Nowe jądro 2.6 - programowanie jądra i sterowników

TECHNIKA JĄDROWA

W pierwszych trzech odcinkach Techniki jądrowej pisaliśmy o samodzielnym tworzeniu modułów jądra dla sterowników, których funkcje będą wywoływane przez aplikacje. Teraz zajmiemy się przetwarzaniem asynchronicznym. Dzięki niemu funkcje w jądrze mogą pracować niezależnie od procesów obszaru użytkownika.

EVA-KATHARINA KUNST, JÜRGEN QUADE

Automatyczne wykonywanie procesów w jądrze to oszczędność czasu. Dowiódł tego zintegrowany z jądrem 2.4 serwer WWW *khttpd*, znany też jako Tux. Zanikł on w jądrze 2.6 – lecz technika wykonywania poszczególnych funkcji, a nawet całych aplikacji w jądrze, została rozwinięta i udoskonalona. Programiści, którzy muszą zaimplementować ściśle czasowo zaplanowane procedury lub cyklicznie przetwarzać dane, tworzyć statystyki lub przesyłać informacje, korzystają z funkcji asynchronicznych.

Porządki

Linus Torvalds w wersji 2.6 położył kres panującej w wersji 2.4 dzikiej rozbudowie funkcji jądra. Oprócz przedstawionych w ostatnim odcinku Techniki jądrowej [1] przerwań (IRQ sprzętowych), istnieją obecnie inne techniki podstawowe: IRQ programowe i wątki jądra.

Na czym polegają różnice? Podczas przetwarzania sprzętowych IRQ z reguły nie są dopuszczalne dalsze przerwania na tym samym procesorze. Programowe IRQ i wątki jądra mogą natomiast być same przerywane przez przerwania. IRQ programowe są wykonywane w tak zwanym kontekście przerwania, zaś wątki jądra w kontekście procesu (zobacz Ramka „Pojęcia z jądrowego świata"). Rozróżnienie to jest kluczowe dla programisty jądra, gdyż niektóre

Pojęcia z jądrowego świata

Kontekst: kontekst (otoczenie) określa, do jakich usług i zasobów dany fragment kodu ma dostęp.

Kontekst użytkownika: kod aplikacji jest wykonywany w tak zwanym kontekście użytkownika. Ma do dyspozycji usługi systemu operacyjnego, określone w interfejsie funkcji systemowych.

Konteks jądra: kod jądra systemu operacyjnego jest uruchamiany w kontekście jądra. W kontekście jądra wyróżnia się kontekst procesu i kontekst przerwania.

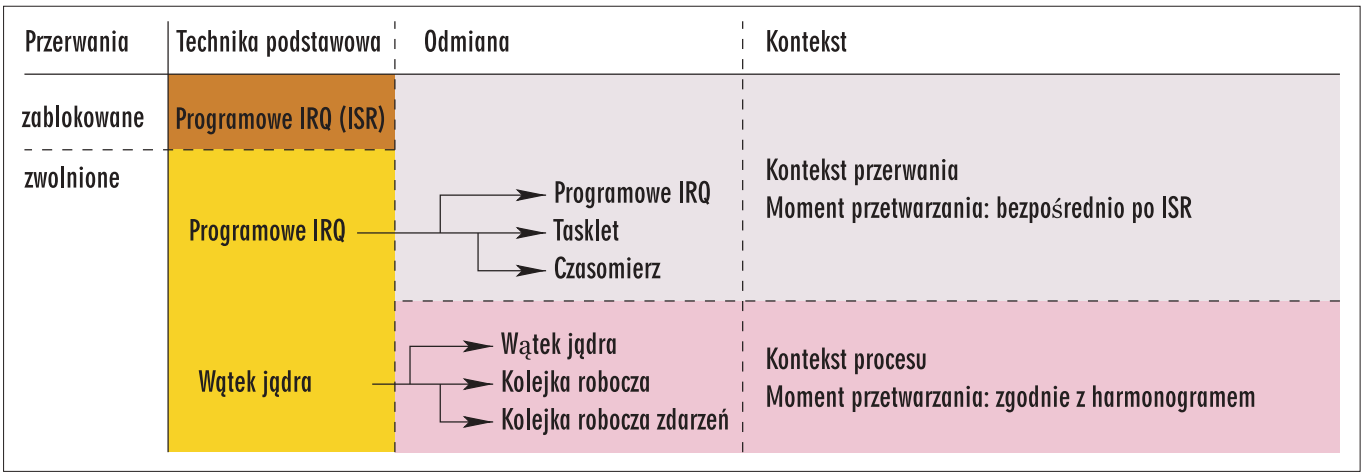
Kontekst procesu: zwykłe otoczenie, w którym wykonuje się kod jądra, nazywa się kontekstem procesu. Są w nim do dyspozycji wszystkie funkcje jądra systemu, łącznie z przetwarzaniem kodu opóźnienia na określony czas (oczekiwania) oraz przesyłaniem danych między obszarami pamięci aplikacji i jądra. Wywoływane przez aplikacje funkcje sterownika urządzeń (np. *DriverRead ()*), wywołania funkcji systemowych, wątki jądra oraz kolejki robocze są przetwarzane w kontekście procesu.

