Wykład siódmy

Mechanizmy dolnych połówek w Linuksie

Procedury obsługi przerwań muszą spełniać ścisłe wymagania czasowe. Ponieważ przerwania pojawiają się w systemie w sposób asynchroniczny, mogą one przerwać wykonywanie innych istotnych czynności. W szczególności ich obsługa blokuje odbiór przerwań korzystających z tej samej linii, a w niektórych przypadkach blokuje lokalnie wszystkie przerwania w systemie (dla jednego procesora). Dodatkowo sprzęt zgłaszający przerwanie również może narzucać ograniczenia co do czasu jego obsługi. To wszystko sprawia, że procedury obsługi przerwań muszą działać szybko i nie podlegają usypianiu. Jeśli obsłużenie przerwania wymaga bardziej czasochłonnych operacji, które mogą mimo to być odroczone, to ich wykonanie jest przenoszone do tzw. *dolnych połówek*[1] *(ang. bottom halves)*. Mechanizm dolnych połówek w Linuksie nie jest czymś innowacyjnym, podobne mechanizmy stosują również inne systemy operacyjne. Dzięki tym mechanizmom czynności czasochłonne nie są wykonywane bezpośrednio w procedurze obsługi przerwania, ale są odraczane do czasu, kiedy system będzie mniej obciążony (najczęściej zaraz po odblokowaniu przerwań). Dolne połówki gwarantują przełożenie wykonania złożonych czynności związanych z obsługą przerwania na później, ale nie wszystkie pozwalają określić kiedy dokładnie to wykonanie nastąpi[2]. Niestety, nie ma precyzyjnych reguł określających, które czynności należy wykonać w procedurze obsługi przerwania, a które przenieść do dolnej połówki. Można jednak określić cztery przesłanki, które mogą pomóc podjąć decyzję:

- czynności ściśle ograniczone czasowo należy wykonać w górnej połówce,
- czynności wymagające intensywnej komunikacji ze sprzętem również należy umieścić w górnej połówce,
- czynności, które nie mogą być przerwane przez inne przerwania z tej samej linii powinny być umieszczone w górnej połówce,
- czynności, które nie mają wyżej wymienionych ograniczeń można umieścić w dolnej połówce.

To czy podjęliśmy słuszną decyzję można stwierdzić badając wydajność systemu po zaimplementowaniu obsługi danego przerwania. W jądrze Linuksa w wersji 2.6 i nowszych istnieją cztery mechanizmy będące realizacjami systemu dolnych połówek: przerwania programowe[3] i tasklety[4], które wyparły mechanizm BH[5] istniejący w starszych wersjach systemu oraz kolejki prac (*ang. work queue*), które wyparły kolejki zadań[6] (*ang. task queue*). Istnieje również mechanizm liczników, które pozwalają odroczyć czynności na określony czas, ale ich opis zostanie przełożony na później. Mechanizm BH był prosty w użyciu, ale podlegał globalnej synchronizacji i pozwalał tylko na istnienie w systemie 32 dolnych połówek, które były tworzone statycznie. Został on zastąpiony przez dwa inne mechanizmy. Pierwszy z nich – przerwania programowe – są również alokowane statycznie podczas kompilacji jądra i wykorzystywane dosyć rzadko. W systemie mogą istnieć maksymalnie 32 przerwania programowe, spośród których obecnie wykorzystywanych jest tylko kilka (10 w jądrze w wersji 4.4). Przerwania te opisywane są strukturą *softirq_action*:

```
struct softirq_action
{
      void (*action)(struct softirq_action *);
};
```

Tablica 32 elementów tego typu, o nazwie *sortirq_vec,* jest umieszczona w pliku *kernel/softirq.c*. Jedyne pole tej struktury jest wskaźnikiem na funkcję nazywaną procedurą obsługi przerwania programowego, która implementuje operacje wykonywane w ramach tego przerwania. W starszych wersjach jądra istniało drugie pole, które było wskaźnikiem typu *void* * na dane dla tej funkcji. Prototyp funkcji obsługującej przerwanie programowe jest następujący:

