z niej proces o najwyższym priorytecie przejmie procesor.

Priorytety.

Procesy tak jak dawniej podzielone są na trzy klasy:
Procesy czasu rzeczywistego `SCHED_RR`, `SCHED_FIFO` (priorytety z zakresu 1..99, ustalane przez użytkownika i nie modyfikowane przez scheduler). Różne polityki w klasach `SCHED_RR`i `SCHED_FIFO`.
Zwykłe procesy: `SCHED_NORMAL`(priorytety z zakresu 100..139 w zależności od wartości pola nice (-20..19)). Priorytet tych procesów jest modyfikowany przez scheduler.
Proces idle.

Opis działania całości.

Schemat działania algorytmu jest bardzo podobny. Podstawowe różnice to:
1. Sposób wyliczania epoki:

```
struct prio_array array = rq->active;
 if (array->nr_active=0) {
   swap (rq->active, rq->expired);
   rq->expired_timestamp = 0;
 }
```

2. Sposób przyznawania kwantów (kwant jest przyznawany procesowi gdy skończy mu się stary):

```
#define BASE_TIMESLICE(p) (MIN_TIMESLICE + \
   ((MAX_TIMESLICE - MIN_TIMESLICE) * \
   (MAX_PRIO-1-(p)->static_prio)/(MAX_USER_PRIO - 1)))

static inline unsigned int task_timeslice(task_t *p){
  return BASE_TIMESLICE(p);
}
```

3. Wybór procesu do wykonania (procesu o najwyższym priorytecie). Dzięki temu że funkcja `sched_find_first_bit(bitmap)` wykonuje się w czasie stałym, to wybór procesu dokonuje się w czasie stałym.

```
schedule(){ ...
int idx;
struct list_head queue;
struct task_t next;
struct prio_array_t *array;

idx = sched_find_first_bit(array->bitmap);
     /* idx = wartość procesu o najwyższym priorytecie*/
queue = array->queue + idx;
     /* queue = kolejka procesów o priorytecie idx */
next = list_entry(queue->next, task_t, run_list);
     /* next = pierwszy element z kolejki queue */
... }
```

4. Promowanie procesów interaktywnych.
Jego implementacja jest w wielu miejscach schedulera. Istota polega na połączeniu dwóch technik. Pierwsza każe co jakiś czas przeliczać ”stopień interaktywności procesu” - pole static_prio z `task_struct`. Jego wartość waha się w granicy -5..5. Druga technika mówi, że proces uznany za interaktywny po wyczerpaniu kwantu przeważnie (aby uniknąć zagłodzenia) nie trafia do tablicy expired, tylko z powrotem do tablicy active. ”Stopień interaktywności procesu” jest

wyliczany w funkcji `effective_prio()`i w uproszczeniu jest to różnica czasu jaki proces czeka na zdażenie i czasu w jakim proces się wykonuje.

3.1.4 Podsumowanie, czyli zalety nowego schedulera.

- Dobra interaktywna wydajność systemu nawet podczas dużego obciążenia - jeśli użytkownik napisze coś, albo kliknie, to system powinien odpowiadać cały czas i powinien wykonywać płynnie zadania użytkownika, nawet podczas rozsądnie wielkiego obciążenia w tle.
- Wysoka wydajność funkcji `schedule`i `wakeup`przy niewielu procesach gotowych.
- Sprawiedliwość - żaden proces nie powinien pozostać bez procesora na zbyt długi bądź krótki okres czasu.
- Priorytety - mniej istotne mają niższy priorytet, a te ważniejsze wyższy.
- Efektywność SMP - żaden procesor nie powinien zostać idle gdy jest zadanie do wykonania.
- Powiązanie SMP - proces który działa na jednym procesorze, powinien pozostać z nim powiązany. Procesy nie mogą ”skakać” między procesorami.
- Szeregowanie w pełni O(1) - nie ma pętli przeliczającej kwanty (która była zmorą przez zapychanie cache’u L1). Nie ma pętli goodness. `wakeup()`, `schedule()`, timer interrupt wszystkie są algorytmami O(1).
- Skalowalność SMP - nie ma wielkiej funkcji `runqueuelock`- kolejki procesów są przypisane do procesorów. 2 procesy na 2 różnych procesorach mogą w tym samym czasie się budzić, wywoływać schedulera i zmieniać kontekst na 2 procesorach całkowicie równolegle, bez wzajemnego wykluczania (interlocking). Wszystkie struktury danych schedulera są zoptymalizowane pod względem skalowalności.
- Szeregowanie porcjami (batch scheduling) - znacząca większość procesów o charakterze obliczeniowym odnosi korzyści z szeregowania porcjami. Nowy scheduler wykonuje takie szeregowanie dla procesów o najniższym priorytecie (a więc procesy o wartości pola nice równym +19 będą miały taki status automatycznie). W tym algorytmie szeregującym procesy o nice +19 są SCHEDILDE z punktu widzenia użytkownika.
- Scheduler radzi sobie płynnie z ekstremalnie wysokim obciążeniem systemu bezawaryjnie i bez żadnych zachwiań.
- O(1) RT scheduling.
- Wywołanie synów (powstałych w wyniku funkcji `fork`) w pierwszej kolejności.

3.2 Wywłaszczanie jądra

3.2.1 Wprowadzenie.

Co to jest wywłaszczanie?

Wywłaszczanie, to odbieranie procesowi procesora. Umożliwia płynne działanie wielu procesów, umożliwia ich częstą zmianę, dzięki czemu użytkownik ma wrażenie, że zadania są wykonywane współbieżnie.

Jak to było do tej pory?

W poprzednich wersjach jądra (włączając w to jądra 2.4), było niemożliwe wywłaszczenie procesu pracującego w trybie jądra (włączając w to procesy, które weszły w tryb jądra przez wywołanie funkcji systemowej) dopóki, albo o ile proces sam nie zrzekł się procesora.

Co daje wywłaszczanie procesu pracującego w trybie jądra?

Wywłaszczanie jądra powinno znacząco obniżyć czas oczekiwania na reakcje aplikacji interaktywnych użytkownika, aplikacji multimedialnych i tym podobnych. Ta cecha jest szczególnie przydatna dla systemów czasu rzeczywistego.

Historia.

Łatka umożliwiająca wywłaszczanie jądra została wprowadzona do jądra 2.5. Jej twórcą był Robert Love. W jądrze 2.6 jest ona dostępna jako opcja podczas kompilacji jądra (domyślnie chyba jest włączona).

Czy zawsze?

Oczywiście, nie wszystkie sekcje kodu jądra mogą być wywłaszczone. W niektórych jego krytycznych sekcjach jest to nie dozwolone. Są one zablokowane przed wywłaszczaniem. Blokada powinna zapewniać że zarówno struktury danych (przypisane do CPU) jak i stany procesorów są chronione przed wywłaszczeniem.

3.2.2 Od środka

gdzie nie można wywłaszczać jądra?
Poniższy kod przedstawia problem struktur danych procesorów (w systemie SMP):

```
  int arr[NR_CPUS];
arr[smp_processor_id()] = i;
   /* gdyby w tym miejscu wywłaszczyć jądro */
j = arr[smp_processor_id()]
   /* okazałoby się że i i j nie muszą być takie same   /
   /  gdzyż smp_processor_id() nie musi być takie same */
```

W tej sytuacji jeśli przydarzyło by się wywłaszczenie jądra w tym właśnie punkcie, proces mógłby zostać przydzielony do innego procesora przez algorytm szeregujący. Wówczas funkcja `smp_processor_id()`zwróciłaby inną wartość.

