Laboratorium 6: „Tasklety i kolejki prac”
(jedne zajęcia)

dr inż. Arkadiusz Chrobot

19 kwietnia 2024

Spis treści

Wprowadzenie

Tasklety i kolejki prac są mechanizmami zaliczanymi do tzw. dolnych połówek, czyli elementów obsługi przerwań, w których programiści umieszczają czynności związane z tą obsługą, ale takie, których wykonanie można odroczyć i które są czasochłonne. Mechanizmy te można wykorzystać także w innym celu niż obsługa przerwań. Rozdział 1 tej instrukcji zawiera informacje na temat taskletów i sposobów korzystania z nich. Z kolei rozdział 2 poświęcony jest kolejkom prac. Ostatni rozdział instrukcji zawiera listę zadań do samodzielnego rozwiązania w ramach zajęć laboratoryjnych.

1. Tasklety

Tasklety są mechanizmem pozwalającym odroczyć czynności, które muszą być wykonane w kontekście przerwania, ale ze względu na ograniczenia czasowe nie mogą być zrealizowane w procedurze obsługi przerwania (górnej połówce). Nie muszą one być koniecznie związane z obsługą przerwań, ale najczęściej do tego celu są wykorzystywane. W wieloprocesorowych systemach komputerowych tasklet określonego rodzaju w danym momencie może być wykonywany tylko na jednym procesorze. Z uwagi na to, że działa on w kontekście przerwania, to nie może zawierać żadnych czynności, które mogłyby wymagać wstrzymania działania. Zaletą taskletów jest to, że można z nich korzystać w modułach jądra. Do dyspozycji programistów są dwa typy taskletów: zwykłe i o wysokim priorytecie. Te drugie są zawsze wykonywane przed pierwszymi. Wykonanie taskletu jest jednorazowe, tzn. aby go ponownie wykonać należy ponownie go zaszeregować.

1.1. Opis API

API taskletów jest umieszczone w pliku nagłówkowym `linux/interrupt.h`. Tasklety są reprezentowane w jądrze systemu przez zmienne typu `struct tasklet_struct`, które mogą być tworzone dynamicznie (w trakcie działania modułu) lub statycznie (w trakcie kompilacji modułu). Czynności, które powinny być wykonane w ramach taskletu muszą być umieszczone w funkcji, której prototyp jest następujący:

```
void func(unsigned long);
```

Do obsługi taskletów zdefiniowano między innymi następujące funkcje i makra:

`DECLARE_TASKLET(name, func, data)` - makro, które służy do statycznego tworzenia i inicjowania taskletów. Jego pierwszym argumentem jest nazwa taskletu (nazwa zmiennej typu `struct tasklet_struct`, która będzie przez to makro utworzona), drugim jest wskaźnik na funkcję, która w ramach taskletu będzie realizowana, a trzecim wartość typu `unsigned long int`, która zostanie przekazana jako dana do tej funkcji.

`DECLARE_TASKLET_DISABLED(name, func, data)` - makro, które działa podobnie do DECLARE_TASKLET i przyjmuje te same argumenty, ale tworzy tasklet, który domyślnie jest zablokowany, tzn. po zaszeregowaniu nie zostanie uruchomiony, dopóki nie zostanie odblokowany.

`void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data)` - funkcja, która inicjuje zmienną typu `sturct tasklet_struct`. Pierwszym jej argumentem jest adres tej zmiennej, drugim adres funkcji, która będzie realizowana w ramach taksletu, a trzecim wartość typu `unsigned long int`, która będzie przekazana do tej funkcji.

`void tasklet_schedule(struct tasklet_struct *t)` - funkcja `inline`, która służy do szeregowania zwykłych taskletów do wykonania. Jako argument wywołania przyjmuje adres zmiennej typu `struct tasklet_struct`, która reprezentuje tasklet.

