Wykład 14
Rozwój systemu w kolenych wersjach jądra

Plan wykładu

- Wstęp
- Zmiany związane z urządzeniami wejścia wyjścia.
- Pamięć podręczna wpisów katalogowych.
- Alokator plastrowy (ang. slab allocator)dla pamięci jądra.
- Drobnoziarniste mechanizmy blokowania dla celów SMP - wirujące blokady (ang. spin locks) oraz semafory.
- Wywłaszczanie jądra.
- Systemy plików z kroniką.

Wstęp

- Linuks jest utrzymywany przez setki ochotników z całego świata. Całością prac kieruje Linus Torvalds, do którego należy decyzja akceptacji zmian w kodzie. Linus przekazuje część uprawnień osobą odpowiedzialnym za poszczególne podsystemy. Np. osoba odpowiedzialna za podsystem IA64 (procesory Itanium) akceptuje zmiany w tym podsystemie.
- W schemacie numeracji jądra Linuksa numery kończące się liczbą parzystą oznaczają wersję stabilną a nieparzystą wersję rozwojową. Po sfinalizowaniu prac nad wersją rozwojową staje się ona kolejną w wersją stabilną (np. 2.5 => 2.6). Po jakimś czasie od wersji stabilnej odłącza się kolejna wersja rozwojowa (kolejną będzie 2.7). Przez pewien czas praca przebiega równolegle nad dwiema wersjami.
- Zmiany w wersji stabilnej teoretycznie polegają na wprowadzaniu poprawek i np. dodawaniu sterowników nowych urządzeń. Interfejsy pomiędzy podsystemami jądra nie ulegają zmianie (np. w przypadku sterowników jest to struktura file_operations).
- Podczas prac nad wersją rozwojową (i w końcu w nowej wersji stabilnej) wprowadzone są znaczne zmiany w interfejsach łączących podsystemy jądra. Może to prowadzić np. do konieczności zmian w kodzie źródłowym wszystkich sterowników urządzeń.
- W powyższy sposób z jądra „pozbywane są” sterowniki przestarzałych urządzeń, których nikt nie utrzymuje, np. dysk twardy XT.

Nowe urządzenia wejścia/wyjścia i nowe typy urządzeń

- Wynikają z pojawienia sią nowych szyn (PCI, USB, PCMCIA) i portów (SATA, AGP).
  - urządzenia typu „plug nad play” - system (albo BIOS) przydziela numery przerwań, portów we-wy i adresy pamięci.
  - urządzenia typu hot-plug
  - urządzenia z możliwością sterowania poborem energii (ACPI).
  - możliwość enumeracji wszystkich urządzeń podłączonych do szyny.
  - urządzenia z własną translacją adresów (AGP)
- Wprowadzenie powyższych standardów wymagało dodania nowych interfejsów do jądra.
  - Przeszukiwanie szyny.
  - Reakcja na podłączenie/odłącznie urządzenia do systemu.
  - Przełączenie w tryb oszczędzania energii.

Alokator plastrowy (ang. slab allocator)

- Wprowadzony po raz pierwszy w systemie Solaris 2.4 i zaimplementowany w Linuksie od wersji 2.2
- Uniwersalny alokator pamięci kmalloc/kfree jest prosty w użyciu ale niezbyt wydajny.
  - Przeznaczony jest dla bloków pamięci o dowolnym (<128KB) rozmiarze i nie wykorzystuje faktu, że zdecydowana większość alokacji dotyczy struktur o określonych rozmiarach. Przykłady to i-węzły (struct inode), bufory (struct buffer_head), otwarte pliki (struct file), pakiety sieciowe (struct sk_buff).
  - Zastosowanie odrębnego alokatora dla każdej z tych struktur spowodowałoby, że ten odrębny alokator obsługiwałby żądania o stałym rozmiarze, co znacznie uprościłoby jego implementację oraz pozwoliłoby na znaczny wzrost wydajności.
- Dodatkowy alokator plastrowy przeznaczony jest do alokacji/dealokacji bloków o stałym rozmiarze (rozmiar ustalony jest w momencie tworzenia instancji alokatora zwanej pamięcią podręczną).
- Pamięć podręczna dla bloków stałego rozmiaru jest podzielona na plastry (składające się z kilku stron pamięci). Wewnątrz plastra przechowywane są obiekty o stałym rozmiarze. ***Zwiększa to znacznie wydajność alokacji/dealokacji.***
  - Ponadto zoptymalizowano przydział tak, aby optymalnie wykorzystać pamięć podręczną procesora (różne przesunięcie pierwszego obiektu od początku plastra)