Kontekst przerwania: w kontekście przerwania nie są dostępne wszystkie usługi jądra. Funkcje w tym kontekście nie mogą „spać". Nie wolno im też wywoływać funkcji typu *kmalloc ()*, gdyż te procedury w pewnych okolicznościach wymagają wprowadzania sztucznych opóźnień.

Dostęp do obszarów pamięci aplikacji (np. poprzez funkcję *copy_from_user ()*) jest całkowitym tabu. Przetwarzanie w kontekście przerwania powinno trwać jak najkrócej, aby nie zakłócało zwykłych procesów obliczeniowych. Procedury, które wykonują się w kontekście przerwania, to tak zwane procedury obsługi przerwania (ISR), programowe IRQ, tasklety i czasomierze.

Obszar jądra: jako obszar jądra określa się te obszary pamięci, do których procedury wewnątrz jądra mogą uzyskiwać bezpośredni dostęp (przez wskaźnik), nie korzystając przy tym ze specjalnych funkcji. Tylko jądro ma dostęp do obszaru jądra. Dostęp do obszaru użytkownika dają mu funkcje *copy_to_user ()*, *copy_from_user ()*, *put_user ()* i *get_user ()* – ale tylko do pamięci aktywnego w danym momencie procesu obliczeniowego.

Obszar użytkownika: obszary pamięci, do których aplikacja ma bezpośredni dostęp, na przykład przez wskaźnik, nazywamy „obszarem użytkownika". Do obszaru jądra aplikacja nie ma dostępu, nawet z prawami superużytkownika.



Rysunek 1: Asynchroniczne przetwarzanie w jądrze jest możliwe przy użyciu wielu technik. Programowe IRQ i wątki jądra mają trzy różne warianty.

Listing 1: Moduł taskletu

```
01 #include <linux/version.h>
02 #include <linux/module.h>
03 #include <linux/init.h>
04 #include <linux/interrupt.h>
05
06 MODULE_LICENSE("GPL");
07
08 static void TaskletFunction
( unsigned long data )
09 {
10     printk("Tasklet
called...\n");
11 }
12
13 DECLARE_TASKLET( TlDescr,
TaskletFunction, 0L );
14
15 static int __init ModInit(void)
16 {
17    printk("ModInit
called\n");
18    tasklet_schedule
( &TlDescr );// Tasklet wywołać
jak najszybciej
19     return 0;
20 }
21
22 static void __exit ModExit(void)
23 {
24     printk("ModExit called\n");
25     tasklet_kill( &TlDescr );
26 }
27
28 module_init( ModInit );
29 module_exit( ModExit );
```

funkcje jądra są niedostępne w kontekście przerwania.

Techniki podstawowe, czyli IRQ programowe i wątki jądra, występują w różnych odmianach. Tak więc IRQ programowe dzieli się na IRQ programowe, tasklety i czasomierze. Natomiast wątki jądra dzielą się na wątki jądra, kolejki robocze i kolejki robocze zdarzeń.

Programiści jądra powinni w miarę możności unikać IRQ programowych i zamiast nich używać taskletów i czasomierzy. Tasklet przejmuje dłużej trwające zadania procedury obsługi przerwania (ISR). Czasomierz jest przydatny do wykonywania zadań w określonym momencie lub w określonych odstępach w czasie.

Gdy w grę wchodzą złożone zadania, programiści sięgają po wątki jądra. Jako kolejka robocza wątek może wywoływać funkcje przez jądro systemu operacyjnego. Jeszcze mniej pracy mamy w wypadku predefiniowanej kolejki roboczej zdarzeń; należy jednak przynajmniej dokładnie określić położenie zadania w tej kolejce.

Przerwania programowe i tasklety

Programowe IRQ lądują jako pierwsze w kolejce po przetworzeniu wszystkich procedur obsługi przerwań sprzętowych. Na 32 możliwe typy w źródłach jądra (*linux/interrupt.h*) wymienionych jest sześć wstępnie zdefiniowanych typów programowych IRQ. Dla programistów sterowników interesujące są tam tasklety i czasomierze.