`void tasklet_hi_schedule(struct tasklet_struct *t)` - funkcja `inline`, która służy do szeregowania taskletów o wysokim priorytecie. Przyjmuje ten sam argument wywołania, co `tasklet_schedule()`.

`void tasklet_disable(struct tasklet_struct *t)` - funkcja `inline`, która blokuje zaszeregowany do wykonania tasklet. Jako argument przyjmuje adres zmiennej `struct tasklet_struct`, reprezentującej ten tasklet. Jeśli tasklet jest w trakcie wykonania, to funkcja wstrzymuje swoje działanie do czasu zakończenia tego taskletu.

`void tasklet_disable_nosync(struct tasklet_struct *t)` - funkcja `inline`, która blokuje tasklet. Przyjmuje ten sam argument co `tasklet_disable()`, ale w przeciwieństwie do tej ostatniej kończy swoje działanie, niezależnie do tego, czy funkcja związana z taskletem była w trakcie wykonania, czy nie.

`void tasklet_enable(struct tasklet_struct *t)` - funkcja `inline`, która odblokowuje tasklet do wykonania. Jako argument wywołania przyjmuje adres zmiennej typu `struct tasklet_struct` reprezentującej tasklet.

`void tasklet_kill(struct tasklet_struct *t)` - funkcja, która usuwa zaszeregowany do wykonania tasklet z kolejki. Jako argument wywołania przyjmuje adres struktury `struct tasklet_struct` reprezentującej tasklet.

1.2. Przykład

Listing 1 zawiera kod źródłowy modułu, który tworzy i szereguje do wykonania cztery tasklety. Dwa z nich są zwykłymi taskletami, a dwa taskletami o wysokim priorytecie.

Listing 1: Przykładowy moduł z dwoma taskletami

```
#include<linux/module.h>
#include<linux/interrupt.h>

static void normal_tasklet_handler(unsigned long int data)
{
        pr_info("Hi! I'm a tasklet of a normal priority. My ID is: %lu\n",data);
}

static void privileged_tasklet_handler(unsigned long int data)
{
        pr_info("Hi! I'm a tasklet of a high priority. My ID is: %lu\n", data);
}

static DECLARE_TASKLET(normal_tasklet_1,normal_tasklet_handler,0);
static DECLARE_TASKLET(normal_tasklet_2,normal_tasklet_handler,1);
static DECLARE_TASKLET(privileged_tasklet_1,privileged_tasklet_handler,0);
static DECLARE_TASKLET(privileged_tasklet_2,privileged_tasklet_handler,1);


static int __init tasklets_init(void)
{
        tasklet_schedule(&normal_tasklet_1);
        tasklet_schedule(&normal_tasklet_2);
        tasklet_hi_schedule(&privileged_tasklet_1);
```

```
25          tasklet_hi_schedule(&privileged_tasklet_2);
26          return 0;
27  }
28
29  static void __exit tasklets_exit(void)
30  {
31          tasklet_kill(&normal_tasklet_1);
32          tasklet_kill(&normal_tasklet_2);
33          tasklet_kill(&privileged_tasklet_1);
34          tasklet_kill(&privileged_tasklet_2);
35  }
36
37  module_init(tasklets_init);
38  module_exit(tasklets_exit);
39  MODULE_LICENSE("GPL");
40  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
41  MODULE_DESCRIPTION("A module demonstrating the use of tasklets.");
42  MODULE_VERSION("1.0");
```