API alokatora plastrowego (jądro 2.2)

```
kmem_cache_t * kmem_cache_create(const char *name, size_t size, size_t offset,
unsigned long flags, void (*constructor)(void *, kmem_cache_t *, unsigned long
flags), void (*destructor)(void *, kmem_cache_t *, unsigned long flags) );

moje_cache=kmem_cache_create(„Moje struktury”, sizeof(struct moja_stuktura), 0, 0,
NULL, NULL);
```

- Stworzenie nowej pamięci podręcznej o nazwie name dla obiektów o rozmiarze size, (offset=0). constructor i destructor to adresy funkcji inicjalizujących i deinicjalizujących obiekt, w pamięci jądra rzadko używane. Alokator stara się re-użytkować obiekty aby zaoszczędzić na pamięci. flags to bitowa suma SLAB_NO_REAP (jądro nie próbuje odzyskiwać całkowicie wolnych plastrów), SLAB_HWCACHE_ALIGN (wyrównaj obiekty do granicy rozmiaru wiersza sprzętowej pamięci cache) SLAB_CACHE_DMA (przydzielaj obiekty w pamięci dostępnej dla ISA DMA).

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
moj_obiekt=kmem_cache_alloc(moje_cache,GFP_KERNEL);
```

- Alokacja obiektu, flagi takie same jak dla kmalloc (GFP_KERNEL/GFP_ATONMIC).

```
void kmem_cache_free(kmem_cache_t *cache, const void *object);
kmem_cache_free(moje_cache, moj_obiekt);
```

- Zwolnienie obiektu

```
void *kmem_cache_destory(kmem_cache_t *cache);
kmem_cache_destroy(moje_cache);
```

- Usunięcie pamięci podręcznej (instancji alokatora) i zwolnienie pamięci wszystkich płyt.

Pamięć podręczna wpisów katalogowych (ang. dentry cache)

- Jej wprowadzenie wiąże się z dużą złożonością algorytmu tłumaczenia ścieżki na numer i-węzła (algorytm *namei*). Realizacja tego algorytmu wymaga wykonania metody lookup dla każdego członu-podkatalogu ścieżki. np. /home/wkwedlo/plik.txt składa się z cztere
- Aby przyspieszyć tę translację do warstwy VFS dodano nowy obiekt: pozycji katalogów (ang. dentry = directory entry).
- Jeden obiekt wpisu katalogowego zawiera translację nazwa_wpisu => i_węzeł w pamięci. W odróżnieniu od i-węzła, wpisu katalogowy istnieje w pamięci.
- Niektóre operacje i-węzła, które posługiwały (zwracały) się strukturami i-node, posługują się (zwracają) struktury wpisów katalogowych
- Obiekty wpisów katalogowych (nawet nie używane z licznikiem odniesień równym zero są przechowywane w pamięci podręcznej).
- Podczas translacji ścieżki na i-węzeł przeszukiwana jest najpierw pamięć podręczna. Wykorzystuje się tablice mieszające, których kluczem jest para <dentry_katalogu, nazwa_członu_ścieżki>.

Semafory

- Do zbioru operacji (down, up) dodano nową down_interruptible powodującą obudzenie zablokowanego procesu w reakcji na sygnał - szczególnie użyteczna dla sterowników.
- Operacja down_trylock powraca natychmiast jeżeli wejście do semafora jest niemożliwe.
- W wersji jednoprocesorowej semafory stosujemy do ochrony danych jądra w sytuacji, gdy podczas wykonywania sekcji krytycznej wykonywana jest funkcja jądra mogąca przełączyć kontekst.
- W wersji wieloprocesorowej, dodatkowo możemy stosować semafory do synchronizacji kodu wykonującego się na wielu procesorach.
- Semafory czytelników i pisarzy - odrębne operacje dla czytelników (down_read/up_read) i pisarzy (down_write/up_write). Stosowane gdy niektóre procesy (czytalnicy) nie modyfikują chronionych danych - w takim przypadku można pozwolić na jednoczesne wykonanie się wielu czytelników.
- Przesłanki do wykorzystania semaforów są następujące:
  - w chronionej sekcji krytycznej może nastąpić przełączenie kontekstu (wtedy nie wolno użyć wirujących blokad)
  - Sekcja krytyczna nie występuje wewnątrz przerwania (wtedy nie wolno użyć semaforów)
  - Przewidywany czas oczekiwania na wejście do sekcji krytycznej jest długi.

Wirujące blokady - spin locks

- Przeznaczone do synchronizacji kodu wykonującego się na różnych procesorach. Oparte są na aktywnym czekaniu (podobnie do blokady jądra w Linuksie 2.0 - patrz poprzedni wykład) przez jeden procesor, aż drugi opuści sekcje krytyczną. Kod chroniony przez wirującą blokadę ***nie ma prawa*** wywołać funkcji przełączających kontekst.
  - W związku z tym na architekturze jedno procesorowej ***pętla aktywnego oczekiwania jest redukowana*** do kodu pustego (bo jeden procesor i tak wszedłby w posiadanie blokady). Wejście do wirującej blokady wyłącza tylko wywłaszczanie jądra (w wersji 2.6).