*void softirq_handler(struct softirq_action *)*

Jedynym argumentem pobieranym przez tę funkcję jest wskaźnik na strukturę *softirq_action*. Przerwania programowe nie wywłaszczają się wzajemnie, ale na platformach równoległych możliwe jest współbieżne wykonanie kilku przerwań programowych (nawet tych samych) na osobnych procesorach. Zanim przerwanie programowe zostanie uruchomione jest oznaczane przez górną połówkę jako przeznaczone do uruchomienia. Czynność oznaczania nazywa się wyzwalaniem przerwania programowego. Uruchomienie przerwań oczekujących może się dokonać na trzy sposoby W każdym z tych przypadków wykonywana jest funkcja *do_softirq*(), która pobiera maskę oczekujących przerwań programowych, zachowuje ją w zmiennej lokalnej, zeruje oryginalną maskę (przy wyłączonych przerwaniach sprzętowych) i pobiera tablicę *softirq_vec*. Po wykonaniu tych czynności przegląda ona tablicę i sprawdza odpowiadające kolejnym indeksom pozycje w masce bitowej. Jeśli ta pozycja jest ustawiona, to uruchamiana jest odpowiednia procedura obsługi, a po jej wykonaniu zwiększany jest o jeden indeks w tablicy[7] i maska jest przesuwana w prawo o jedną pozycję. Numery przerwań programowych deklaruje się statycznie przy pomocy typu wyliczeniowego zawartego w pliku *linux/interrput.h*. Wartości tego zbioru określają priorytety tych przerwań (im wyższy priorytet, tym niższa wartość). Procedury obsługi przerwania programowego są rejestrowane za pomocą funkcji *open_softirq*(), która przyjmuje dwa argumenty: indeks tego przerwania, określony w wyżej wspomnianym typie oraz wskaźnik na procedurę obsługi. W starszych wersjach jądra ta funkcja przyjmowała trzeci argument wywołania, którym był wskaźnik na dane dla procedury obsługi przerwania programowego. Należy pamiętać, że procedury obsługi przerwań programowych działają w kontekście przerwania i w ich kodzie należy uwzględnić zagadnienia związane z synchronizacją. Wyzwolenie przerwania programowego następuje przez wywołanie funkcji *raise_softirq()*, która wyłącza przerwania, oznacza przerwanie do wykonania i ponownie włącza system przerwań. Jeśli ten system już jest wyłączony, to można zamiast niej użyć *raise_softirq_irqoff ()*.

Tasklety bazują na przerwaniach programowych, ale są prostsze w użyciu i nadają się do zadań, które nie są wykonywane z bardzo dużą częstotliwością, ani nie dają się podzielić na dużą liczbę wątków. Tasklety reprezentowane są za pomocą struktury typu *struct tasklet_struct*, podobnej do *softirq_action*, ale zawierającej cztery dodatkowe pola: wskaźnik na następny element, pole opisujące stan taskletu, oraz pole będące licznikiem odwołań (określa, czy tasklet jest zablokowany, czy nie). Pole *data*, które zostało usunięte w strukturze *struct softirq_action* w strukturze *struct tasklet_struct* istnieje i ma typ *unsigned long*. Pole stanu może przyjmować trzy wartości: 0, TASKLET_STATE_RUN i TASKLET_STATE_SCHED. Druga wartość wykorzystywana jest tylko w systemach wieloprocesorowych do określenia, że tasklet jest już wykonywany na jednym z procesorów, trzecia oznacza, że tasklet został zaszeregowany do wykonania. Istnieją dwa rodzaje taskletów: zwykłe tasklety i wysokiego priorytetu. Pierwsze umieszczane są w liście *tasklet_vec*, a drugie w liście *tasklet_hi_vec*. Umieszczenia pojedynczego taskletu na odpowiedniej liście dokonuje się za pomocą jednej z dwóch funkcji: *tasklet_schedule()* lub *tasklet_hi_schedule()*. Wywołanie tej samej funkcji dwukrotnie dla tego samego taskletu nie spowoduje jego dwukrotnego zaszeregowania, jeśli jeszcze się nie wykonał. Dwa takie same tasklety nie mogą działać współbieżnie, różne mogą. Tasklety możemy deklarować statycznie i inicjować za pomocą *DECLARE_TASKLET(name, func, data)* lub *DECLARE_TASKLET_DISABLED(name, func, data)*. Drugie makro inicjuje tasklet, którego wykonanie jest domyślnie zablokowane. Samej inicjacji taskletu można dokonać przy pomocy *tasklet_init(t, taskklet_handler, data)*[8]. Wykonanie zaszeregowanego taskletu można zablokować na pewien czas za pomocą funkcji *tasklet_disable*() lub *tasklet_disable_nosync*(), a następnie odblokować przy pomocy *tasklet_enable*(). Usunięcie taskletu z listy oczekujących na wykonanie dokonywane jest za pomocą *tasklet_kill*(). Funkcja implementująca czynność wykonywaną w ramach taskletu przyjmuje tylko jeden argument i jest nim liczba typu *unsigned long*.

Zarówno w przypadku przerwań programowych, jak i taskletów pojawia się problem przerwań programowych, które występują z bardzo dużą częstotliwością lub takich, które same się reaktywują. Problem ten rozwiązano w Linuksie odkładając ich obsługę w czasie i powierzając te dolne połówki pod opiekę wątkowi jądra *ksoftirqd* o najniższym możliwym priorytecie. Ten wątek co jakiś czas sprawdza, czy nie jest konieczne uruchomienie opisanych wcześniej dolnych połówek i uruchamia je, a następnie sam przechodzi w stan TASK_INTERRUPTIBLE.