W wierszu nr 2 kodu źródłowego modułu włączany jest plik `linux/interrupt.h`, zawierający API taskletów. Wiersze 4-7 zawierają definicję funkcji realizowanej w ramach zwykłych taskletów, a wiersze 9-12 definicję funkcji realizowanej w ramach taskletów o wysokim priorytecie. Różnią się one tylko dwoma elementami: nazwą i komunikatem umieszczanym w buforze jądra. Poza tym kod tych funkcji jest taki sam. O priorytecie taskletu nie decydują funkcje w ramach niego realizowane, ale sposób w jaki został on zaszeregowany do wykonania. Wiersze 14-17 zwierają deklarację taskletów. Dwa pierwsze, o nazwach `normal_tasklet_1` i `normal_tasklet_2` będą realizowały funkcję `normal_tasklet_handler()`. Aby je odróżnić, do wspomnianej funkcji zostaną przekazane różne argumenty wywołania przy pomocy trzeciego argumentu makra `DECLARE_TASKLET`. Pierwszy tasklet otrzyma wartość `0`, a drugi `1`. Kolejne dwa tasklety o nazwach `privileged_tasklet_1` i `privileged_tasklet_2` będą realizowały funkcję `privileged_tasklet_handler()` i również dla odróżnienia pierwszy z nich otrzyma jako argument wywołania wartość `0`, a drugi `1`. W konstruktorze modułu najpierw szeregowane są dwa zwykłe tasklety (wiersze nr 22 i 23), a następnie dwa o wysokim priorytecie (wiersze nr 24 i 25). Proszę zwrócić uwagę po wydaniu polecenia `dmesg` w konsoli, że jako pierwsze na ekranie ukażą się komunikaty od taskletów o wysokim priorytecie, mimo, że były one zaszeregowane później do wykonania niż zwykłe tasklety. Kolejność komunikatów w tych dwóch grupach taskletów jest taka, jak kolejność ich zaszeregowania. W destruktorze wszystkie tasklety są usuwane z kolejek do wykonania, w których zostały zaszeregowane. To działanie jest konieczne na wypadek, gdyby moduł został usunięty z jądra systemu zanim związane z nim tasklety zdążyłby się wykonać. Jeśli destruktor by ich nie usuwał, to w momencie wykonania próbowałby one wykonać związane z nimi funkcje, które nie byłby już w tym czasie dostępne, a to mogłoby zdestabilizować pracę systemu operacyjnego. Wykonanie funkcji `tasklet_kill()` dla taskletu, który się już wykonał nie jest błędem i nie ma żadnych skutków ubocznych. Jeśli jednak tasklet nie został jeszcze wykonany, to ta funkcja bezpiecznie go usunie z kolejki, w której jest zaszeregowany.

2. Kolejki prac

Kolejki prac (ang. *work queue*), podobnie jak tasklety są mechanizmem pozwalającym opóźnić wykonanie działań związanych z obsługą przerwań, ale mogą również służyć do innych celów. W przeciwieństwie do taskletów, kolejki prac wykonują zlecone im operacje w kontekście procesu, a nie przerwania. Oznacza to, że ich wykonanie może zostać zawieszone w oczekiwaniu na określone zdarzenie pochodzące od innych podsystemów jądra. Kontekst dla kolejek prac jest dostarczany przez specjalizowany wątek jądra nazywany *wątkiem roboczym*. Domyślnie z pojedynczą kolejką prac jest związanych tyle wątków roboczych, ile jest procesorów w komputerze. Upraszczając, mechanizm kolejek prac można opisać jako system list i związanych z nimi wątków. W listach, pełniących rolę kolejek FIFO są umieszczane struktury zawierające wskaźniki na funkcje realizujące określone operacje. Wątki robocze usuwają po jednej strukturze z listy i wywołują związane z nią funkcje. Działanie to kończy się z chwilą opróżnienia listy. Każda praca jest wykonywana jednorazowo, tzn. po jej zaszeregowaniu realizowana jest tylko raz, aby znów ją

wykonać należy ponownie ją zaszeregować. Kolejki prac mogą wykonywać dwa rodzaje prac. Pierwszy rodzaj to prace, których wykonanie jest przełożone na bliżej niesprecyzowaną chwilę. Dla uproszczenia będziemy je nazywać *pracami odłożonymi*. Drugi rodzaj, to prace, których wykonanie przełożono o pewien konkretny odcinek czasu, np. jedną sekundę. Będziemy je określać mianem *prac opóźnionych*. Należy pamiętać, że **z mechanizmu kolejek prac mogą korzystać wyłącznie moduły, których kod jest dostępny na licencji GPL**.