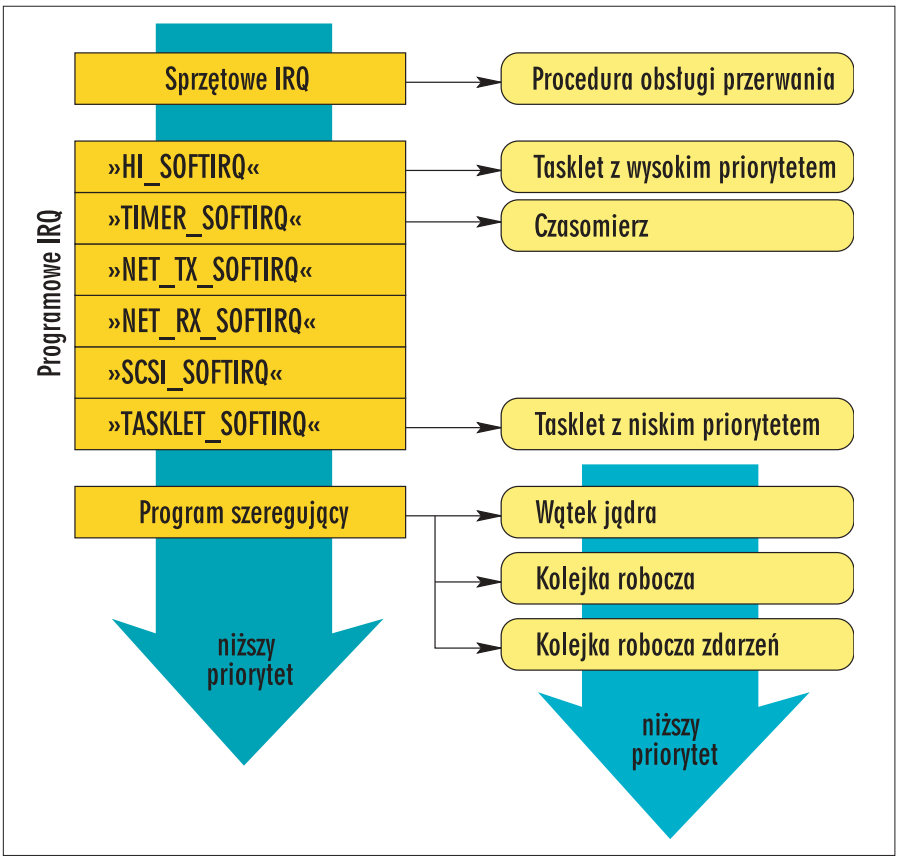
Tasklety przejmują złożone zadania od procedur obsługi przerwań sprzętowych. Wykonywanie długich procedur tego rodzaju prowadzi do zwiększania czasów oczekiwania na obsługę przerwań. Z tego względu należy tego unikać. Większy sens ma podział takich procedur na dwie części. Pierwsza przeprowadza kluczowe (pod względem czasowym) operacje, gdy przerwania są zablokowane. Pozostałe obliczenia mają miejsce w drugiej części, po powrocie z przerwania. Do realizacji tej części, zwanej dawniej „dolną połową" (ang. bottom half; zobacz R amka „Większa przenośność"), nadają się właśnie tasklety.

Jądro zapewnia, że w danym momencie wykonuje się maksymalnie jeden tasklet. Dotyczy to także systemów wieloprocesorowych. Można jednak korzystać z różnych taskletów jednocześnie. Ponieważ tasklety należą do grupy programowych IRQ, mają w systemie drugi co do wysokości priorytet, bezpośrednio po przerwaniach sprzętowych. Mimo to tasklety mają dwa poziomy priorytetu.

Zawsze po kolei

Tasklety z wysokim priorytetem wchodzą jako pierwsze programowe IRQ do kolejki, bezpośrednio za przerwaniami sprzętowymi. Jako ostatnie wstawiane są tasklety z niskim priorytetem. Zaleca się, aby programiści w miarę możliwości korzystali z wariantów o niskim priorytecie.

Aby wskazać tasklet, w procedurze obsługi przerwania należy podać dwie informacje: adres funkcji, która ma wywołać jądro, oraz parametr, który jądro przekaże w momencie wywołania funkcji. Obydwie dane znajdują się w strukturze *struct tasklet_struct*. Makro *DECLARE_TASKLET* ułatwia podanie zawartości tej struktury danych.



Rysunek 2: Przerwania programowe i sprzętowe mają wysokie priorytety. Dopiero kiedy wszystkie przerwania zostaną przetworzone, przychodzi kolej na procesy obliczeniowe, do których zaliczają się także wątki jądra.

Jądro rozpoznaje tasklet po funkcji *tasklet_schedule ()*. Ta funkcja musi zostać wykonana we właściwym momencie, najczęściej w ramach procedury obsługi przerwania sprzętowego. Jasny i kompletny przykład znajduje się na Listingu 1, gdzie kod wywołuje funkcję schedule taskletu już w czasie inicjacji modułu w wierszu 18.

Uwaga: sytuacja wyścigu

Tasklety grożą uwikłaniem w sytuację wyścigu. Po usunięciu modułu z pamięci (*rmmod*), a następnie wywołaniu przez jądro zaplanowanego przez ten moduł taskletu procesor powinien przetwarzać kod, który już nie istnieje. To jednak niemożliwe. Jądro reaguje komunikatem o błędzie.

Twórca sterownika musi zadbać o to, żeby jego moduł przed usunięciem skasował wszystkie tasklety, których nie przetwarzało jeszcze jądro. Służy do tego funkcja *tasklet_kill ()*. Sterownik może ją bezpiecznie wykonywać, nawet jeśli tasklet jeszcze nie został zaplanowany do wykonania.

Po skompilowaniu i załadowaniu [2] modułu z Listingu 1 w dzienniku syslog [4] powinien pojawić się komunikat *Tasklet called...*. Do śledzenia zawartości dzienników systemowych świetnie nadaje się polecenie *tail -f* plik dziennika. A więc: otwieramy kolejne okno z interpreterem poleceń i wpisujemy *tail -f /var/log/messages* lub coś w tym stylu.

