Systemy Operacyjne 2

Dolne Połówki

Arkadiusz Chrobot

Katedra Systemów Informatycznych

18 kwietnia 2024

Plan

Wstęp

Współczesne systemy operacyjne dzielą obsługę przerwań sprzętowych na dwie części: górną i dolną połówkę. Tę pierwszą stanowią procedury obsługi przerwań, które muszą działać szybko, ponieważ linia IRQ z nimi związana, a niekiedy nawet cały system przerwań są wyłączone w trakcie ich wykonywania. Zazwyczaj wykonują one tylko niezbędne prace związane z obsługą przerwania, a pozostałe czynności są odraczane i wykonywane w dolnej połówce. Jądro Linuksa posiada kilka mechanizmów dolnych połówek. Jeden został już przedstawiony w ramach tych wykładów, to wątki przerwań. W tym wypadku górna połówka sprawdza, że przerwanie zostało zgłoszone przez urządzenie peryferyjne, z którym jest skojarzona i zwraca wartość, która powoduje, że jądro budzi wątek zajmujący się resztą czynności związanych z obsługą przerwania. Jest on wykonywany w kontekście procesu, więc może przejść w stan oczekiwania. Ten mechanizm pozwala jądru zająć się najpierw innymi czynnościami o wyższym priorytecie, niż obsługa przerwania.

Wstęp

W ramach tego wykładu zostaną przedstawione trzy inne mechanizmy dolnych połówek. Pozostałe, czyli liczniki czasu (ang. *timers*) będą omówione w przyszłości.
Nie ma ścisłych reguł odnośnie tego, które operacje związane z obsługą przerwań powinny być wykonywane w górnej, a które w dolnej połówce. Istnieją tylko pewne wskazówki:

1. jeśli czynność jest ograniczona czasowo, powinna być wykonana w górnej połówce,
2. jeśli czynność wymaga dostępu do urządzenia peryferyjnego, także powinna być wykonana w górnej połówce,
3. jeśli czynność nie może być przerwana przez to samo lub inne przerwanie, to powinna być w górnej połówce,
4. inne czynności mogą być wykonane w ramach dolnej połówki.

To, czy decyzja o zakwalifikowaniu danej czynności do górnej lub dolnej połówki była słuszna można sprawdzić jedynie badając wydajność jądra po zaimplementowaniu zmian.

Przerwania programowe

Przerwania programowe razem z *taskletami* zastąpiły mechanizm dolnych połówek (ang. *bottom halves*) z wersji jądra wcześniejszych niż 2.4, ale te pierwsze bardzo go przypominają. Są one statycznie deklarowane, co oznacza, że nie mogą być używane w modułach jądra. Ich liczba jest ograniczona do 32, ale to w zupełności wystarcza. Przykładowo, w wersji jądra 5.10 używanych jest tylko 10 (na podstawie wyniku `cat /proc/softirqs`). Wszystkie przerwania programowe są wykonywane w kontekście przerwania i w przypadku systemów wieloprocesorowych mogą być wykonywane równolegle. Każde takie przerwanie jest reprezentowane przez strukturę typu `struct softirq_action`, którego definicję zawiera listing 1.

```
1 struct softirq_action
2 {
3         void (*action)(struct softirq_action *);
4 };
```

Listing 1: Definicja typu `struct softirq_action`

Przerwania programowe

We wcześniejszych wersjach jądra w tej strukturze występowało dodatkowe pole typu `void *` o nazwie `data`. Nie było jednak używane, więc je usunięto. Pole, które pozostało jest wskaźnikiem na funkcję nazywaną *procedurą obsługi przerwania programowego* (ang. *softirq handler*). Jej prototyp jest następujący:

```
void softirq_handler(struct sortirq_action *);
```

Nazwa tej funkcji jest zazwyczaj inna, niż ta podana w prototypie. We wcześniejszych wersjach jądra parametr miał typ `void *`. Kiedy usunięto pole `data`, to musiał on zostać zastąpiony innym wskaźnikiem i zdecydowano się na wskaźnik do struktury typu `struct softirq_action`. Oznacza to, że funkcja przyjmuje jako argument adres struktury, która zawiera wskaźnik na nią. Plik `kernel/sofirq.c` zawiera definicję tablicy `softirq_vec`, która ma 32 elementy typu `struct softriq_action`. Jest to tablica deskryptorów przerwań programowych. Jej indeksy określają priorytety tych przerwań.