Tryb FPU jest kolejnym przypadkiem, kiedy stan procesora powinien być chroniony przed wywłaszczeniem. Gdy jądro wykonuje operacje zmiennoprzecinkowe, stan FPU nie jest zapamiętywany. Jeśli nastąpiłoby tu wywłaszczenie, to po zmianie kontekstu stan FPU jest całkowicie różny od tego który był przed wywłaszczeniem. Tak więc kod zmiennopozycyjny musi zawsze być zablokowany przed wywłaszczeniem.

Takich sytuacji powinno się unikać przez blokowanie.

Techniki zabezpieczające struktury jądra
Blokowanie może być wykonane poprzez nie pozwalanie na wywłaszczanie w krytycznych sekcjach i pozwalaniem na nie później. Następujące #defines zostały dołączone do jądra 2.6 aby zezwalać i blokować wywłaszczanie:
`preempt_enable()`- zmniejsza licznik wywłaszczeń (preempt counter).
`preempt_disable()`- zwiększa ów licznik.
`get_cpu()`- wykonuje najpierw `preempt_disable()`, a potem `smp_processor_id()`.
`put_cpu()`- pozwala na wywłaszczanie.
Używając tych definicji możemy przepisać poprzedni kawałek kodu:

```
  int cpu, arr[NR_CPUS];
arr[get_cpu()] = i;
    /* blokowanie wywłaszczanie */
j = arr[smp_processor_id()];
    /* sekcja krytyczna */
put_cpu()
    /* zezwala na wywłaszczanie */
```

Warte uwagi jest to że funkcje `preemp_disable()`, `preemp_enable()`są zagnieżdżone. To znaczy `preempt_disable()`może być wywołane n razy i wywłaszczenie będzie dopuszczone gdy pojawi się n razy `preempt_enable()`.

Wywłaszczanie jest pośrednio zabronione jeśli włączone są spin locki. Na przykład wywołanie `spin_lock_irqsave()`, pośrednio zapobiega wywłaszczaniu poprzez wywołanie `preempt_disable(`
wywołanie `spin_unlock_irqrestore()`włącza możliwość wywłaszczaniam wołając funkcję `preempt_enable()`.

3.2.3 Podsumowanie

co to dało?

Jeśli chodzi o jądro 2.6, to może być ono wywłaszczone, tak że ważne aplikacje użytkownika mogą kontynuować działanie. Główną zaletą tego jest to że nie będzie już większych ”zacięć” w użytkowej interaktywności systemu i użytkownik będzie odczuwał że wszystko wykonuje się szybciej.

4 Odwrotne odwzorowanie stron (reverse mapping)

4.1 Stara zasada działania

Przy zarządzaniu pamięcią każda strona (<struct page>) może być wykorzystywana w wielu miejscach. Zwyczajowo strona może być zwolniona dopiero, gdy wszyscy 'zainteresowani' się jej zrzekną.

Jednak przy dynamicznym zarządzaniu stronami, polecenie zwolnienia danej strony może powstać oddolnie, czyli z punktu widzenia tejże strony. Wówczas należy wszystkich 'zainteresowanych' powiadomić o stanie tej strony, aby mogli ją sobie jej zawartość przechować gdzieś indziej.

W jądrze 2.4 trzeba było w tym celu przeszukać wszystkie takie potencjalne miejsca, bo nie wiadomo było dokładnie, kto jest wśród Grupy Trzymającej Stronę.

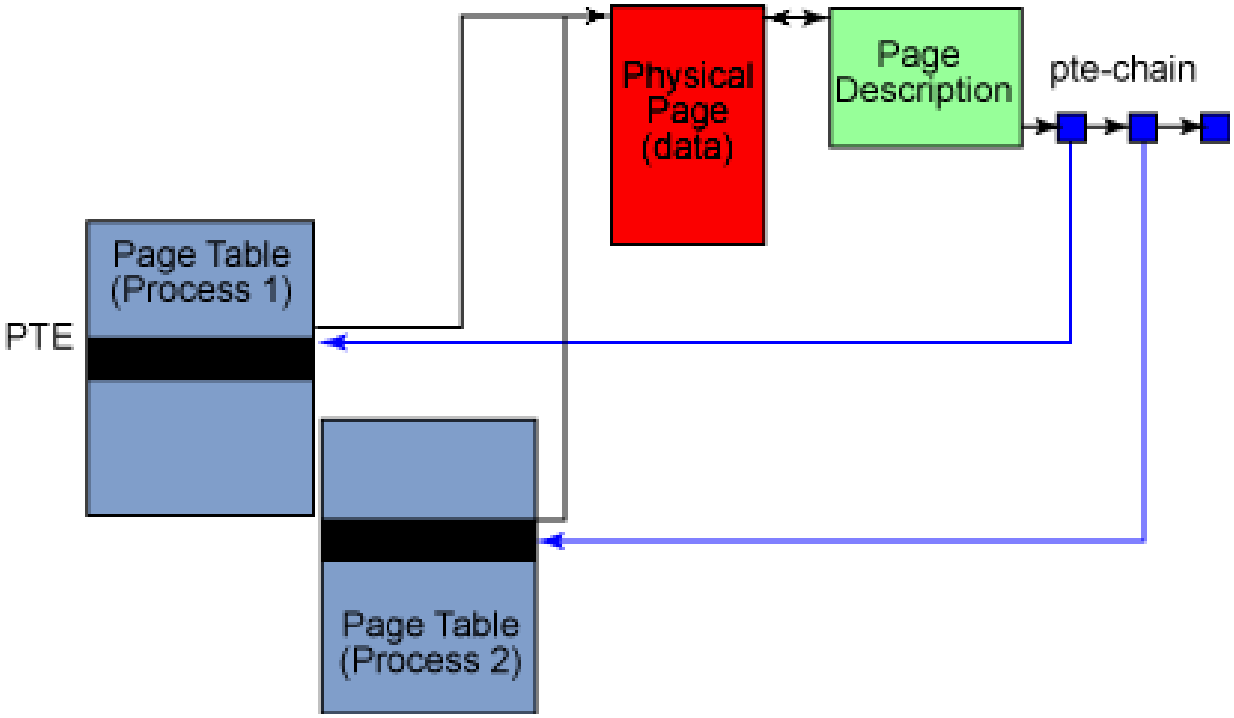
4.2 Zmiany - podejście pierwsze

Już na początku 2002 roku Riek van Riel zaproponował dokładne i wierne odwrotne odwzorowanie, do którego odnoszono się później jako <rmap>. Riek w swoim patchu dodał do struktury <page> jednokierunkową listę wskaźników do tych tablic stron, w których znajduje się odniesienie do danej strony.

```
union {
  struct pte_chain *chain;/* Reverse pte mapping pointer. */
                          * protected by PG_chainlock */
  pte_addr_t direct;
} pte;
```

Dodatkowo, przy tylko jednym odniesieniu, używany jest bezpośredni wskaźnik do tablicy stron, nie wykorzystując dodatkowej pamięci na struktury listy.

4.2.1 Zasada <rmap>



4.2.2 Problemy

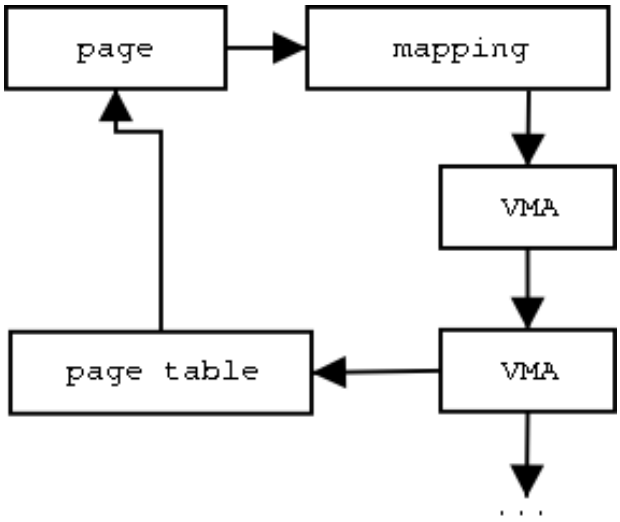
Patch <rmap> spisywał się nieźle, jeśli chodzi o prędkość działania. Gorzej było z zużyciem pamięci. Łańcuchy wskaźników zużywały cenną pamięć z niskich rejonów. Dodatkowo były małe, więc nieefektywne w zarządzaniu. Cała idea była słabo skalowalna. Dodatkowo, przy forku, kopiowanie łańcuchów dla każdej strony było czasochłonne.

