Rozdział 1

Obsługa kolejek procesów oczekujących

Algorytmy `add_wait_queue`, `remove_wait_queue`.
Autor: Piotr Karpiuk

1.1 Wprowadzenie

Często zdarza się, że wykonujący się proces chcąc kontynuować swoje działanie musi mieć dostęp do czegoś więcej niż tylko procesor - potrzebna jest pewna informacja, której być może chwilowo nie ma w systemie. Przykładowo funkcja systemowa `read` moze potrzebowac załadowania danych z twardego dysku do obszaru pamięci procesu, proces może chciec wiedziec jaki klawisz nacisnął użytkownik, moze być zainteresowany statusem zakończenia procesu potomnego, itd. W takich przypadkach sensowne jest wstrzymanie działania procesu do czasu wystąpienia spodziewanego zdarzenia, dzięki czemu w międzyczasie inne procesy będą mogły efektywnie wykorzystać czas procesora. Aby było to możliwe, system operacyjny musi dysponować określonym mechanizmem i strukturami danych, pozwalającymi na wstrzymywanie i wznawianie procesów, jak również na przechowywanie informacji o takich procesach i szybki dostęp do nich.

W Linuksie strukturą danych odpowiedzialną za przechowywanie informacji o procesach wstrzymanych w oczekiwaniu na zdarzenie jest zbiór kolejek, a dokładniej list cyklicznych, z których każda odpowiada za zdarzenie innego typu. Elementami każdej z list są wskaźniki do struktur `task_struct` wstrzymanych procesów. W dużym uproszczeniu mechanizm oczekiwania sprowadza się do tego, że proces który chce czekać na zajście określonego zdarzenia wstawia się sam do kolejki odpowiedniego typu i dobrowolnie oddaje sterowanie. Gdy zajdzie odpowiednie zdarzenie, wszystkie procesy kolejki procesów oczekujących na to właśnie zdarzenie są budzone i mogą kontynuować działanie.

1.2 Struktury i mechanizmy

Typ elementu kolejki procesów oczekujących zdefiniowany jest następująco:

```
struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
}
```

... a jego prostota odbiera sposobność głębszej dyskusji.

Kolejki do oczekiwania są często modyfikowane przez procedury obsługi przerwań i stanowią strategiczną część systemu. Z uwagi na wieloprocesorowość, bezpośredni dostęp do nich jest zabroniony, a odbywać się może wyłącznie za pośrednictwem interfejsu składającego się z dwóch funkcji:

```
void add_wait_queue(struct wait_queue **queue,
    struct wait_queue *entry);

void remove_wait_queue(struct wait_queue **queue,
    struct wait_queue *entry);
```

Gwarantują one spójność kolejek m.in. blokując przerwania i tym samym zapobiegając jednoczesnej modyfikacji przez procedurę obsługi przerwania. Parametr `queue` oznacza tu kolejkę do modyfikacji, `entry` element który ma być dodany lub usunięty.

Ich algorytmy są proste:

Algorytm 1 add_wait_queue

Dane wejściowe: kolejka `queue`, element `entry`
Wynik: zmodyfikowana kolejka

1: zablokuj przerwania, opuść semafor
2: **jeśli** lista jest pusta **to**
3: uczyń `entry` jedynym elementem listy
4: **w przeciwnym przypadku**
5: wstaw `entry` na początek listy `queue`
6: **koniec jeśli**
7: podnieś semafor, odblokuj przerwania

Algorytm 2 remove_wait_queue

Dane wejściowe: kolejka `queue`, element `entry`
Wynik: zmodyfikowana kolejka

1: zablokuj przerwania, opuść semafor
2: usuń element `entry` z kolejki modyfikując pole `next` jego poprzednika
3: podnieś semafor, odblokuj przerwania

Problem sekcji krytycznej został rozwiązany poprzez zastosowanie funkcji `write_lock_irqsave` oraz `write_unlock_irqrestore`[1]

[1] Osoby zainteresowane problemami synchronizacji w środowisku wieloprocesorowym Linuksa mogą zerknąc do 20-stronicowego opracowania *Zamky a Semafory* pod adresem http://www.ms.mff.cuni.cz/[tylda]mnov6416/locking.html; jest to jednak lektura dla osób zdeterminowanych, albowiem Czesi też nie gęsi i swój język mają.

1.3 Zastosowania

Opisane wyżej funkcje są wywoływane w wielu miejscach kodu źródłowego jądra Linuksa. Przyjrzyjmy się niektórym.

Proces który chce się zawiesić na konkretnej kolejce, wywołuje funkcję

```
void sleep_on(struct wait_queue **p);
```

która nadaje statusowi procesu (`current->state`) wartość `TASK_UNINTERRUPTIBLE`, wstawia siebie (`current`) do kolejki procesów oczekujących, po czym wywołuje schedulera (`schedule()`) oddając tym samym sterowanie. Gdy je z powrotem odzyska, wywołuje `remove_wait_queue` celem usunięcia siebie z kolejki.

Funkcja ta ma swoją bliźniaczą postać:

```
void interruptable_sleep_on(struct wait_queue *p);
```

Różnica w ich działaniu polega na tym że pierwsza nadaje procesowi status `TASK_UNINTERRUPTIBLE`, podczas gdy druga - `TASK_INTERRUPTIBLE`, dzięki czemu uśpione zadanie może być ponownie uaktywnione przez sygnały. Stan `TASK_UNINTERRUPTIBLE` oznacza że proces czeka na spełnienie warunku związanego ze sprzętem i nie będzie z tego powodu przyjmował żadnych sygnałów.

Zawieszony proces jest uaktywniany tylko wtedy, gdy jego statusowi zostanie nadana wartość `TASK_RUNNING` - zwykle czynność tą wykonuje inny proces poprzez wywołanie funkcji

```
void wake_up(struct wait_queue **p);
```

która przebiega po wszystkich elementach kolejki nadając odpowiadającym im procesom wspomniany status.

Ciekawostką jest, że Linuks wykorzystuje kolejki procesów oczekujących do implementacji semaforów służących do synchronizacji dostępu do współdzielonych struktur danych w obszarze jądra (nie należy mylić tych semaforów z semaforami udostępnianymi dla użytkowników w UNIX System V).

```
struct semaphore {
    int count;
    struct wait_queue *wait;
}
```

Semafor traktowany jest jako zajęty, jeśli `count` ma wartość mniejszą lub równą 0. Wszystkie procesy, które chcą zająć semafor, umieszczają się w kolejce do oczekiwania. Są one następnie zawiadamiane o zwolnieniu semafora przez inny proces. Istnieją dwie funkcje pomocnicze służące do zajmowania i zwalniania semaforów:

```
void down(struct semaphore *sem) {
    while (sem -> count <= 0)
        sleep_on(sem->wait);
    sem -> count --;
}
```

```
void up(struct semaphore *sem) {
    sem -> count++;
    wake_up(sem -> wait);
}
```

Literatura

[Rochkind 93] Rochkind M. J.: *Programowanie w systemie Unix dla zaawansowanych.* Warszawa, WNT 1993.

[Stevens 95] Stevens W. R.: *Programowanie zastosowań sieciowych w systemie Unix.* Warszawa, WNT 1995.