Przerwania programowe

Przerwanie programowe o indeksie 0 ma najwyższy priorytet. W pliku nagłówkowym `linux/interrupt.h` zdefiniowany jest typ wyliczeniowy, którego elementy są używane jako takie indeksy. Tablica `softirq_vec` jest używana przez funkcję `__do_softirq()` odpowiedzialną za wywołanie procedur obsługi przerwań programowych. Zanim takie przerwanie zostanie uruchomione musi być *wyzwolone* przez górną połówkę przy pomocy funkcji `raise_softirq()`, która jako argument pobiera element wspomnianego typu wyliczeniowego, który odpowiada danemu przerwaniu programowemu. Funkcja ta wyłącza system przerwań i ustawia na 1 wartość jednego z bitów w 32-bitowej mapie bitowej o nazwie `pending`. Pozycja tego bitu odpowiada priorytetowi przerwania programowego, a także jego indeksowi w tablicy `softirq_vec`. Po wykonaniu tej czynności funkcja `raise_softirq()` przywraca działanie systemu przerwań. Jeśli przerwania są wcześniej wyłączone, to zamiast niej można użyć funkcji `raise_softirq_off()`. Wyłączenie systemu przerwań zapobiega powstaniu sytuacji hazardowych.

Przerwania programowe

Zazwyczaj przerwania programowe są uruchamiane po zakończeniu górnej połówki, ale w niektórych przypadkach, gdy jądro musi wykonać ważne czynności, to uruchomienie może być odroczone na nieokreślony, ale zwykle krótki, czas. To, czy są przerwania programowe oczekujące na obsłużenie jądro sprawdza po zakończeniu górnej połówki, w wątku jądra `ksoftirqd` lub w kodzie, który bezpośrednio sprawdza, czy są takie przerwania. Jeśli jest to prawdą, to wywoływana jest funkcja `__do_softirq()`, która wyłącza przerwania, kopiuje zawartość bitmapy `pending` do lokalnej zmiennej, zeruje tę bitmapę i włącza przerwania, a następnie w pętli sprawdza wartość najstarszego bitu lokalnej kopii bitmapy. Jeśli jest on ustawiony na 1, to funkcja wywołuje procedurę obsługi przerwania programowego, której adres jest zapisany w tablicy `softirq_vec`. Potem przesuwa ona w prawo o jeden bit kopię bitmapy i przechodzi do następnego elementu tablicy, dodając 1 do wskaźnika na tę tablicę. Pętla kończy się, gdy nie ma więcej przerwań programowych do obsłużenia.

Przerwania programowe

Nowe przerwania programowe mogą być zarejestrowane w jądrze z użyciem funkcji `open_softirq()`, która przyjmuje dwa argumenty. Pierwszym jest element typu wyliczeniowego, który związany jest z nowym przerwaniem programowym (musi być wcześniej dodany do tego typu), a drugim jest wskaźnik (nazwa funkcji) do procedury obsługi tego przerwania. Ta procedura jest wykonywana przy włączonych przerwaniach i nie może przejść w stan uśpienia, bo działa w kontekście przerwania. Jeden procesor może wykonywać w danym czasie tylko jedną procedurę obsługi przerwania programowego, ale w systemie wieloprocesorowym może być jednocześnie wykonywanych tyle tych procedur, ile jest procesorów, lub tyle samo instancji jednej z tych procedur. Ponadto procedury obsługi przerwań programowych zazwyczaj korzystają tylko z danych lokalnych dla danego CPU. Oznacza to, że przerwania programowe są bardzo skalowalne.