```
spin_lock_t blokada=spin_lock_init;
// ...............
spin_lock(blokada); // Wejście do sekcji krytycznej, mogę czekać w pętli
// Tu sekcja krytyczna, tylko jeden procesor, pozostałe aktywne czekają
spin_unlock(blokada); // Wyjście z sekcji krytycznej
```

- Istnieją wersje wyłączające/włączające przerwania na lokalnym procesorze na czas posiadania blokady (spin_lock_irq/spin_unlock_irq)
- Istnieją wersje (spin_lock_irqsave/spin_lock_irqrestore) zapisujące stan rejestru znaczników przed wejściem do sekcji krytycznej i odtwarzające go po wyjściu z sekcji. Stosujemy je (wraz z funkcję local_irq_disable) gdy nie wiemy, czy w danej ścieżce wykonania jądra przerwania będą wyłączone czy też nie.
- Uwaga w odróżnieniu od blokady BKL w wersji 2.0 wirujące blokady ***nie są rekurencyjne***. Jeżeli procesor, który posiada blokadę spróbuje do niej wejść ponownie dojdzie do zakleszczenia !!!

Synchronizacja sterownika z handlerem przerwania na maszynie SMP

- Często sterownik urządzenia (np. operacje read/write) modyfikuje struktury danych do których dostęp ma także handler przerwania. W maszynie SMP proste wyłączenie lokalnych przerwań nie wystarcza - instrukcja cli wyłącza przerwania na aktualnym procesorze, a przerwanie może odebrać drugi.

Operacja write/read

```
spin_lock_irq(blokada)
// Sekcja krytyczna
spin_unlock_irq(blokada)
```

Handler przerwania