Zadeklarowana w 13 wierszu struktura *TlDescr* wskazuje funkcję *TaskletFunction ()* jako punkt wejścia taskletu. W wierszu 18 funkcja *tasklet_schedule ()* zapewnia, że jądro będzie przetwarzać tasklet jako IRQ programowe z niskim priorytetem. Aby użyć programowego IRQ z wysokim priorytetem, stosujemy funkcję *tasklet_hi_schedule ()*.

Licznik jiffy

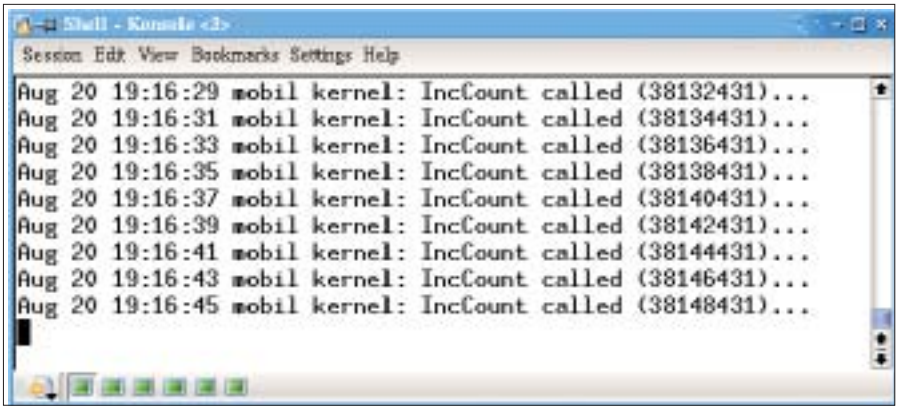
Równie ważne jak tasklety są czasomierze, druga odmiana programowych IRQ. Za ich pomocą programista żąda wykonania przez jądro funkcji w określonym momencie. Jądro mierzy te momenty czasowe nie bezwzględnie, lecz względnie od chwili włączenia komputera. Jako podstawa służy liczba cyklicznych przerwań zegarowych, które wystąpiły od chwili włączenia komputera: jądro mierzy je za pomocą zmiennej *jiffies*.

Aby wywołać funkcję w określonym momencie, moduł musi zdefiniować strukturę

Listing 2: Funkcje czasomierza

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/timer.h>
04 #include <linux/sched.h>
05 #include <linux/init.h>
06
07 MODULE_LICENSE("GPL");
08
09 static struct timer_list MyTimer;
10
11 static void IncCount (unsigned
long arg)
12 {
13     printk("IncCount called
(%ld)...\n",
14         MyTimer.expires);
15     MyTimer.expires = jiffies +
(2*HZ); // 2 sekundy
16     add_timer( &MyTimer );
17 }
18
19 static int __init ktimerInit
(void)
20 {
21     init_timer( &MyTimer );
22     MyTimer.function = IncCount;
23     MyTimer.data = 0;
24     MyTimer.expires = jiffies +
(2*HZ); // 2 sekundy
25     add_timer( &MyTimer );
26     return 0;
27 }
28
29 static void __exit ktimerExit
(void)
30 {