Tasklety

Tasklety są dolnymi połówkami wykonywanymi w kontekście przerwania i dostępnymi z poziomu modułu jądra. Przypominają one przerwania programowe, ale nie skalują się tak dobrze jako one. W systemie wieloprocesorowymi tylko jedna instancja danego taskletu może być wykonywana w tym samym czasie, ale tyle różnych taskletów, ile jest procesorów może być wykonywanych równolegle. Niemniej jednak tasklety mogą być użyte do prac, które wykonywane są z dużą częstotliwością. Każdy tasklet reprezentowany jest strukturą typu `struct tasklet_struct`, posiadającą pięć składowych. Pierwsze pole jest wskaźnikiem na strukturę tego samego typu (są one połączone w listę). Drugie przechowuje informację o stanie taskletu, czyli jedną z następujących wartości: 0 — tasklet jest nieaktywny, `TASKLET_STATE_RUN` — tasklet jest już w trakcie wykonania na innym procesorze (wartość używana w komputerach wieloprocesorowych), `TASKLET_STATE_SCHED` — tasklet został zaszeregowany do wykonania.

Tasklety

Trzecie pole jest licznikiem odwołań. Jeśli jego wartość jest większa niż zero, to tasklet jest zablokowany (ang. *disabled*), a jeśli równa zero to odblokowany (ang. *enabled*). Czwarte pole jest wskaźnikiem na *procedurę taskletu* (ang. *tasklet handler*). Jest to funkcja o następującym prototypie:

```
void tasklet_handler(unsigned long);
```

Nazwa właściwej funkcji może być inna. Pobiera ona tylko jeden argument — daną dla taskletu. Piąte pole struktury taskletu jest typu `unsigned long` i zawiera rzeczoną daną. Istnieją dwa typy tasklетów: o wysokim priorytecie i zwykłe. Te pierwsze są połączone w listę obsługiwaną przez procedurę obsługi wysoko-priorytetowego przerwania programowego. Zwykłe tasklety również są zorganizowane w listę, która jest obsługiwana przez procedurę obsługi przerwania programowego o priorytecie wynoszącym 6 (licząc od zera). Obie procedury uruchamiają zgromadzone w listach tasklety.

Tasklety

Lista zwykłych taskletów nazywa się `tasklet_vec`, a tych o wysokim priorytecie `tasklet_hi_vec`. Oba typy taskletów są zarządzane przez te same funkcje jądra, z jednym wyjątkiem. Tasklety o wysokim priorytecie są szeregowane (dodawane do listy) przez funkcję `tasklet_hi_schedule()`, a zwykłe za pomocą `tasklet_schedule()`. Jeśli tasklet już jest dodany do listy, to nie może być zaszeregowany ponownie, aż się nie wykona. Oba typy taskletów mogą być zadeklarowane z użyciem makra `DECLARE_TASKLET`. Jeśli tasklet od razu po stworzeniu musi być zablokowany, to można użyć makra `DECLARE_TASKLET_DISABLED`. Struktura taskletu może być zainicjowana przy pomocy funkcji `tasklet_init()`. Do blokowania zaszeregowanego taskletu może być użyta funkcja `tasklet_disable()`. Jeśli blokowany tasklet jest już w trakcie wykonania, to ta funkcja czeka na jako zakończenie i dopiero wtedy kończy swoje działanie.

Tasklety

Mniej bezpieczna wersja funkcji `tasklet_disable()` nazywa się `tasklet_disable_nosync()`. Nie czeka ona na zakończenie taskletu, jeśli jest on w trakcie wykonania. Zaszeregowany tasklet może być usunięty z listy przy pomocy funkcji `tasklet_kill()`. Nie może być ona użyta w kontekście przerwania ponieważ może oczekiwać na zakończenie wykonania usuwanego taskletu. Aby odblokować zablokowany tasklet należy użyć funkcji `tasklet_enable()`. Tasklety mogą być zaszeregowane przez procedurę obsługi przerwania lub inny kod jądra. Podobnie jak w przypadku przerwań programowych nie da się przewidzieć kiedy dokładnie zostaną wykonane. Szczegóły API taskletów są opisane w szóstej instrukcji laboratoryjnej.