```
spin_lock(blokada)
// Sekcja krytyczna
spin_unlock(blokada)
```

- W operacji write/read musi być zastosowana wersja wyłączająca przerwania lokalne. W innym przypadku mógłby wykonać się scenariusz: procesor wchodzi w posiadanie blokady => przerwanie odebrane przez ten sam procesor => ten sam procesor ponownie usiłuje posiąść blokadę => zakleszczenie (ang. deadlock) :) :) :) :)

- W przypadku kompilacji jądra nie-SMP (i zakładając że nie jest włączone wywłaszczenie kodu jądra w wersji 2.6) , wykorzystanie makrodefinicji preprocesora sprawia że:
  - Operacje lock/unlock w handlerze przerwania nie wygenerują żądnego kodu
  - Operacja lock/unlock w kodzie write/read wygenerują instrukcje cli/sti

Wirujące blokady - podsumowanie

- Wirujące blokady stosowane są gdy:
  - blokada przetrzymywana jest przez krótki czas.
  - synchronizacji podlega kod wykonywujący się na różnych procesorach.
  - w sekcji krytycznej nie może nastąpić przełączenie kontekstu.
- Blokady czytelników-pisarzy (ang. readers-writers lock). Podobnie jak w przypadku semaforów, wirujące blokady występują w wersji czytelników (typ rwlock_t, inicjalizacja poprzez rw_lock_init).
  - wszystkie operacje mają odrębne postacie dla czytelników i pisarzy (np. read_lock/ read_unlock, write_lock/write_unlock, read_lock_irq/read_unlock_irq, ...
  - jednocześnie wiele czytelników (tzn. ścieżek wykonania jądra czytających daną strukturę danych) może wejść w posiadanie blokady.
  - alternatywnie blokadę może posiadać jeden pisarz.
- W przypadku kompilacji wersji jedno procesorowej jądra kod aktywnego czekania nie jest generowany, ponieważ mamy gwarancje że (jedyny w systemie) procesor uzyska blokadę.

Wywłaszczalne (ang. preemptible) jądro

- W wersjach jądra <2.6 proces wykonujący się jądra nie może zostać wywłaszczony, wbrew swojej woli. Może jedynie zrzec się procesora wywołując schedule (np. interruptible_sleep_on, down, kmalloc(,GFP_KERNEL)).
- W wersji 2.6 wprowadzono możliwość skompilowania jądra z możliwością wywłaszczenia. Zmniejsza ona czas reakcji procesu na zdarzenia, co jest ważne dle procesów czasu rzeczywistego.
- Wydawać by się mogło, że wprowadzenie możliwości wywłaszczenia znacznie skomplikuje synchronizację. Tak jednak nie jest.
  - Wykorzystano sprytną sztuczkę: jeżeli kod jądra jest poprawnie zsynchronizowany w trybie SMP przy pomocy wirujących blokad, to zablokowanie wywłaszczenia na czas wykonywania sekcji krytycznej chronionej blokadą, sprawi, że kod będzie bezpieczny w trybie z możliwością wywłaszczenia jądra.
  - Jedyny wyjątek, to kod operujący na danych prywatnych dla procesora, na którym się wykonuje - taki kod nie jest synchronizowany przy pomocy wirtualnych blokad. Przykładem jest kod odwołujący się do kolejki procesów gotowych (w wersji 2.6 każdy procesor ma swoją własną kolejkę). W przypadku takiego kodu, należy użyć funkcji preempt_disable/preempt_enable.
- Mamy cztery kombinacje: wywłaszczanie/bez oraz 1 procesor/SMP. Jądro 2.6 optymalizuje poszczególne przypadki. Np. dla jądra skompilowanego dla jednego procesora z możliwością wywłaszczania spin_lock wywołuje preempt_disable (i nic poza tym).

Ext3 i inne systemy plików z kroniką

- Cel - minimalizacja prawdopodobieństwa katastrofalnej awarii systemu plików, skrócenie czasu naprawy systemu po awarii. Uwaga: systemy z kronikowaniem nie zastąpią UPSa !!!
- Specjalny obszar zwany kroniką (ang. journal) w EXT3 jest to plik.
- Zmiana w systemie plików przebiega w sposób następujący: (a) zapisz (commit) bloki dyskowe które zmieniasz do kroniki - zapisywane są bloki składające się na ***transakcję*** (b) jeżeli pierwszy commit powiódł się wykonaj właściwy commit do systemu plików (c) jeżeli drugi zapis się powiódł to możesz (niekoniecznie od razu) usuń bloki z kroniki (kronika działa jak trochę podobnie jak kolejka FIFO.
  - Jeżeli awaria nastąpiła przed zakończeniem (a) to nic się nie dzieje - nastąpi utrata danych, ale system plików pozostanie w stanie spójnym.
  - Jeżeli awaria nastąpi pomiędzy (a) - (b), to przy naprawie systemu plików wykonaj ponowny commit transakcji z kroniki.
- EXT3 ma trzy możliwości funkcjonowania kroniki: (a) *journal* - kronikowane są bloki z danymi jak i meta-danymi (tablica i-węzłów, bitmapy i-węzłów i bloków, bloki indeksowe), (b) *ordered* - kronikowane są wyłącznie bloki z metadanymi, ale bloki z danymi są zapisywane jako pierwsze - minimalizuje prawdopodobieństwo utraty danych w pliku. (c) *writeback* - kronikuje wyłącznie bloki z meta-danymi, po awarii zasilania system plików zachowa spójność ale jest duża szansa na utratę danych plików zapisywanych w momencie awarii.
- Inne systemy z kronikowaniem to XFS, JFS, ReiserFS, Reiser4

Literatura godna polecenia

- Robert Love, Linux Kernel. Przewodnik programisty, Helion 2004

Opis jądra 2.6. W odróżnieniu od następnej pozycji, nie tylko omawia konkretne mechanizmy jądra, ale stara się tłumaczyć jak z nich korzystać. ***Godne polecenia.***

- Daniel Bovet, Marco Cesati, Linux Kernel, wydawnictwo RM, 2001.

Opis jądra 2.2. Najnowsze trzecie wydanie (niestety na razie tylko angielskie, tytuł Understanding Linux kernel) dotyczy wersji 2.6.

- Allessandro Rubini, Jonathan Corbet, Linux device drivers, 2nd edition, O'Reilly, 2001.

Opisuje problemy związane z budową sterowników urządzeń. Dotyczy jądra 2.5, ale zawiera uwagi dotyczące wersji wcześniejszych wersji 2.0 oraz 2.2. Dostępne legalnie w Internecie. Najnowsze trzecie wydanie (na razie niedostępne w sieci) dotyczy wyłącznie jądra 2.6. ***Godne polecenia.***

- Claudia Salzberg Rodriguez, Gordon Fischer, Steven Smolski, The Linux® Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures, Prentice Hall, 2005.

Opis jądra w wersji 2.6