31    if( del_timer_sync(&MyTimer))
32         printk("Active timer
deactivated.\n");
33     else
34         printk("No timer
active.\");
35 }
36
37 module_init( ktimerInit );
38 module_exit( ktimerExit );
```



Rysunek 3: Moduł czasomierza co dwie sekundy zapisuje wynik w dzienniku systemowym. Po wartości w nawiasach można rozpoznać, że w jądrze minęło już 2000 tyknięć zegara.

danych typu *struct timer_list* i zainicjować ją funkcją *init_timer ()*. Następnie należy określić adresy funkcji do przetwarzania (pole *function*), moment przetwarzania (*expires*) i dane wejściowe funkcji (*data*). Moment określa się bezwzględnie jako wartość *jiffies*. Gdy moduł ma wyzwolić funkcję w określonym momencie, musi rozwiązać proste zadanie arytmetyczne: zsumować aktualną chwilę i wartość względną.

Gdy czasomierz jest zdefiniowany w postaci struktury danych i zainicjowany, funkcja *add_timer ()* przekazuje wynik do jądra. Kiedy tylko nastąpi lub zostanie przekroczony moment wskazany w polu Expires, jądro wywołuje określoną funkcję, dokładnie raz. Jeśli ma wykonywać funkcję cyklicznie, musi zainicjować jej strukturę *timer_list*, wstawiając do niej następny moment wywołania, a potem znowu przekazać do jądra.

Cykliczne wyniki

Na Listingu 2 wiersz 24 zapewnia, że jądro wykona funkcję *IncCount()* dwie sekundy później. Wiersz 15 powtarza to, aby funkcja była wykonywana cyklicznie co dwie sekundy. Dowodem wykonania funkcji jest wynik w dzienniku systemowym. Uwaga: żądanie wielokrotnego przetworzenia jednego obiektu czasomierza przez jądro doprowadzi do załamania systemu.

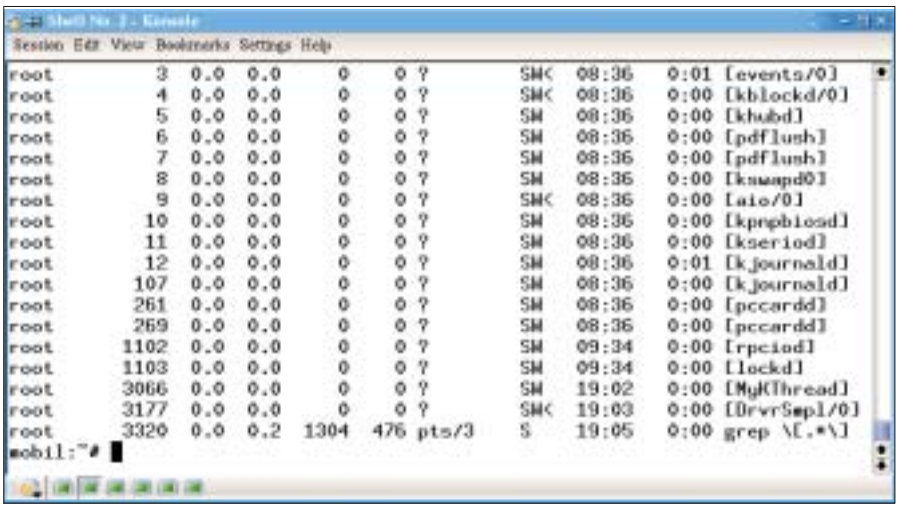
Doświadczony czytelnik z pewnością zauważy, że również w czasomierzach występuje fragment krytyczny. Moduł może być usunięty z pamięci tylko wtedy, gdy w systemie nie ma już realizowanej funkcji czasomierza. Rozwiązanie: funkcja *del_timer_sync ()* usuwa funkcję jako przywiązaną do czasomierza.

Wątki jądra

W przeciwieństwie do taskletów i czasomierzy, wątki jądra mogą i muszą w czasie wykonywania stosować funkcje opóźniania. Wątek jądra odpowiada jednemu wątkowi poziomu użytkownika, z tą różnicą, że wykonuje się całkowicie w przestrzeni jądra. Jest on, jak każdy inny proces obli-

Listing 3. Wątek jądra

```
01 #include <linux/module.h>
02 #include <linux/version.h>
03 #include <linux/init.h>
04 #include <linux/completion.h>
05
06 MODULE_LICENSE("GPL");
07
08 static int ThreadID=0;
09 static wait_queue_head_t wq;
10 static DECLARE_COMPLETION( OnExit );
11
12 static int ThreadCode( void *data )
13 {
14     unsigned long timeout;
15     int i;
16
17     daemonize("MyKThread");
18      allow_signal( SIGTERM ); // funkcja od wersji
2.5.61
19     printk("ThreadCode startet ...\n");
20     for( i=0; i<\<>10; i++ ) {
21         timeout=HZ; // jedna sekunda
22         timeout=wait_event_interruptible_timeout(
23             wq, (timeout==0), timeout);
24         printk("ThreadCode: woke up ...\n");
25         if( timeout==-ERESTARTSYS ) {
26             printk("got signal, break\n");
27             ThreadID = 0;
28             break;
29         }
30     }
31     complete_and_exit( &OnExit, 0 );
32 }
33
34 static int __init kthreadInit(void)
35 {
36     init_waitqueue_head(&wq);
37     ThreadID=kernel_thread(ThreadCode, NULL,
CLONE_KERNEL );
38     if( ThreadID==0 )
39         return -EIO;
40     return 0;
41 }
42
43 static void __exit kthreadExit(void)
44 {
45     kill_proc( ThreadID, SIGTERM, 1 );
46     wait_for_completion( &OnExit );
47 }
48
49 module_init( kthreadInit );
50 module_exit( kthreadExit );
```



Rysunek 4: Polecenie *ps auxw* pokazuje oprócz procesów użytkownika także wątki jądra. Te ostatnie są na liście zadań wyróżnione nawiasami kwadratowymi.

czeniowy, reprezentowany przez strukturę zadania i pojawia się po wydaniu polecenia *ps auxw* także na liście zadań (zobacz Rysunek 4). Moment przetwarzania wątku jądra jest ustalany przez program szeregujący zadania.

Wątki te tworzy się w prosty sposób. Wystarcza wywołanie funkcji *kernel_thread()*. Funkcja ma trzy parametry:

- adres funkcji, którą ma wywołać program szeregujący zadania przy pierwszej aktywacji;
- argument dla wywoływanej funkcji i
- pole bitowe określające sposób tworzenia procesu obliczeniowego przez jądro.

Z reguły w polu bitowym wystarcza wpis *CLONE_KERNEL*; wówczas tworzenie wątku jest bardzo wydajne. Jądro nie musi kopiować ani tworzyć informacji o systemie plików (*CLONE_FS*), deskryptorów plików (*CLONE_FILES*) oraz żadnych procedur obsługi sygnałów (*CLONE_SIGHAND*).

Wartością zwrotną funkcji jest identyfikator nowego procesu (PID). Jest on potrzebny później, do usunięcia wątku jądra. Wartość zwrotna *0* oznacza, że utworzenie wątku nie udało się.