4.3 Zmiany - podejście drugie

Zamiast stosować różne triki przyspieszające <rmap>, rok później Dave McCracken zaproponował zupełnie inne rozwiązanie. Postanowił wykorzystać istniejące jeszcze w wersji 2.4 struktury, i nieco dłuższą drogą wyszukiwać zależności.

Jego technikę ochrzczono mianem ’object-based reverse mapping’ <objrmap>.

4.3.1 Zasada <objrmap>



Strony reprezentujące zmapowane pliki, bądź kod programu, są opisane w obiektach reprezentujących te pliki w kernelu. Obiekty te mają bezpośrednie dowiązania do struktur <vm_area_struct> procesów, które z danego pliku korzystają (czyli mają referencje do stron mapujących ten plik).

<vm_area_struct> procesów zapewniają informację o tym, jaki byłby adres danej strony w przestrzeni adresowej procesu. A to umożliwia nam szybkie znalezienie odpowiedniego wpisu w tablicy stron procesu. Oczywiście jest to jedynie ograniczenie zasięgu poszukiwań, ale znaczne, i tego właśnie poszukujemy.

4.3.2 Problemy

Niestety, jest jeszcze drugi rodzaj stron - tzw. anonimowe strony, które reprezentują zwykłą pamięć procesów. Do nich nasza taktyka zupełnie się nie odnosi.

Tutaj autor zaproponował pozostanie przy standardowym <rmap>, argumentując, że do takich stron są zazwyczaj tylko pojedyncze odniesienia, do których w większości przypadków używane będzie pole <direct>, a nie łańcuch odwrotnych odniesień.

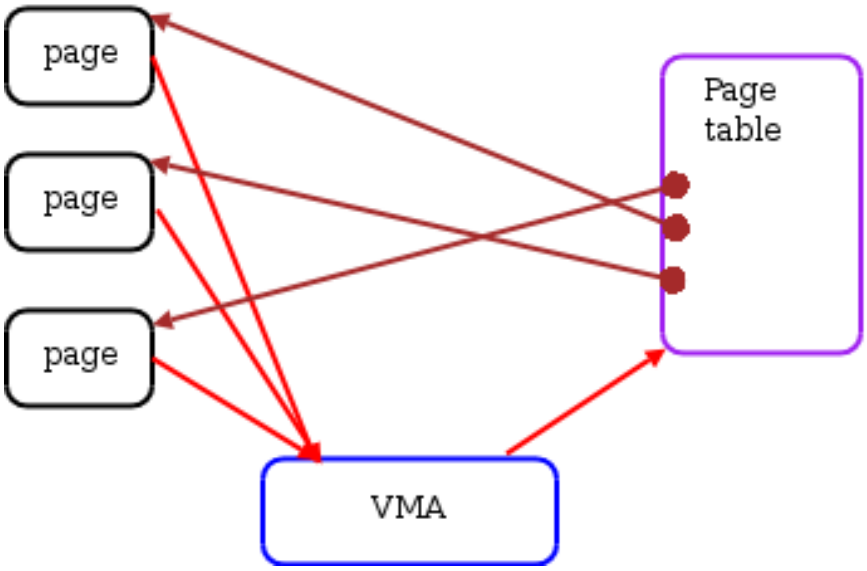
Dodatkowo autor zasugerował, że na systemach obsługujących serwery bazodanowe mapowanie plików jest głównym powodem zużycia stron, i wówczas jego podejście wnosi znaczącą poprawę.

4.4 Zmiany - podejście trzecie

Rok temu Andrea Arcangeli zaproponował finalizację tego podzielonego podejścia. Pozostawił obsługę <objrmap> dla stron plików. Postanowił wprowadzić odpowiednie zmiany do struktury <vm_area_struct>, tak, aby można było podobnie rozwiązać problem anonimowych stron.

4.4.1 Zasada działania

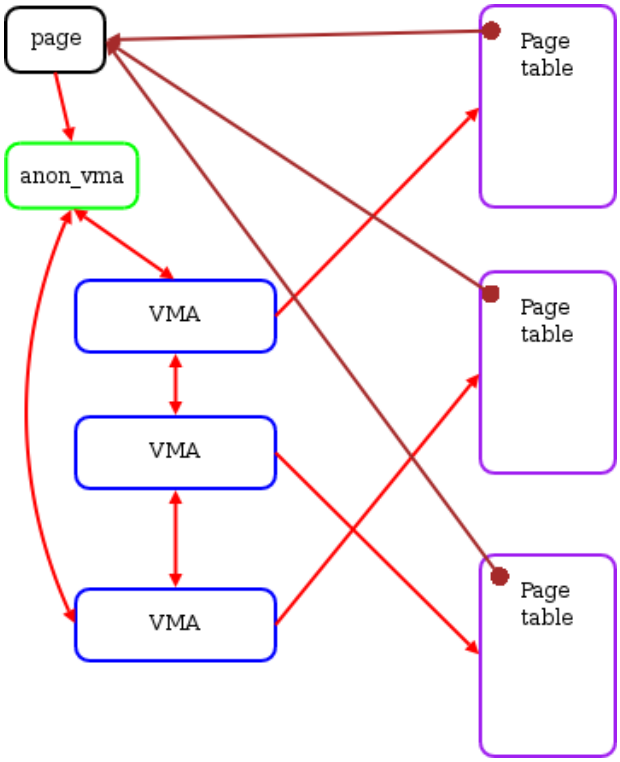
Anonimowe strony zawsze powstają w obrębie jednego procesu. Nie są więc współdzielone zaraz po stworzeniu. Nie potrzeba więc żadnych łańcuchów odwrotnych odnośników. Pole <mapping>, jest wówczas wykorzystywane jako substytut starego pola <direct>. Więc gdy proces ma kilka nie współdzielonych anonimowych stron w obrębie tej samej <vm_area_struct>, wygląda to następująco:



Życie jednak komplikuje się, gdy proces odpala fork(). Wówczas pojedyncza strona będzie wskazywana przez kilka tablic stron procesów i pojednczy wskaźnik nie będzie już wystarczał. By poradzić sobie z tym problemem, Andrea postanowił połączyć struktury <vm_area_struct> w obrębie spokrewnionych procesów.

Dodał też dodatkową strukturę <anon_vma> (linux/rmap.h), na którą bezpośrednio wskazują strony. Jest to strażnik listy, gdyż strony nie mogłyby wskazywać na jakieś poszczególne <vm_area_struct>, które mogłoby w dowolnym momencie zostać zutylizowane wraz z zakończeniem jednego z procesów.

Struktura prezentuje się następująco:



4.4.2 Koszty

Potrzebujemy nowego wpisu list w strukturze <vm_area_struct>, a także jednego <anon_vma> dla każdej listy. Jest to jednak koszt rzędu ilości <vm_area_struct>, a nie ilości stron, jak w starym <rmap>.

4.5 Źródła

- LWN: Driver porting to 2.6 series (`lwn.net/Articles/driver-porting`)
- LWN: Kernel Development
- LWN: The object-based reverse-mapping VM
- Dokumentacja do nowego kernela.
- Kod źródłowy starego i nowego kernela.