Ostatnim opisywanym interfejsem dolnych połówek, ale za to najprostszym w obsłudze są kolejki prac. Kolejki prac odkładają wykonywane czynności do wątków jądra. Dolne połówki wykonywane w ramach takich wątków działają w kontekście procesu i mogą ulegać uśpieniu, ale nie korzystają z pamięci należącej do przestrzeni

1 „Zwykłe" procedury obsługi przerwań nazywane są „górnymi połówkami" (*ang. upper half*).
2 Niektóre mechanizmy dolnych połówek pozwalają określić po jakim czasie powinny być wykonane zlecone im czynności, ale nie gwarantują dokładności.
3 Proszę go nie mylić z rozkazami przerwań programowych, które powiązane są z omawianymi wcześniej wywołaniami systemowymi.
4 Jak się przekonamy nie mają one nic wspólnego z zadaniami (procesami).
5 Celem uniknięcia kolejnej pułapki semantycznej nie będę używał prawidłowej nazwy tego mechanizmu, która brzmi ... „mechanizm dolnych połówek" (!)
6 Tak, te kolejki nie mają nic wspólnego z listą zadań :-)
7 Właściwie jest to wskaźnik na strukturę *softirq_action*.
8 Parametr „t" oznacza wskaźnik na strukturę typu *tasklet_struct*.

użytkownika. Każda czynność, która ma zostać wykonana w ramach kolejki prac jest reprezentowana przez pojedynczy element takiej kolejki, która jest zrealizowana przy pomocy listy dynamicznej. Elementy kolejki wskazują na funkcje, które wykonują zlecone kolejce prace. Przeglądaniem kolejki i wywoływaniem tych funkcji zajmuje się specjalny wątek jądra, nazywany wątkiem roboczym, który realizuje funkcję *worker_thread()*. Wątki te zgrupowane są w tak zwane pule wątków. Każda taka pula zarządzana jest przez mechanizm nazywany *gcwq*. W systemach wieloprocesorowych dla każdego procesora jest tworzona osobna instancja takiego mechanizmu, oraz występuje jeden globalny mechanizm dla tzw. kolejek niepowiązanych (ang. *unbouded queues*). Każdy mechanizm gcwq posiada dwie pule wątków, jedną dla wysoko priorytetowych kolejek, drugą dla zwykłych kolejek. Każda kolejka prac jest reprezentowana przez strukturę *workqueue_struct*[9]. Jądro Linuksa posiada domyślną kolejkę prac, która obsługiwana jest przez wątek *kworker*, w starszych wersjach jądra nazwany *events*. Jeżeli jest to uzasadnione np. wydajnością, to programista może utworzyć dodatkową kolejkę, nawet z poziomu modułu jądra. Do tego służy funkcja *alloc_workqueue()*. Przyjmuje ona trzy argumenty wywołania. Pierwszy argument jest nazwą kolejki, oraz nazwą tzw. ratunkowego wątku roboczego (ang. *rescue work thread*). Takie wątki obsługują kolejki prac, w których zgromadzone są czynności związane z odzyskiwaniem pamięci operacyjnej (ang. *memory reclaim*). Drugi argument to flaga, lub suma bitowa następujących flag: WQ_NON_REENTRANT – domyślnie w środowisku wieloprocesorowym tylko jedna instancja funkcji implementującej pracę może być wykonywana na danym procesorze; jeśli ta flaga jest ustawiona, to w całym systemie (globalnie) może być wykonywana tylko jedna instancja danej pracy z tworzonej kolejki, WQ_UNBOUND – tworzona kolejka nie będzie powiązana z żadnym procesorem, WQ_FREEZABLE – prace zgromadzone w tworzonej kolejce powiązane będą z hibernowaniem systemu, WQ_MEM_RECLAIM – prace zgromadzone w kolejce związane będą z odzyskiwaniem pamięci, WQ_HIGHPRI – tworzona kolejka będzie kolejką wysoko priorytetową, WQ_CPU_INTENSIVE – prace obciążające procesor, które będą zgromadzone w tworzonej kolejce nie będą powodowały opóźnienia rozpoczęcia prac obsługiwanych przez tę samą pulę wątków. Ostania flaga nie wpływa na działania kolejek niepowiązanych. Trzeci argument funkcji *alloc_workqueue*() to liczba określająca ile maksymalnie pracy z danej kolejki może być współbieżne wykonane. Pojedynczy element kolejki prac, który reprezentuje pracę do wykonania może mieć typ *struct work_struct* lub *struct delayed_work*. Pierwszy typ strukturalny opisuje prace odłożone, tj. takie, dla których nie jest określony czas, po którym zostaną wykonane, a drugi prace opóźnione, o określonym czasie, po którym będą wykonane. Wątek roboczy po przebudzeniu wykonuje wszystkie czynności, które są na liście zbudowanej z takich elementów. Jeśli ta lista jest pusta, to od razu przechodzi w stan oczekiwania. Prace odłożone umieszczane w kolejce prac są tworzone statycznie i inicjowane, przy pomocy makrodefinicji *DECLARE_WORK(name, void (*func) (struct work_struct *))* lub tylko inicjowane za pomocą *INIT_WORK*(*struct work_struct *work, void (*func)(sturct work_struct *))*. Prace opóźnione, z kolei są deklarowane statycznie i inicjowane makrodefinicją *DECLARE_DEAYED_WORK(name, void (*func) (struct work_struct*))*, a tylko inicjowane za pomocą *INIT_DELAYED_WORK(struct delayed_work *, void (*func) (struct work_struct *))*. Funkcja implementująca pracę musi mieć następujący prototyp:

*void work_handler(struct work_struct *work)*

Szeregowania czynności w domyślnej (związanej z wątkiem *kworker*) kolejce możemy dokonać na dwa sposoby, za pomocą funkcji *schedule_work*() dla prac odłożonych lub *schedule_delayed_work()* dla prac opóźnionych. Czasem konieczne może się okazać opróżnienie domyślnej kolejki prac. Można tego dokonać za pomocą funkcji *flush_scheduled_work()*. Usunięcia zaszeregowanej pracy odłożonej można dokonać za pomocą funkcji *cancel_work_sync*(), a pracy opóźnionej za pomocą funkcji *cancel_delayed_work()* lub *cancel_delayed_work_sync*(). Do tworzenia nowej kolejki prac służy także makro *create_workqueue,* które w wersjach jądra wcześniejszych niż 3.9 było fukcją. Dla każdej utworzonej przy jego pomocy kolejki tworzony jest również osobny wątek roboczy na każdym procesorze. Ponieważ takie rozwiązanie jest kosztowne, to stworzono makro *create_singlethread_workqueue* (wcześniej to także była funkcja) tworzące kolejkę prac obsługiwaną przez pojedynczy wątek roboczy, działający na pierwszym procesorze w komputerze. Szeregowanie do kolejek utworzonych przy użyciu tych makr lub funkcji *alloc_workqueue*() odbywa się za pomocą funkcji *queue_work()* i *queue_delayed_work()*, które jako jeden z argumentów pobierają wskaźnik do kolejki, w której ma być zaszeregowana praca. Opróżnienie kolejki innej niż domyślna wykonywane jest przy pomocy *flush_workqueue()*. Ponieważ stworzenie kolejki jest kosztowne, należy starannie rozważyć zasadność wykonywania takiej operacji. W wielu zastosowaniach wystarcza kolejka domyślna. W wersjach jądra poprzedzających 3.9 utworzenie kolejki wiązało się ze stworzeniem dla niej jednego wątku roboczego, lub tylu takich wątków, ile jest procesorów w systemie komputerowym. W nowszych wersjach jądra wątki dodawane są i usuwane z puli przez mechanizm gcwq, który monitoruje obciążenie procesorów i tak stara się dobrać liczbę wątków roboczych, aby z jednej strony nie przeciążać ich, a z drugiej strony nie powodować nadmiernych opóźnień w realizacji zleconych pracy.
Mechanizm kolejek prac ulegał wielu zmianom. W starszych wersjach jądra funkcje implementujące prace posiadały parametr typu *void **, przez który był im przekazywany wskaźnik na dane. Wszystkie makra i funkcje tworzące i/lub inicjujące prace posiadały dodatkowy, trzeci parametr, przez który również przekazywany był wskaźnik na te dane. Dodatkowo, zarówno prace opóźnione, jak i odłożone opisywane były strukturami typu *struct work_struct*. Od wersji 2.6.20 do wersji 4.0 dostępne były makrodefinicje *PREPARE_WORK* i *PREPARE_DELAYED_WORK*. Służyły one do ponownej inicjacji prac, które przynajmniej raz były zaszeregowane i wykonane. Od wersji jądra 2.6.27 wprowadzono funkcje *int flush_work(struct work_struct *work)*, która oczekuje na zakończenie pracy odłożonej związanej ze strukturą, na którą wskaźnik jej przekazano przez parametr oraz *int flush_delayed_work(struct delayed_work *dwork)*, która nakazuje wykonanie i oczekuje na zakończenie pracy opóźnionej związanej ze strukturą, na którą wskaźnik jej przekazano przez parametr. W nowszych wersjach jądra zwracają one wartość typu *bool*. Wprowadzono także funkcje pozwalające na zaszeregowanie prac do wykonania na określonym procesorze w systemach wieloprocesorowych.

9 We wcześniejszy wersjach jądra ta struktura reprezentowała też wątki robocze.