Wątek jądra `ksoftirqd`

Przerwania programowe i tasklety, które są powtarzane z dużą częstotliwością lub same się reaktywują stanową dla systemu operacyjnego problem. Mogą generować zbyt duże obciążenie dla procesorów. Aby złagodzić ten problem są one oddawane pod kontrolę wątkowi jądra `ksoftirqd`. Każdy procesor wykonuje jedną instancję takiego wątku, który wykonuje się z najniższym możliwym priorytetem. Kiedy jest on wybudzany, to sprawdza, czy są jakieś przerwania programowe (i tym samym również tasklety) oczekujące na wykonanie. Jeśli tak, to wywołuje funkcję `__do_softirq()`. Po jej zakończeniu wątek `ksoftirqd` zmienia swój stan na `TASK_INTERRUPTIBLE` i usypia.

Kolejki prac

Kolejki prac[1] są implementacją dolnych połówek działającą w kontekście procesu. Każda praca, która ma być wykonana w ramach kolejki prac jest reprezentowana przez pojedynczy element tej kolejki (w zasadzie listy), który zawiera wskaźnik na funkcję wykonującą wspomnianą pracę. Kolejka jest przeglądana przez specjalny wątek jądra, nazywany wątkiem roboczym (ang. *worker thread*), realizujący funkcję `worker_thread()` i wywołujący poszczególne funkcje wskazywane przez elementy kolejki prac. Wątki robocze są zgrupowane w pule wątków (ang. *thread-pools*).
W komputerach wieloprocesorowych każdy procesor ma dwie takie pule, jedną dla kolejek wysoko-priorytetowych i jedną dla zwykłych. Istnieją również pule dla kolejek prac, które nie są na stałe powiązane z żadnym procesorem (ang. *unbound workqueues*). Liczba wątków w puli jest regulowana dynamicznie przez jądro.

[1] Kolejki prac (ang. *work queues* lub *workqueues*) zastąpiły inny mechanizm dolnych połówek, który był stosowany w wersjach jądra poprzedzających serię 2.6 i nazywał się *kolejkami zadań* (ang. *task queues*).

Kolejki prac

Kolejka prac jest reprezentowana w jądrze przez strukturę typu `struct workqueue_struct` (we wcześniejszych wersjach jądra ta struktura reprezentowała wątek jądra). Linux ma domyślną kolejkę prac obsługiwaną przez wątek roboczy o nazwie `kworker`. Nowe kolejki prac mogą są tworzone przez funkcję `alloc_workqueue()`, która pobiera trzy argumenty. Pierwszym jest ciąg znaków określający nazwę kolejki, a także nazwę tzw. *ratunkowego wątku roboczego*. Te wątki jądra obsługują kolejki prac zaangażowane w odzyskiwanie pamięci (ang. *memory reclaim*). Kolejny argument jest zbiorem następujących flag:

`WQ_NON_REENTRANT` domyślnie, w wieloprocesorowym komputerze, wiele instancji funkcji realizującej pracę może być wykonanych równolegle; jeśli ta flaga jest użyta, to tylko jedna instancja tej funkcji może być wykonywana w całym systemie,

Kolejki prac

`WQ_UNBOUND` kolejka nie będzie powiązana z konkretnym CPU,

`WQ_FREEZABLE` kolejka będzie brała udział w hibernacji systemu,

`WQ_MEM_RECLAIM` kolejka będzie brała udział w odzyskiwaniu pamięci,

`WQ_HIGHPRI` kolejka będzie obsługiwała prace wysoko-priorytetowe,

`WQ_CPU_INTENSIVE` w tej kolejce prace obciążające CPU nie będą opóźniały wykonania innych prac w tej samej puli wątków. Ta flaga nie ma wpływu na działanie niepowiązanych kolejek prac (ang. *unbound workqueues*).

Ostatnim argumentem `alloc_workqueue()` jest liczba określająca ile maksymalnie prac będzie mogło być wykonanych współbieżnie.

Kolejki prac

Są również dostępne dwa makra, które tworzą nową kolejkę prac. Pierwsze nazywa się `create_workqueue` i tworzy kolejkę obsługiwaną przez tyle wątków roboczych ile jest w komputerze procesorów. Drugie ma nazwę `create_singlethread_workqueue` i tworzy kolejkę prac, która jest obsługiwana tylko przez jeden wątek. Obecnie używana implementacja kolejki prac nie tworzy ustalonej liczby wątków roboczych dla każdej z kolejek. Jądro monitoruje obciążenie procesorów oraz liczbę prac w kolejce i dynamicznie dodaje wątki robocze, jeśli są potrzebne, lub je usuwa, jeśli są zbędne. Każda kolejka prac, za wyjątkiem domyślnej, może być usunięta z użyciem funkcji `destroy_workqueue()`.