5 Urządzenia (z perspektywy dewelopera)

5.1 Wstęp

- Nowe jądro wprowadza wiele zmian dla deweloperów sterowników
- Ładowacz modułów został całkowicie zmieniony. Stare narzędzia nie będą już działały. Podobnie jak stare, najprostsze nawet moduły i makefile (zob. rozdział (5.5)).
- Unified Device Model - Jednym z ważniejszych zmian w jądrze 2.6 było wprowadzenie ujednoliconego modelu urządzeń. Model reprezentuje architekturę urządzeń i odpowiadającą jej warstwę systemową dzięki nowym strukturom danych. Całość daje zwiększoną kontrolę nad zasilaniem urządzeń i możliwość łatwiejszego zarządzania zadaniami wykonywanymi przez urządzenia. W szczególności możliwość śledzenia:
  - wszystkich urządzeń obecnych w systemie i szyn do których są podłączone
  - stanu zasilania urządzeń
  - obecności sterowników w systemie i odpowiadających im urządzeń
  - struktury szyn - która szyna jest podłączona do której (np. USB -> PCI)
  - klas urządzeń (dyski, partycje, etc)

  Szczegóły można znaleźć w rozdziale (5.2).
- Pojawiła się wsparcie od strony jądra dla zlecania asynchronicznych operacji I/O. Dotychczas można było jedynie skorzystać ze specjalnych bibliotek (czyli tylko z przestrzeni użytkownika). Asynchroniczne I/O opisano w rozdziale (5.3)
- Jest też alternatywa dla struktury `buffer_head` w postaci struktury `BIO`. Więcej w rozdziale (5.4)

5.2 Unified Device Model

5.2.1 Potrzebne terminy

urządzenie
Fizyczne lub wirtualny obiekt podłączony do fizycznej lub wirtualnej szyny.

sterownik
Kawałek oprogramowania, przypisany do urządzenia, które wykrywa i którym zarządza.

szyna
Urządzenie, do którego mogą być podłączone inne urządzenia.

klasa
Typ urządzenia ze względu na funkcje, jakie mogą wykonywać. Klasami mogą być np.: dyski, partycje, porty szeregowe, itd.

podsystem
Najwyższy poziom spojrzenia na urządzenia w systemie. Przykładami podsystemów mogą być: urządzenia (hierarchiczna struktura wszystkich urządzeń), szyny (spojrzenie ma urządzenia z perspektywy podłączenia do szyn), klasy (spojrzenie na urządzenia z perspektywy klas, do jakich należą), sieć (elementy związane z siecią), itp. Najczęściej urządzenia należą do kilku podsystemów, ale w każdym z nich są położone w innej hierarchii.

5.2.2 Sysfs

Sysfs jest wirtualnym systemem plików, który zapewnia dostęp z przestrzeni użytkownika do reprezentacji modelu urządzeń. Najczęściej montowany jest na `/sys/`. Katalogi na najwyższym poziomie odnoszą się do podsystemów. Ponieważ urządzenia mogą się znajdować w różnych podsystemach, znajdują się w różnych miejscach sysfs (dzięki linkom symbolicznym).

Np. Pierwszy dysk IDE będzie widoczny w systemie w kilku miejscach:

```
/sys/devices/pci0/00:11.1/ide0/0.0
/sys/block/hda/device
/sys/bus/ide/devices/0.0
/sys/bus/pci/devices/0.11.1
/sys/bus/pci/drivers/VIA IDE/00:11.1
```

Przykładowy fragment `/sys/`:

```
/sys
|-- block
|   |-- fd0
|   |   |-- dev
|   |   |-- iosched
|   |   |   |-- fifo_batch
|   |   |   |-- front_merges
|   |   |   |-- read_expire
|   |   |   |-- write_expire
|   |   |   ‘-- writes_starved
|   |   |-- range
|   |   |-- size
|   |   ‘-- stat
|-- class
|-- devices
|   |-- legacy
|   |   |-- floppy0
|   |   |   |-- name
|   |   |   ‘-- power
|   |   |-- name
|   |   ‘-- power
|   :
|
|-- firmware
|-- fs
|   |-- devpts
|   |-- ext3
|   |-- iso9660
|   |-- proc
|   |-- sysfs
|   |-- tmpfs
|   |-- usbdevfs
|   ‘-- usbfs
‘-- net
    ‘-- eth0
```

Warto zauważyć, że w przeciwieństwie do `/proc` pliki z `/sys` zawierają najczęściej tylko jedną wartość. Bardziej jasne jest wtedy, co ta wartość oznacza, i co oznacza zapisywanie do tego pliku.

5.2.3 Kobjects, ksets, ktypes, subsystems

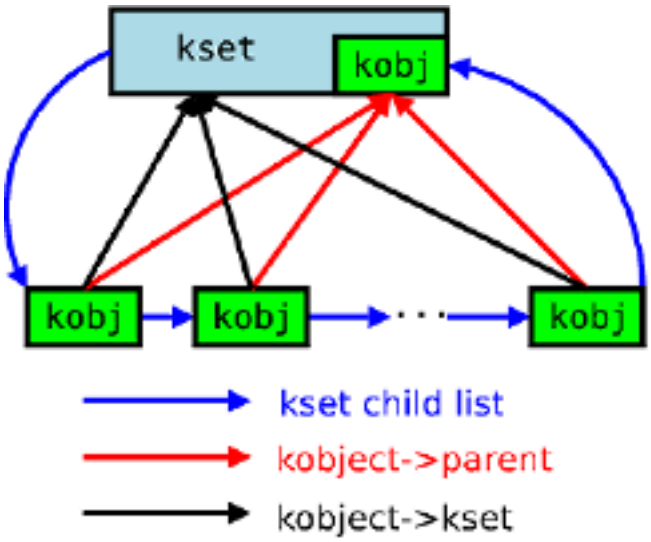
kobject -
Struktura `kobject` jest prostą reprezentacją obiektu znajdującego się w systemie. W sensie obiektowego języka programowania byłaby to klasa, po której większość innych dziedziczy.
Struktura posiada nazwę, i licznik odwołań. Jest także wskaźnik do rodzica, co pozwala organizować kobiekty w hierarchię, referencje do typu, i reprezentacja w sysfs.

ktype -
Typ związany z kobiektem. Kontroluje, co się stanie jeśli do kobiektu nie będzie już referencji oraz określa jego reprezentację w sysfs.

kset -
Kset jest grupą kobiektów będących tego samego typu. W przeciwieństwie do ktype jest bardziej agregacją kobiektów. Można wyróżnić kilka funkcji kset:

- Agregować grupę takich samych kobiektów, np. wszystkie urządzenia blokowe, sterowniki urządzeń PCI, etc.
- Kset pozwalają trzymać model urządzeń (i sysfs) w całości. Każdy kset zawiera kobject, który może być rodzicem dla innych kobiektów. W ten sposób konstruowana jest hierarchia obiektów.
- Kset zapewniają możliwość 'hotplug' kobiektów i wpływają na to, jak to jest raportowane do przestrzeni użytkownika.
- Spoglądając na katalog w sysfs, można powiedzieć że jego zawartością są kobiekty należące do tego samego kset.

subsystem -
Jest kolekcją kset, które wspólnie tworzą najbardziej ogólny podział w systemie. Najczęściej każdy subsystem jest związany z katalogiem w `/sys/`



Poniżej można obejrzeć deklaracje omówionych struktur:

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

struct kobj_type {
            void (*release)(struct kobject *);
            struct sysfs_ops    *sysfs_ops;
            struct attribute    **default_attrs;
    };

struct kset {
        struct subsystem        * subsys;
        struct kobj_type        * ktype;
        struct list_head        list;
        struct kobject          kobj;
        struct kset_hotplug_ops * hotplug_ops;
};

struct subsystem {
        struct kset         kset;
        struct rw_semaphore rwsem;
    };
```

5.3 Asynchroniczne wejście/wyjście

AIO (asynchronous input/output) jest udogodnieniem, pozwalającym użytkownikom na zlecanie wielu operacji wejścia/wyjścia, bez potrzeby czekania na zakończenie którejkolwiek z nich. Wynik danej operacji może być pobrany później. Wszelkie operacje blokowe były już asynchroniczne we wcześniejszych wersjach jądra, dlatego ten patch ich nie dotyczy. Urządzenia znakowe miały jednak czysto synchroniczne API, którego nie dało się nijak obejść bez dodatkowych zmian w kodzie samego systemu.

5.3.1 Cóż może użytkownik?

Na początek

```
int io_setup(int maxevents, aio_context_t *ctxp);
```

Tworzy dla nas kontekst (który określa deskryptor typu <aio_context_t>). Wszystkie operacje AIO muszą wykonywać się w jakimś kontekście. Struktura taka przechowuje potrzebne dane o zlecanych przez nas operacjach. Podajemy maksymalną ilość operacji, które chcielibyśmy móc naraz obsługiwać w tym kontekście.

Podejmujemy akcję

```
int io_submit(aio_context_t ctx, long nr, struct iocb *iocbs[]);
```

Zlecamy wykonanie operacji opisanych przez nas w strukturach <iocb> (IO Control Block). Zwracana jest liczba operacji, które udało się zainicjować. Jeśli procedura wykrzaczy się na którymś zleceniu, nie będzie nawet próbowała obsłużyć pozostałych.

W międzyczasie

```
long io_getevents(aio_context_t ctx_id, long min_nr, long nr,
                  struct io_event *events, struct timespec *timeout);
```

Oczekujemy na wykonanie <min_nr> spośród zleceń w danym kontekście. Oczekiwanie ograniczone jest czasowo. Może się zdarzyć, że będzie już wykonanych więcej niż to minimum. Wówczas deklarujemy chęć pobrania maksymalnie <nr> spośród nich.

```
long io_cancel(aio_context_t ctx_id, struct iocb *iocb,
               struct io_event *result);
```

Próbujemy odwołać zlecenie. Może być już oczywiście na to zbyt późno, wtedy dostaniemy odpowiedni kod błędu.

Na koniec

Zwracane rezulaty operacji, w postaci wypełnionych struktur <io_event>, mają bezpośrednie odniesienie do zadanych przy zlecaniu odpowiadających im struktur <iocb>, po prostu na siebie wskazują. Jeśli jednak nie mamy dostępu do struktur owych zleceń (czyli te wskaźniki nie mogą nam się przydać), możemy zorientować się w eventach, poprzez ich pola prywatne, których wartości pochodzą z pól prywatnych ustawianych przy zlecaniu.

```
int io_destroy(aio_context_t ctx);
```

Rezygnujemy z użytkowania danego kontekstu. Wszelkie nie obsłużone przez nas operacje, zakończone lub nie, zostają przez system masowo odwoływane. Wiąże się to niestety z czekaniem na obsługę tych odwołań, co czesto może trwać długo, i jest blokujące.

5.3.2 Z drugiej strony

Sporo pracy przy obsłudze asynchronicznych urządzeń znakowych wykonuje sam kernel. Aczkolwiek jest to jednak tylko lukier organizacyjny, gdyż większość znaczących operacji przypada na sterowniki odpowiednich urządzeń. Zgodnie ze specyfikacją, od autorów sterowników wymaga się dostarczenia obsługi trzech funkcji:

```
aio_read
aio_write
aio_fsync
```

Mają one składnię przywodzącą na myśl odpowiednie operacje synchroniczne na plikach.

Najprościej
Implementacja funkcji czytającej najprostszego sterownika mogłaby mieć postać:

```
ssize_t simplest_aio_read(struct kiocb *iocb, char *buffer,
                          size_t count, loff_t pos)
{
  return simple_synchronous_read(iocb->ki_filp, buf, count, &pos);
}
```

I rzeczywiście, obsługując wirtualne urządzenia, takie jak RAM-dyski, asynchroniczność nie jest istotna. Możemy po prostu wykonywać zlecenia natychmiast, w miarę przychodzenia.

Najczęściej
W większości przypadków jednak, natychmiastowe wypełnienie zadania nie będzie możliwe. Wtedy naszym zadaniem będzie zapamiętanie zlecenia i zwrócenie potwierdzenia zakolejkowania, czyli wartości -EIOCBQUEUED. Zapewne przy pisaniu potrzebne będzie tu przekopiowanie całego bufora z przestrzeni użytkownika do jakiegoś w naszej pamięci. To pewnie wymagać będzie podzielenia go na strony, itd. Jeżeli sterownik potrzebuje jakichś danych o zlecającym procesie, musi je pobrać przed zakończeniem funkcji, gdyż później nie będzie to już w ogóle możliwe.

Następnie nasz sterownik może wykonywać przeróżne operacje związane z porządkowaniem, łączeniem, czy rozdzielaniem operacji. Cokolwiek by chciał, aby realizować swoją strategię ekonomicznej obsługi urządzenia.

Gdy w którymś momencie dojdziemy do wniosku, że dane zlecenie zostało przez nas zakończone, wywołujemy funkcję systemową

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

gdzie zwracamy rezultat operacji jako pole <res>. Zwyczajowo jest to ilość przeczytanych/zapisanych bajtów, lub kod błędu. Pole <res2> jest na razie nie używane.

Czasami

Odwoływanie zlecenia nie jest w całości implementowane przez kernel. Ale nie obsługuje go też jakaś konkretna funkcja sterownika. Dokładniej, dla każdego zlecenia z oddzielna możemy przypisać jakąś naszą funkcję odwołującą. Gdy nie przypiszemy żadnej, odwołanie danego zlecenia po prostu nie będzie możliwe. Wówczas kernel sam zwróci użytkownikowi błąd - EAGAIN.

5.4 Blokowe wejście/wyjście

5.4.1 Stare rozwiązania

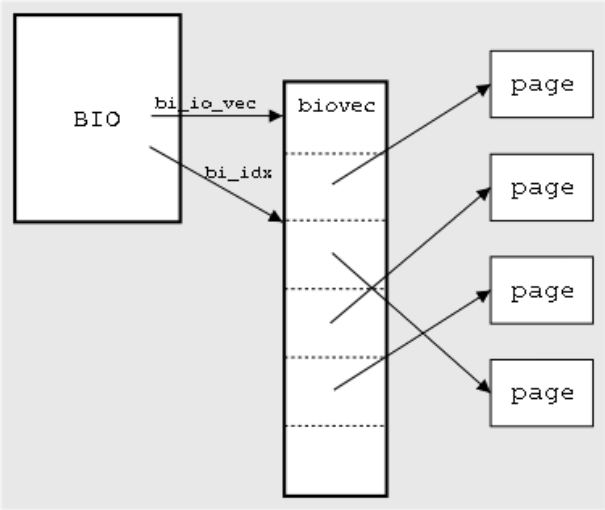
W kernelach serii 2.4 operacje blokowe były opisywane przez system za pomocą struktur <buffer_head> (linux/fs.h). Pojedyncza struktura tego typu opisywała żądanie transferu pojedynczej strony pamięci. Przechowywała przy tym masę meta-danych. Była uniwersalna - nadawała się zarówno do minimalnych transferów, jak i do sporych bloków, które opisywały listy takich struktur.

5.4.2 Problemy

Jednym z głównych problemów, które zasugerowały zastąpienie jej czymś nowym, była potrzeba dzielenia przez system każdego transferu na pojedyncze <buffer_head>. Tymczasem po przekazaniu takiego 'łańcuszka żądań' do sterownika urządzenia, sterownik najczęściej znów je łączył, bo spokojnie był w stanie obsługiwać w jednym rzucie o wiele większe żadania. Chcielibyśmy więc wypchnąć redundantne meta-dane do jakiejś wyższej struktury.

Chcielibyśmy też mieć większą swobodę w określaniu parametrów całego procesu obsługi transferów blokowych, nie musząc wchodzić w detale implementacyjne samego kernela. Nowe struktury powinny być znakomicie parameryzowalne.

5.4.3 Struktura bio



Nowa struktura przypomina kontener dla starych <buffer_head>. Jej główną częścią jest bowiem tablica elementów <bio_vec> (linux/bio.h). Pojedynczy taki element opisuje transfer dotyczący jednej strony.