```
ThreadID = kernel_thread(
    ThreadCode, NULL, CLONE_KERNEL);
```

Po utworzeniu nowy wątek musi przeprowadzić inicjację. Najpierw funkcją *daemonize()* zwalnia wszystkie zasoby w obszarze użytkownika, które mu standardowo przekazało jądro.

Przemiana w demona

Funkcja daemonize oczekuje jako pierwszego parametru nazwy nowego wątku. Interpretuje ją – podobnie jak *printf ()* – jako ciąg formatowany; odnośne parametry są zawarte na liście argumentów. Z ciągu formatowanego funkcja tworzy nazwę wątku jądra. Jej maksymalna długość to 16 znaków. Resztę funkcja *daemonize ()* obcina.

Wątek jądra musi się zakończyć na czas lub zostać uśpiony. W przeciwnym razie odbierze cały czas procesora innym procesom, zwłaszcza aplikacjom użytkownika. Jak pokazaliśmy w trzecim odcinku techniki jądrowej [2], pomaga tu kolejka oczekiwania.

Na Listingu 3 w wierszu 22 wątek jądra jest usypiany za pomocą kolejki oczekiwania: *wait_event_interruptible_timeout()*. W praktyce istnieją lepsze kryteria dla usypiania i budzenia wątków jądra: na przykład oczekiwanie na dane nadchodzące przez połączenie TCP/IP.

Listing 4. Kolejka robocza

```
01 #include <linux/version.h>
02 #include <linux/module.h>
03 #include <linux/init.h>
04 #include <linux/workqueue.h>
05
06 MODULE_LICENSE("GPL");
07
08 static struct workqueue_struct *wq;
09
10 static void WorkQueueFunction( void *data )
11 {
12     printk("WorkQueueFunction<\\>n");
13 }
14
15 static DECLARE_WORK( work, WorkQueueFunction, NULL
);
16
17 static int __init ModInit(void)
18 {
19     wq = create_workqueue( "DrvrSmpl" );
20     if( queue_work( wq, &work ) )
21         printk("work has been queued ...\n");
22     else
23         printk("queue_work failed ...\n");
24     return 0;
25 }
26
27 static void __exit ModExit(void)
28 {
29     pr_debug("ModExit called\n");
30     if( wq ) {
31         destroy_workqueue( wq );
32         printk("workqueue destroyed\n");
33     }
34 }
35
36 module_init( ModInit );
37 module_exit( ModExit );
```

W poszukiwaniu końca

Problemem jest zakończenie wątku. Wszystkie procesy obliczeniowe, czy to w obszarze użytkownika, czy to wątki jądra, z reguły dochodzą do końca. Kiedy jeden proces chce zakończyć inny, przesyła mu sygnał. Jednak funkcja *daemonize()* zablokowała wszystkie sygnały. Wątek jądra musi dopiero znowu je dopuścić. W tym celu wywołuje funkcję *allow_signal()*.

To, czy w stanie uśpienia nadszedł sygnał, wskazuje wartość zwrotna makra *wait_event_interruptible_timeout ()*. Wiersz 25 służy do sprawdzania, czy odpowiada ona wartości *-ERESTARTSYS*, oznaczającej „zakończyć wątek". Także funkcja *signal_pending ()* stwierdza, czy sygnał nadszedł, czy też nie. Zmienna globalna *current* wskazuje na aktywny wątek.