2.1. Opis API

Plik nagłówkowy `linux/workqueue.h` zawiera API związane z obsługą kolejek prac. Kolejki prac są reprezentowane przez struktury typu `struct workqueue_struct`. Pojedyncze prace odłożone opisują struktury typu `struct work_struct`, a prace opóźnione struktury typu `struct delayed_work`. Operacje, które mają być wykonane w ramach pracy dowolnego typu muszą być zawarte w funkcji o następującym prototypie:

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

Nazwa funkcji zwarta w opisywanym prototypie (`work_handler`) jest przykładowa i w rzeczywistej funkcji może być inna, bardziej adekwatna do pracy, którą ta funkcja ma wykonać. Proszę zwrócić uwagę na typ parametru tej funkcji. Jest on taki sam dla prac odłożonych, jak i dla opóźnionych. Jest to możliwie, ponieważ struktura `struct delayed_work` zawiera jako jedno ze swoich pól strukturę typu `struct work_struct`. Do obsługi kolejek prac zdefiniowano między innymi następujące makra i funkcje:

- `create_workqueue(name)` - makro, które tworzy kolejkę prac tworząc również tyle wątków roboczych ją obsługujących, ile jest procesorów (rdzeni) w komputerze. Zwraca ono wskaźnik na strukturę typu `struct workqueue_struct`, a jako argument przyjmuje ciąg znaków będący nazwą kolejki. Również każdy z wątków roboczych związanych z taką kolejką będzie miał taką nazwę.
- `create_singlethread_workqueue(name)` - to makro, podobnie jak `create_workqueue` tworzy kolejkę prac, ale taką, z którą związany jest tylko jeden wątek roboczy, wykonywany na pierwszym procesorze w komputerze. Ponieważ tworzenie i obsługa wątków jądra jest kosztowna, to użycie tego makra jest preferowanym przez programistów jądra sposobem tworzenia kolejek prac.
- `void destroy_workqueue(struct workqueue_struct *wq)` - funkcja, która usuwa kolejkę prac stworzoną przy pomocy jednego z wcześniej opisanych makr. Nie zwraca ona żadnej wartości, a jako argument wywołania przyjmuje wskaźnik na usuwaną kolejkę.
- `DECLARE_WORK(n, f)` - makro, które tworzy i inicjuje strukturę typu `struct work_struct`. Jego pierwszym argumentem jest nazwa struktury (identyfikator), a drugim wskaźnik funkcji zawierającej kod, który ma być wykonany w ramach pracy odłożonej.
- `DECLARE_DELAYED_WORK(n, f)` - makro, które tworzy i inicjuje strukturę typu `struct delayed_work`. Ma takie same argumenty jak `DECLARE_WORK`.
- `INIT_WORK(_work, _func)` - makro, które inicjuje strukturę typu `struct work_struct`. Przyjmuje ono dwa argumenty. Pierwszym jest wskaźnik na inicjowaną strukturę, a drugim wskaźnik funkcji zawierającej kod, który ma być zrealizowany w ramach pracy odłożonej. Inicjowana struktura może być utworzona w sposób dynamiczny lub statyczny.
- `INIT_DELAYED_WORK(_work, _func)` - to makro, podobnie jak `INIT_WORK` inicjuje strukturę, ale tym razem typu `struct delayed_work`.
- `bool queue_work(struct workqueue_struct *wq, struct work_struct *work)` - funkcja typu `inline`, która szereguje pracę odłożoną do wskazanej kolejki prac. Przyjmuje ona dwa argumenty wywołania: wskaźnik na kolejkę prac oraz wskaźnik na strukturę typu `struct work_struct` związaną z tą pracą. Zwraca wartość `false`, jeśli praca była już wcześniej w tej kolejce zaszeregowana, lub `true` jeśli praca zostanie pomyślnie zaszeregowana.
- `bool queue_delayed_work(struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)` - funkcja `inline`, która szereguje do wskazanej kolejki pracę opóźnioną. Zwraca ona taką samą wartość jak `queue_work()` oraz przyjmuje taki sam pierwszy argument wywołania. Drugim jej argumentem jest natomiast wskaźnik na strukturę typu `delayed_work`, z którą związana jest praca opóźniona, a trzecim wartość opóźnienia wyrażona w liczbie taktów zegara systemowego.
- `bool queue_work_on(int cpu, struct workqueue_struct *wq, struct work_struct *work)` - funkcja działa jak `queue_work()`, ale jako pierwszy argumenty wywołania przyjmuje numer procesora,