```
struct bio_vec {
  struct page   *bv_page;
  unsigned int    bv_len;
  unsigned int    bv_offset;
};
```

Główne pola struktury <bio> (linux/bio.h):

```
struct bio {
  struct bio      *bi_next;   /* polaczenie na kolejce zadan */
  struct block_device *bi_bdev;
  unsigned long       bi_rw;      /* czytanie/pisanie, priorytety */

  unsigned short      bi_vcnt;    /* ilosc elementow bio_vec */
  unsigned short      bi_idx;     /* aktuala pozycja w tablicy bio_vec */

  struct bio_vec      *bi_io_vec; /* tablica elementow bio_vec */
  .........
};
```

5.4.4 Używanie bio

Najprościej

Pole <bi_idx> pełni tu funkcję iteratora po tablicy struktur <bio_vec>. Mając to na uwadze, z <bio> można korzystać strona po stronie. Zupełnie jak ze starych <buffer_head>, tylko zgrabniej, gdyż można tu użyć makra:

```
bio_for_each_segment(bvec, bio, i)
```

Sprytniejsze sterowniki mogą natomiast wykorzystać fakt, że tablica <bio_vec> reprezentuje ciągły obszar pamięci, i przeprowadzić transfer w jednym rzucie.

Na koniec trzeba powiadomić kernela o zakończeniu obsługi <bio> za pomocą:

```
void bio_endio(struct bio *bio, unsigned int bytes_done, int error);
```

Dodatki

Nowa strukturka zaiste jest elastyczna. Kernel udostępnia nam operację rozdzielenia danego <bio> na dwie części. Przydaje się to znakomicie do obsługi takich urządzeń jak LVM lub RAID. Drugą ciekawą operacją jest klonowanie <bio>. Klonują się jedynie meta-dane. Możemy ich wówczas użyć jako dodatkowych iteratorów po tablicy <bio_vec>.

5.4.5 Nieco wyższy poziom, czyli o kolejkach żądań (request queue)

Bazowo, <bio>, zaraz po powstaniu są przez kernela ładowane do struktur typu <struct request> (linux/blkdev.h). Te z kolei wchodzą w skład kolejek żądań - <struct request_queue> (linux/blkdev.h), na których dopiero operuje sterownik urządzenia.

Standardowo, naszym zdaniem, pisząc sterownik, jest wyłuskiwać kolejne <bio> i je obsługiwać, np. używając makr:

```
rq_for_each_bio(bio, rq) {
  bio_for_each_segment(bio_vec, bio, i) {
    /* obslugujemy dany <bio_vec> */
  }
}
```

Możemy przy tym wykorzystywać posiadane struktury do odpowiedniego sortowania czy łączenia transferów.

Następnie, dla danego <struct request> możemy poprosić kernela, aby za nas posprzątał, poprzez:

```
int end_that_request_first(struct request *req, int uptodate, int
nsectors);
```

Gdy całe żądanie zostanie obsłużone, zostanie zwrócone zero, i będziemy mogli ostatecznie skończyć z nim skończyć.

Alternatywnie

Niektóre urządzenia nie potrzebują takich złożonych rozwiązań, jak kolejki żądań. Ich sterowniki mogą wówczas obejść ten mechanizm i zajmować się żądaniami bezpośrednio, w miarę ich przychodzenia.

Tak jak i w wersjach 2.4, taki sterownik może pozostawić swoją funkcję bezpośredniej obsługi.

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

Jeśli taka funkcja uzna, że może zainicjować transfer opisany w danym <bio>, powinna to zrobić, po czym zwrócić zero. Sterowniki wielopoziomowe mogą przekierować <bio> do innego sterownika, poprzez zmianę odpowiedniego pola w <bio>, po czym zwrócić wartość niezerową. Wtedy system powtórnie rozpatrzy <bio>.

Parametryzacja

Samą kolejkę żądań możemy parametryzować do woli.

- Kernel pracuje zawzięcie nad łączeniem transferów. Sterowniki często mają jednak swoje limity. W takim przypadku zaznaczamy maksymalną wielkość jednego żadania - w ilości segmentów.

- Jeśli nie umiemy obsługiwać ciągłych transferów z dowolnym offsetem, możemy nakazać rozdzielać takie żądania.
- Czasem zdarzyć się mogą bardziej ezoteryczne wymagania co do łączenia transferów. Wówczas przydać się może ustawienie funkcji testującej:

  ```
  typedef int (merge_bvec_fn) (request_queue_t *q, struct bio *bio,
                               struct bio_vec *bvec);
  ```

- Dla urządzeń preferujących korzystanie z DMA, połączone transfery muszą mieć odpowiedni alignment.

  ```
  void blk_queue_dma_alignment(request_queue_t *q, int mask);
  ```

5.5 Moduły

5.5.1 Wprowadzenie.

Co to są moduły i do czego służą?

Moduły, są to dynamiczne części jądra. Ładowane ”na żądanie” w czasie działania systemu. Załadowane moduły funkcjonują jakby były częścią jądra. Mechanizm modułów umożliwia zmniejszenie rozmiarów jądra, bez zmniejszania jego funkcjonalności.

Zmiana mechanizmu modułów w jądrze 2.6

W jądrze 2.5 system ładowania modułów jądra został od nowa zaimplementowany, co oznacza że mechanizm budowania jądra w dużym stopniu się zmienił w porównaniu mechanizmu z 2.4. Do ładowania i wyładowywania modułów potrzebny jest nowy zbiór narzędzi. Makefile napisane do 2.4 nie będą działać w 2.6.

Nowy mechanizm ładowania modułów jest dziełem Rusty Russel.

5.5.2 Najprostszy moduł - stare i nowe.

Przeanalizujmy przykład przerabiany na laboratoriach (`http://rainbow.mimuw.edu.pl/SO/PUBLIC-SO/2004-05/07_modules/p1_hello/`). Jeśli chodzi o plik hello_mod.c, to najważniejszą różnicą jest to, że `module_init` i `module_exit` muszą być użyte zamiast `init_module` i `cleanup_module` (używanych w jądrze 2.4).

Większe są różnice między plikami Makefile. Skompilowany moduł będzie miał nazwę hello.ko i jest wynikiem kompilacji pliku hello.c i linkowania z vermagic. `KERN_SRC` wskazuje na katalog źródeł jądra.

STARA WERSJA; PLIK MAKEFILE:

```
MODULES=hello_mod.o

# KERNELDIR can be speficied on the command line or environment
ifndef KERNELDIR
        KERNELDIR = /usr/src/linux
endif
# The headers are taken from the kernel
        INCLUDEDIR = $(KERNELDIR)/include

#Install dir
VERSIONFILE = $(INCLUDEDIR)/linux/version.h
VERSION     = $(shell awk -F\" '/REL/ {print $$2}' $(VERSIONFILE))
INSTALLDIR = /lib/modules/$(VERSION)/misc

CFLAGS=-D__KERNEL__ -DLINUX -O3 -I$(INCLUDEDIR)

all: $(MODULES)

install: $(MODULES)
        install -d $(INSTALLDIR)
        install -c $(MODULES) $(INSTALLDIR)

clean:
        rm -f $(MODULES) *~
```

NOWA WERSJA; PLIK MAKEFILE:

```
KERN_SRC=/usr/src/linux

obj-m = hello.o

build:
        make -C $(KERN_SRC) SUBDIRS=‘pwd‘ modules

clean:
        rm -f *.o *.ko *.mod.o *.mod.c .*.{cmd,flags}
```

STARA WERSJA; PLIK HELLO_MOD.C:

```
#define MODULE
#include <linux/module.h>
/*
 * To jest najprostszy mozliwy modul. Po prostu dziala.
 */

int init_module(void) {
  printk("<1>Hello world\n");
  return 0;
}

void cleanup_module(void) {
  printk("<1>good bye\n");
}
```

NOWA WERSJA; PLIK HELLO.C:

```
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/init.h>

static int __init hellomod_init(void) {
        printk("<1>Hello world\n");
        return 0;
}
static void __exit hellomod_exit(void) {
        printk("<1>good bye\n");
}
module_init(hellomod_init)
module_exit(hellomod_exit)
MODULE_LICENSE("GPL");
```

Z istotniejszych zmian trzeba zaznaczyć że zmieniło się nazewnictwo. Zamiast .o (object) moduły mają rozszerzenia .ko (kernel object). Wbudowany w jądro system kompiluje wpierw moduł, później linkuje do vermagic.o. To tworzy specjalną sekcję w obiekcie modułu zawierającą informacje takie jak wersja kompilatora, wersja jądra, czy wywłaszczanie jądra jest używane, i tym podobne.

5.5.3 Po co to wszystko?

Po stworzeniu moduł (.ko), może być załadowany i wyładowany przy użyciu nowych narzędzi do modułów (module-init-tools). Starsze narzędzia, używane w 2.4 nie mogą być już używane do ładowania i wyładowywania modułów w jądrze 2.6. Nowe narzędzie ładujące moduły stara się minimalizować występowanie ”race conditions”, które mogą pojawić się gdy urządzenie jest otwierane, a odpowiadający mu moduł zaczął się wyładowywać (po upewnieniu się uprzednio że nie jest przez nikogo używany). Jedną z przyczyn pojawienia się ”race conditions” jest to że manipuluje się licznikiem użytkowania modułu w nim samym (poprzez MOD_DEC/INC_USE_COUNT).

W 2.6, moduł nie musi modyfikować licznika odwołań. Jest to robione poza kodem modułu. Każdy kod, który próbuje się odwołać do modułu musi najpierw wywołać: try_module_get(&module). Jeśli ta funkcja się załamie, to znaczy że moduł został wyładowany. Jeśli nie, kod może korzystać z modułu. Analogicznie licznik odwołań do modułu może być zmniejszony przez funkcję module_put().

Można także się spotkać z opinią, że nowe podejście do modułu jest bardziej elastyczne.

6 Nowe obsługiwane urządzenia i funkcje

6.1 Wewnętrzne urządzenia

- **Ulepszona obsługa PCI** (”Peripheral Component Interconnect”) - Hot-Plug PCI i zarządzanie energią
- Obsługa wielu **AGP** (”Accelerated Graphics Ports” oddzielne wysokiej prędkości rozszerzenie szyny PCI)
- **ISA Plug-and-Play** (”Industry Standard Architecture”), Wsparcie dla urządzeń plug and play takich jak pierwsze karty ISAPnp został udoskonalony w jądrze 2.6
- Obsługa innych szyn jak **MCA** (”Microchannel Architecture”) i **EISA** (”Extended ISA”)

Można powiedzieć, że Linux stał się w pełni systemem ”Plug-and-Play”. Dodatkowo obsługuje teraz liczne nowe rozszerzenia BIOSów, np.:

- **Obsługa prostej flagi rozruchu (Simple boot flag)** Specyfikacja SBF jest rozszerzeniem BIOSu x86 które pozwala na szybszy rozruch systemu. Robi to poprzez oznaczanie pola w CMOS mówiącego ”Rozruch był w porządku, pomiń rozległy POST przy następnym rozruchu”
- **Obsługa EDD** Obsługa dla rozszerzonych usług napędów dyskowych BIOS (BIOS Enhanced Disk Driver Services (EDD)). Eksportuje informacje o tym co BIOS sadzi który napęd jest rozruchowy i inne użyteczne informacje do /sys/firmware/edd

6.2 Zewnętrzne urządzenia

- **USB 2.0** (zamiast USB 1.1) 480Mbit/s (12)
- specjalny system plików usbfs (zamiast usbdevfs)
- **USB storage** zmieniło swoje zachowanie. Urządzenie które jest odłączone i ponownie podłączone nie jest związane ze starym węzłem /dev.
- **USB storage** także otrzymał kilka usprawnień wydajności.

6.3 Urządzenia bezprzewodowe

Zwiększono siłę obsługi urządzeń bezprzewodowych przez wprowadzenie jednego wspólnego API. Dzięki temu narzędzia będą obsługiwały więcej urządzeń.

Pojawiła się już nie eksperymentalna obsługa Bluetooth (krótko zasięgowy bezprzewodowy protokół).

6.4 Dźwięk i Multimedia

- **ALSA (Advanced Linux Sound Architecture)**, czyli sterowniki dźwięku zostały włączone do jądra 2.6. Ich najważniejsze własności to:
  - dobrze działają na SMP
  - obsługują wiele kart dźwiękowych naraz i umożliwiają sprzętowe miksowanie
  - obsługują USB audio i urządzenia MIDI
  - ALSA może też emulować starsze sterowniki OSS, od których się już odchodzi.
- **AGP 3.0** Zostało włączone generyczne wsparcie dla AGP 3.0.

6.5 LVM, ELVM, Device mapper

Device Mapper umożliwia tworzenie nowych wirtualnych urządzeń na podstawie już istniejących urządzeń blokowych. Korzysta z niego LVM2 (Linux Logical Volume) pozwalający zarządzać logicznymi woluminami skojarzonymi z urządzeniami blokowymi, partycjami.

LVM oraz ELVM (Enterprise LVM) są szczególnie przydatne jeśli w systemie jest wiele dysków twardych, których wspólną powierzchnię chcielibyśmy logicznie widzieć inaczej, niż to wynika z pojemności urządzeń. Niemniej jednak LVM może się również okazać pomocne w zarządzaniu jednym, czy dwoma dyskami.

7 Systemy plików

Kilka dodatkowych systemów plików dodano do 2.6. Obecnie są obsługiwane: ext2, ext3, reiserfs, jfs, xfs, minix, romfs, iso9660, udf, msdos, vfat, ntfs (ro), adfs, amiga ffs, apple macintosh hfs, BeOS befs (ro), bfs, efs (ro), cramfs, free vxfs, os/2 hpfs, qnx4fs, sysvfs, ufs.

Dzięki możliwości obsługi adresowania 64-bitowego urządzeń blokowych nawet na architekturach 32-bitowych można teraz obsługiwać 16TB systemy plików.

7.1 EA - Extended Attributes

Extended attributes są parami (nazwa, wartość) dołączanymi do plików lub folderów. Mogą służyć do przechowywania systemowych obiektów takich jak właściwości plików wykonywalnych czy ACL (access control lists), ale również do przechowywania zdefiniowanych przez użytkownika właściwości.

Często nie chcemy, albo nie możemy przechowywać w pliku pewnych meta-informacji - np. listy osób, które edytowały plik, wraz z datami edycji, poprzednia nazwa pliku etc.

EA są wspierane na systemach plików ext2, ext3, XFS, ReiserFS. Niestety potrzebne są nowsze narzędzia do zarządzania plikami (np. `cp`). Należy zatem uważać, by nie stracić EA używając starych narzędzi.

7.2 ACL Access Control Lists

Na UNIXowych systemach prawa dostępu do plików są realizowane przez tzw. file mode. Jest jednak zbyt słaby mechanizm dla niektórych aplikacji. ACL pozwala na bardziej szczegółowe nadawanie uprawnień do plików i katalogów poszczególnym użytkownikom i grupom.

ACL korzysta z EA.

7.3 EXT3

- System plików ext3 zyskał obsługę indeksowanych katalogów, co oferuje znaczące przyrosty wydajności na systemach plików z katalogami zawierającymi duże ilości plików.(aby skorzystać z tej opcji (htree), potrzebujesz e2fsprogs w wersji przynajmniej 1.32)
- Obecne systemy plików można przekonwertować komendą