```
if( signal_pending(current) ) {
    // nadszedł sygnał
} else {
    // brak sygnału
}
```

Wątek jądra kończy się przez opuszczenie instrukcją *return* wcześniej uruchomionej funkcji wątku. Sygnał można wysłać na przykład w interpreterze poleceń poleceniem *kill* lub wewnątrz jądra funkcją *kill_proc ()*. W obydwu wypadkach wątek jest wybierany identyfikatorem PID.

Obiekt Completion

I znowu pojawia się ryzyko, że powstanie niechroniony, krytyczny fragment kodu. Tak jak w pozostałych metodach programista musi zapewnić, że wątek jądra zakończy się przed usunięciem modułu. Wymaganie zakończenia jest realizowane w prosty sposób funkcją *kill_proc()*.

Ponieważ proces kończy się sam, moduł musi poczekać teraz na faktyczne zakończenie. Ochrona tego krytycznego fragmentu dodatkowo komplikuje się w systemach wieloprocesorowych, gdzie kod usuwania modułu z pamięci może być wykonywany na innym procesorze niż wątek jądra.

Z tego powodu społeczność programistów pod wodzą Torvalda opracowała obiekt Completion. Moduł jądra musi go globalnie zdefiniować makrem *DECLARE_COMPLETION* (wiersz 10 na Listingu 3). Po zakończeniu wątku wywołuje on funkcję *complete_and_exit ()* (wiersz 31). Jej parametrami są adresy obiektu Completion i kod zakończenia wątku.

Moduł funkcji usuwania wybiera funkcja *wait_for_completion()* (wiersz 46), która jako parametr otrzymuje adres obiektu Completion. Zwraca ona sterowania dopiero wtedy, gdy zostanie wywołana pasująca funkcja *complete_and_exit()*.

Nawet jeśli moduł uruchomi wiele wątków jądra, wystarcza tylko jeden obiekt Completion. Każde wywołanie *wait_for_completion ()* w module musi mieć pokrycie w wywołaniu funkcji *complete_and_exit ()* w wątku. Gdy moduł oczekuje na dwa wątki jądra, wówczas w funkcji usuwania dwukrotnie wywołuje funkcję *wait_for_completion ()*. W obydwu wątkach jeden raz musi zostać wywołana funkcja *complete_and_exit ()*.

Kolejka robocza zamiast snu

Usypianie wątku łączy się z pewnym kosztem. Zamiast wywoływać funkcję z niskim priorytetem w kontekście procesu, często lepiej wykorzystać kolejkę roboczą. Tego rodzaju wątek jądra przyjmuje adresy funkcji, aby je potem przetwarzać. Są do tego wymagane dwa obiekty: *struct work_queue_struct* odnosi się do kolejki roboczej; *struct work_struct* opisuje funkcję do przetwarzania.

Pamięć obiektu kolejki roboczej jest tworzona przez funkcję jądra *create_workqueue ()* (Listing 4, wiersz 19). Jej jedynym parametrem jest nazwa kolejki roboczej, względnie wątku jądra. Uwaga: nazwa może mieć długość maksymalnie dziesięciu znaków.

Pamięć obiektu roboczego udostępnia sam moduł. Do definiowania i inicjacji stosuje się w nim makro *DECLARE_WORK* (wiersz 15). Jego trzema parametrami są nazwa obiektu roboczego, adres wywoływanej funkcji oraz wartość parametru zwracana przez kolejkę roboczą do funkcji.

Funkcja *queue_work()* (wiersz 20) przenosi obiekt roboczy do kolejki roboczej. Jej parametry określają adres kolejki roboczej oraz adres obiektu roboczego. Przy następnej okazji kolejka robocza wywołuje określoną w obiekcie roboczym funkcję. Do jednej kolejki roboczej można wstawić dowolnie wiele funkcji (obiektów roboczych). Każdy obiekt jest wywoływany dokładnie raz, a następnie usuwany.

Obiekt roboczy nie może sam wstawić się do kolejki roboczej, gdyż wówczas zostałby bezpośrednio ponownie wywołany. Wniosek:

Listing 5: Kolejka robocza zdarzeń