na którym działa wątek roboczy, który ma wykonać szeregowaną pracę odłożoną. Przydatna w systemach wieloprocesorowych.

`bool queue_delayed_work_on(int cpu, struct workqueue_struct *wq, struct delayed_work *work, unsigned long delay)` - funkcja, która działa podobnie do `queue_delayed_work()`, ale jako pierwszy argument wywołania przyjmuje numer procesora, na którym działa wątek roboczy, który ma wykonać szeregowaną pracę opóźnioną. Przydatna w systemach wieloprocesorowych.

`static inline bool mod_delayed_work(struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)` - funkcja `inline`, która pozwala zmienić opóźnienie zaszeregowanej pracy opóźnionej. Przyjmuje ona takie same argumenty wywołania jak funkcja `queue_delayed_work()`, ale trzeci argument jest traktowany jako nowa wartość opóźnienia. Jeśli jego wartość będzie wynosiła `0`, to praca zostanie natychmiast wykonana. Funkcja zwraca wartość `false` jeśli praca nie była jeszcze zaszeregowana. Wówczas `mod_delayed_work()` zadziała jak `queue_delayed_work()` z dokładnością do wartości zwracanej. Jeśli praca była zaszeregowana i udało się zmienić jej opóźnienie, to funkcja zwraca wartość `true`.

`bool mod_delayed_work_on(int cpu, struct workqueue_struct *wq, struct delayed_work *dwork, unsigned long delay)` - funkcja, która działa jak `mod_delayed_work()`, ale jako pierwszy argument przyjmuje numer procesora, na którym działa wątek roboczy, który ma wykonać pracę.

`bool cancel_work_sync(struct work_struct *work)` - funkcja ta anuluje wykonanie zaszeregowanej pracy odłożonej. Jeśli funkcja związana z tą pracą jest w trakcie wykonania, to `cancel_work_sync()` czeka na jej zakończenie. Jak argument wywołania przyjmuje ona wskaźnik na strukturę typu `struct work_struct` związaną z pracą do anulowania. Funkcja zwraca wartość `true` jeśli praca była zaszeregowana i udał się ją anulować lub `false` w przeciwnym przypadku.

`bool cancel_delayed_work(struct delayed_work *dwork)` - funkcja ta anuluje opóźnioną pracę, ale jeśli związana z tą pracą funkcja jest w trakcie wykonania, to nie czeka na jej zakończenie. Przyjmuje jako argumenty wywołania wskaźnik na strukturę typu `struct delayed_work` reprezentującą pracę do anulowania, a zwraca wartość `true` jeśli ta praca była zaszeregowana i udało się ją anulować i `false` w przeciwnym przypadku.

`bool cancel_delayed_work_sync(struct delayed_work *dwork)` - ta funkcja jest bezpieczniejszą wersją `cancel_delayed_work()`, gdyż w przeciwieństwie do swojej odpowiedniczki, w przypadku gdy funkcja związana z pracą jest już w trakcie wykonania, to czeka aż się ona zakończy.