```
tune2fs -O dir_index /dev/hdXXX
```

- Najnowsze e2fsprogs można znaleźć pod adresem: http://prdownloads.sourceforge.net/e2fsprogs
- Systemy plików ext2 i ext3 maja nowa politykę alokacji plików (”Alokator Orlova”) który umieszcza podkatalogi bliżej siebie na dysku. To oznacza, ze operacje na wielu plikach w danym katalogu są dużo szybsze jeśli zostało to drzewo stworzone w jądrze 2.6.

- Dodano możliwość zmiany rozmiaru niezamontowanej partycji Ext3. Z dodatkowym patchem na jądro również dla Ext2 jest taka możliwość.

7.4 Reiserfs

- Obsługa zapisów większych niż 4KB, co oznacza przyrost prędkości na zapisach dużych plików, szczególnie w trybie dopisywania, powinno także być bardziej zgodne z architekturami wieloprocesorowymi (SMP)
- Obsługa zmiennego rozmiaru bloku. (tj. Możesz wybrać dowolny rozmiar bloku z zakresu `1024 .. PAGE_CACHE_SIZE`, musi być potęgą 2).

7.5 NTFS

Nowy przepisany sterownik NTFS, został włączony do 2.6. Głównymi zaletami są:

- Bezpieczny w kontekście SMP i powtórnych wejść (reentrant safe)
- Obsługa rozmiarów klastra większego niż 4kB
- Pełna obsługa rzadkich plików na W2K/XP/W2K3
- Obsługa mmap()
- Bardziej stabilny i dużo szybszy niż poprzednia wersja.
- Nadal tylko do odczytu, jednak z bezpiecznym zapisywaniem pliku nie zmieniającym jego rozmiaru.

7.6 sysfs

Opisany w podrozdziale 5.2.2.

7.7 NFS

- Podstawowe wsparcie dla NFSv4 (serwer i klient)

7.8 FAT

Obsługa CVF (Compressed VFAT) została usunięta. Oznacza to, że nie będzie już dostępu do partycji DriveSpace.

7.9 JFS, XFS

Obsługa JFS (journaling file system) i XFS została dodana.

8 Inne nowości

8.1 Laptopy

8.1.1 Sterowanie częstotliwością CPU (CPU frequency scaling)

Niektóre procesory maja metody sterowania ich napięciem/częstotliwością. Jądro 2.6 przedstawia interfejs do tej cechy. Ta funkcjonalność obejmuje opcje takie jak Intel speedstep i Powernow! obecna w mobilnych Athlonach AMD. Dodatkowo, poza wariantami x86, istnieje możliwość obsługi procesorów architektury ARM

Możesz znaleźć demona przestrzeni użytkownika monitorującego żywotność baterii i regulującego zgodnie z tym częstotliwość pod adresem: `http://sourceforge.net/projects/cpufreqd`

8.1.2 Laptop Mode

Od wersji 2.6.6 pojawił się tryb dla laptopa pozwalający oszczędzać energię. Polega na sterowaniu częstotliwością i ograniczaniu pewnych operacji do minimum, np. dostępu do dysku (różne systemy plików lubią raz na jakiś czas coś sobie zmieniać).

8.2 Szybsze wywołania systemowe

- Systemy które obsługują rozszerzenie SYSENTER (Właściwie Intel Pentium-II i wyższe, oraz Athlony AMD) mają teraz szybsze metody przejścia z przestrzeni użytkownika do przestrzeni jądra kiedy wykonywane jest wywołanie systemowe.
- Bez zaktualizowanej biblioteki glibc będzie to niezauważalne.

8.3 Nagrywanie CD

- Jens Axboe dodał możliwość używania DMA przy nagrywaniu CD na urządzeniach ATAPI. Nagrywanie powinno być o wiele szybsze niż w 2.4 a także mniej Podatne na wyczerpanie bufora itp.
- Z nowym cdrecord, nie jest także potrzebny ide-scsi by używać nagrywarkę CD na IDE.
- Zgrywanie ścieżek audio z płyt CD używa teraz DMA i powinno być zauważalnie szybsze.

8.4 Udoskonalone monitorowanie systemu

- `lm_sensors`
  - sterowniki czujników hardwareowych
  - odczytują np. status zasilania, napięcia, temperaturę, etc.
  - Dostarczane z jądrami dystrybucji od lat, `lm_sensors` jest już częścią kodu. Ma jednak inny interfejs (`/sysfs` zamiast `/proc`).
  - obsługuje także więcej układów
  - `http://www.xs4all.nl/~thospel/ASIS/bin/psensors` to wygodny skrypt do przetwarzania nowych pól sysfs.
- IPMI. (Intelligent Platform Management Interface)
  - IPMI to standard monitorowania sprzętu w systemie.
  - Strona domowa pod adresem `http://openipmi.sourceforge.net`
  - Specyfikacja sprzętowa pod adresem `http://www.intel.com/design/servers/ipmi/spec.htm`

8.5 Softwareowe wstrzymywanie systemu (suspend mode)

- zapis stanu maszyny w partycji wymiany (w swapie) przez Sysrq-d lub przez shutdown.
- przywracanie po rebootowaniu
- zupełnie softwareowo, bez użycia APM
- niestety to jeszcze wersja eksperymentalna
- więcej na `http://falcon.sch.bme.hu/~seasons/linux/swusp.html`

9 Porady praktyczne

- System budowy jądra został ulepszony:
  - ulepszone `make xconfig` i `make gconfig`
  - domyślnie tworzony jest zImage
  - domyślnie make nie jest verbose (gadatliwy), żeby włączyć: `set KBUILD_VERBOSE=1` lub `make V=1`
  - `make katalog/` kompiluje pliki do katalogu `katalog/`
  - `make dep` nie jest potrzebny już w ogóle
- Wzrosła ilość numerów głównych (major) urządzeń z 255 do 4095, a ilość numerów minor z 255 do ponad miliona.
- zamiast `linux/malloc.h` dołączamy `linux/slab.h`
- Szybszy Zegar wewnętrzny Zwiększono stałą `HZ` z 100 do 1000, przez to
  - częściej wykonywany switching
  - lepsza interakcja

  Pojawiła się w związku z tym zmienna `jiffy_64`, gdyż zakres wartości `jiffy` kończył się zbyt szybko.
  Jest też nowa funkcja `ndelay()` - czekanie przez nanosekundy.

10 Podsumowanie

Szybciej

- NUMA
- Scheduler O(1) → dużo lepsze osiągi na architekturach serwerowych, i nie tylko.
- HyperThreading

Więcej

- 32-bitowe x86 mogą adresować do 64GB RAM
- systemy plików wielkości do 16TB
- więcej urządzeń - nr major i minor

Lepiej

- NFS - szybszy, bardziej skalowalny i bezpieczny
- obsługa systemów plików z jurnalowaniem - bezpieczniejsze dane

Desktop/Laptop

- ulepszona interakcyjność, szczególnie w aplikacjach multimedialnych (wywłaszczanie, scheduler, HZ, scheduler I/O)
- nowe multimedia (ALSA, lepsza obsługa Joysticka i inne)
- obsługa nowych urządzeń (na USB 2.0, AGP 3.0, etc)
- lepsza obsługa bezprzewodowych urządzeń
- oszczędzanie energii