```
01 #include <linux/version.h>
02 #include <linux/module.h>
03 #include <linux/init.h>
04 #include <linux/workqueue.h>
05
06 MODULE_LICENSE("GPL");
07
08 static struct work_struct work;
09
10 static void WorkQueueFunction( void *data )
11 {
12     printk("WorkQueueFunction\n");
13 }
14
15 static DECLARE_WORK( work, WorkQueueFunction, NULL );
16
17 static int __init ModInit(void)
18 {
19     if( schedule_work( &work )==0 )
20         printk("schedule_work not successful ...\n");
21     else
22         printk("schedule_work successful ...\n");
23     return 0;
24 }
25
26 static void __exit ModExit(void)
27 {
28     flush_scheduled_work();
29 }
30
31 module_init( ModInit );
32 module_exit( ModExit );
```

system jeszcze przetwarza tę jedną funkcję. Podana w obiekcie roboczym funkcja powinna także nie przechodzić w stan uśpienia: cały wątek kolejki roboczej zostałby uśpiony, a zatem opóźnione zostałyby wszystkie pozostałe funkcje, które się w niej znajdują.

Opóźnianie pracy

Jeśli funkcja musi być wykonana nie od razu, lecz z pewnym opóźnieniem, wówczas świetnym rozwiązaniem są kolejki robocze. Do ich obsługi używamy funkcji *queue_delayed_work()* zamiast *queue_work()*. Jako dodatkowy parametr przyjmuje ona limit czasowy (liczony w jiffy).

Także kolejki robocze należy likwidować przed usunięciem sterownika z pamięci: *destroy_workqueue()*. Ta funkcja oczekuje na przetworzenie wszystkich obiektów roboczych w kolejce. Programista musi się zatroszczyć o to, aby żaden egzemplarz jego sterownika nie wstawił ponownie obiektu roboczego do kolejki, co potencjalnie mogłoby sprawić nieprzewidziane problemy.

Po skompilowaniu kodu z Listingu 4 i załadowaniu wygenerowanego modułu można odkryć wątek jądra kolejki roboczej w tabeli procesów. Kolejka nosi nazwę *DrvrSmpl* (wiesz 1, Rysunek 4) i wyświetla się jako *[DrvrSmpl/0]*. Jeszcze jedna wskazówka: kolejki robocze można wykorzystywać w rozszerzeniach jądra wyłącznie pod warunkiem, że podlegają one licencji GPL lub BSD. Dla rozszerzeń prawnie ograniczonych pod względem modyfikacji i dystrybucji te możliwości są niedostępne.

Zakres zdarzeń

Na zakończenie należy wspomnieć jeszcze o kolejce roboczej zdarzeń (zobacz Listing 5). Jądro domyślnie dla każdego procesora tworzy taką kolejkę. Występuje ona zamiast znanego z wcześniejszych wersji demona *keventd*. Jej zaleta: odpada uciążliwe ustawianie kolejki roboczej.

Dzięki kolejce roboczej zdarzeń programista za pomocą funkcji *schedule_work ()* (wiersz 19) może bez ograniczeń przekazywać funkcje do przetwarzania w jądrze. Funkcja *schedule_work ()* jako jedyny parametr przyjmuje adres obiektu roboczego. Funkcja *schedule_delayed_work ()* uruchamia funkcję z opóźnieniem w czasie.

Do prac porządkowych służy zaś funkcja *flush_scheduled_work ()* (wiersz 28). Popycha ona naprzód przetwarzanie kolejki roboczej zdarzeń i czeka na zakończenie jej pracy.

W następnym odcinku

Wszystkie odcinki Techniki jądrowej wskazują na pewne szczególne wymaganie: unikanie sytuacji wyścigów i ochrona krytycznych fragmentów kodu. W następnym odcinku rzucimy okiem na służące do tego celu metody w jądrze 2.6. *(fjl)* ■

INFO

[1] Eva-Katharina Kunst i Jürgen Quade: „Technika jądrowa", odcinek 1, Linux Magazine nr 12 ze stycznia 2005

[2] Eva-Katharina Kunst i Jürgen Quade „Technika jądrowa", odcinek 2, Linux Magazine nr 13 z lutego 2005

[3] Eva-Katharina Kunst i Jürgen Quade „Technika jądrowa", odcinek 3, Linux Magazine nr 16 z maja 2005

AUTORZY

Eva-Katharina Kunst, dziennikarka i Jürgen Quade, profesor w Hochschule Niederrhein, należą od czasu powstania Linuksa do fanów oprogramowania Open Source.

Pomoc przy przenoszeniu

Ważne różnice między jądrami 2.4 i 2.6

Bottom Half: nie istnieją już tak zwane „dolne połówki". Zamiast nich w użycie wchodzą tasklety. Programista może wybrać wersję z wysokim lub niskim priorytetem. Może być pewien, że tasklet w danym momencie jest aktywny tylko jeden raz, bez względu na to, jaką ustawił częstość wykonywania. Tasklety są wykonywane – jak „dolne połówki" – w kontekście przerwania.

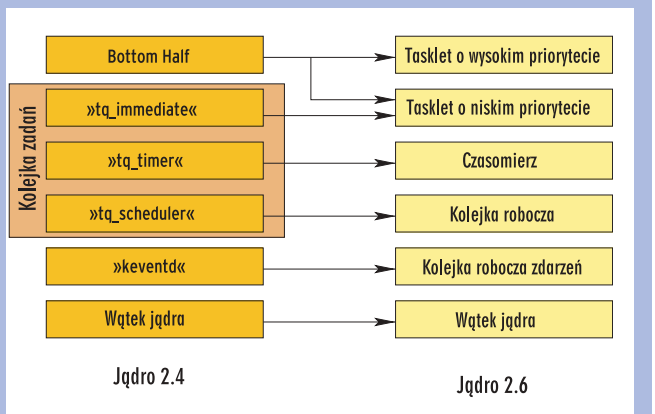
Kolejka zadań: czasy kolejek zadań minęły. Programiści najczęściej używali trzech ich wariantów: *tq_immediate*, *tq_timer* i *tq_scheduler*. Kolejka zadań *tq_immediate* może być zastąpiona przez tasklet. Zamiast *tq_timer* używa się obiektu czasomierza; jako moment na platformie x86 można teraz podawać dziesiętne wartości jiffy. Zamiast *tq_scheduler* można użyć kolejki roboczej zdarzeń.

Keventd: demon zdarzeń jądra *keventd* już nie istnieje. Zamiast niego używa się kolejki roboczej zdarzeń.

Jiffy: zmienna globalna *jiffies* była w poprzednich wersjach jądra wartością 32-bitową. Wniosek: na platformie x86 w normalnych okolicznościach po 407 dniach następowało przepełnienie licznika. Teoretycznie mogło wówczas dojść do niespójnego stanu systemu – dla programistów systemów wbudowanych nie była to szczególnie kusząca perspektywa.

Jądro 2.6 zawiera trzy zmiany. Po pierwsze zmienna *jiffies_64* to rozszerzony, 64-bitowy licznik. Po drugie, podwyższono częstotliwość taktów zegara, która zwiększa wartość licznika, ze 100 Hz do 1000 Hz. Po trzecie, licznik jest tak wstępnie inicjowany, że po pięciu minutach się przepełnia. W ten sposób programiści szybciej dostrzegają konsekwencję i mogą w takim wypadku usunąć błędy z kodu.

Daemonize: funkcja *daemonize ()* była w jądrze 2.4 bezparametryczna. Wątek jądra, z którego ta funkcja była wywoływana, sam ustawiał nazwę wątku. W jądrze 2.6 funkcja daemonize otrzymuje nazwę jako parametr i przejmuje zadanie ustawiania nazwy wątku.



Rysunek 5: Techniki przetwarzania synchronicznego w starym i nowym jądrze są odmienne. Każda możliwość ma jednak zastępnik.