Kolejki prac

Praca jest reprezentowana przez strukturę typu `struct work_struct` lub `struct delayed_work`. Pierwszy typ jest przeznaczony dla prac, które są odroczone na nieokreślony czas, a drugi dla prac z określonym czasem, po którym powinno się rozpocząć ich wykonanie. Kolejki prac gwarantują jedynie, że opóźnione prace nie rozpoczną się przed upływem tego czasu, ale nie że zostaną wykonane natychmiast po jego upływie. Każda struktura wymienionych typów ma wskaźnik na funkcję nazywaną procedurą obsługi pracy (ang. *work handler*), o następującym prototypie:

```
void work_handler(struct work_struct *work);
```

Właściwa funkcja, czyli procedura pracy, nie musi mieć takiej nazwy, jak w prototypie. W jej kodzie można wykonywać operacje, które powodują przejście wątku roboczego w stan oczekiwania, ale nie mogą one korzystać z pamięci w przestrzeni użytkownika. Struktury dla prac mogą być utworzone z użyciem makra `DECLARE_WORK` lub `DECLARE_DELAYED_WORK`. Pierwsze tworzy pracę typu `struct work_struct`, a drugie typu `struct delayed_work`.

Kolejki prac

Struktury pierwszego typu mogą być zainicjowane z użyciem makra `INIT_WORK`, a drugiego typu przez makro `INIT_DELAYED_WORK`. Zainicjowaną pracę do domyślnej kolejki można dodać przy użyciu funkcji `schedule_work()`. Jeśli praca musi zostać wykonana na określonym CPU, to należy użyć funkcji `schedule_work_on()`. Praca reprezentowana przez strukturę typu `struct delayed_work` jest dodawana do domyślnej kolejki przez funkcję `schedule_delayed_work()`. Jeśli ta praca dodatkowo musi być wykonana na określonym procesorze, to należy ją dodać funkcją the `schedule_delayed_work_on()`. Funkcja `flush_scheduled_work()` wymusza wykonanie wszystkich prac z domyślnej kolejki.

Kolejki prac

API kolejek prac ma funkcje, które mogą być zastosowane do dowolnej kolejki prac. Funkcje `queue_work()` i `queue_delayed_work()` dodają do określonej kolejki prace reprezentowane odpowiednio przez struktury typu `struct work_struct` lub `struct delayed_work`. Jeśli praca musi być wykonana na określonym CPU, to zamiast nich należy użyć funkcji `queue_work_on()` i `queue_delayed_work_on()`. Jeśli praca opóźniona o ustalony okres jest już dodana do kolejki, to ten czas można zmienić przy pomocy jednej z dwóch funkcji: `mod_delayed_work()` lub `mod_delayed_work_on()`. Z kolei funkcja `cancel_work_sync()` usuwa z kolejki pracę. Jeśli procedura obsługi pracy jest już w trakcie wykonania, to ta funkcja zaczeka na jej zakończenie. Do usuwania opóźnionych o ustalony czas prac służy funkcja `cancel_delayed_work_sync()`. Do usuwania prac opóźnionych na określony czas służy także funkcja `cancel_delayed_work()`, ale wykonuje ona tę operację w mniej bezpieczny sposób — jeśli procedura obsługi takiej pracy jest już wykonywana, to nie czeka na jej zakończenie.

Kolejki prac

Funkcja `flush_work()` oczekuje, aż określona praca zostanie wykonana lub natychmiast kończy swoje działanie, jeśli ta praca nie jest dodana do kolejki. Funkcja `flush_delayed_work()` wykonuje to samo dla określonej pracy odroczonej na ustalony czas. W końcu funkcja `flush_workqueue()` wymusza wykonanie wszystkich prac dodanych do określonej kolejki i oczekuje na zakończenie wykonania procedury obsługi pracy dla ostatniej z nich.
Więcej szczegółów na temat API kolejek prac zostało podanych w szóstej instrukcji laboratoryjnej.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!