`bool flush_work(struct work_struct *work)` - ta funkcja oczekuje na zakończenie wykonania zaszeregowanej pracy odłożonej. Zwraca ona wartość `true`, jeśli praca była zaszeregowana i wykonała się lub `false` jeśli nie była zaszeregowana. Opisywana funkcja przyjmuje argument wywołania w postaci wskaźnika na pracę odłożoną.

`bool flush_delayed_work(struct delayed_work *dwork)` - odpowiedniczka funkcji `flush_work()` dla prac opóźnionych reprezentowanych przez strukturę typu `struct delayed_work()`, ale oprócz oczekiwania na zakończenie realizacji pracy opóźnionej nakazuje jej bezwłoczne wykonanie.

`void flush_workqueue(struct workqueue_struct *wq)` - funkcja ta opróżnia wskazaną kolejkę, tzn. wymusza wykonanie wykonanie wszystkich prac w kolejce, które zaszeregowano do czasu jej wywołania i oczekuje na ich zakończenie. Przyjmuje ona jako argument wywołania wskaźnik na wspomnianą kolejkę prac. Wykonanie tej funkcji jest kosztowne, a w przypadku złożonych prac nawet niebezpieczne. Lepiej oczekiwać na wykonanie pojedynczych prac, lub je anulować za pomocą wcześniej opisanych funkcji, które służą do tego.

2.2. Opis API domyślnej kolejki prac

Jądro systemu Linux dysponuje domyślną kolejką prac, która jest obsługiwana przez wątki robocze o nazwie `kworker`. Informacje o nich można uzyskać przy pomocy polecenia `ps aux`, które oprócz nazwy poda jeszcze identyfikator procesora, na którym dany wątek działa. Ponieważ tworzenie nowych kolejek prac jest kosztowne, to zaleca się korzystanie z domyślniej, chyba że są ku temu wyraźne przeciwwskazania. Do obsługi domyślniej kolejki prac zdefiniowano między innymi następujące makra i funkcje:

`bool schedule_work(struct work_struct *work)` - funkcja `inline`, która służy do szeregowania pracy odłożonej do domyślnej kolejki. Jako argument wywołania przyjmuje wskaźnik na strukturę typu `struct work_struct` reprezentującą tę pracę, a zwraca wartość `true` jeśli praca została pomyślnie zaszeregowana lub `false` jeśli już była zaszeregowana.

`bool schedule_work_on(int cpu, struct work_struct *work)` - funkcja `inline`, która tak jak funkcja `schedule_work()` szereguje pracę odłożoną do domyślnej kolejki, ale wskazuje dodatkowo, na którym procesorze będzie ona wykonana. Ta funkcja jest przydatna w przypadku komputerów wieloprocesorowych. Przyjmuje ona dwa argumenty wywołania. Pierwszym jest numer procesora, a drugim wskaźnik na strukturę typu `struct work_struct` reprezentującą pracę odłożoną. Funkcja zwraca takie same wartości jak `schedule_work()`.

`bool schedule_delayed_work(struct delayed_work *dwork, unsigned long delay` - funkcja `inline`, która szereguje opóźnioną pracę do domyślniej kolejki. Przyjmuje ona dwa argumenty wywołania. Pierwszym argumentem jest wskaźnik na strukturę typu `struct delayed_work`, która reprezentuje opóźnioną pracę, a drugim czas opóźnienia wyrażony w liczbie taktów zegara systemowego. Funkcja zwraca `true` jeśli zaszeregowanie pracy powiodło się lub `false` jeśli praca była już zaszeregowana.

`bool schedule_delayed_work_on(int cpu, struct delayed_work *dwork, unsigned long delay)` - funkcja ta działa jak `schedule_delayed_work()`, ale jej pierwszym argumentem wywołania jest procesor, który ma zrealizować opóźnioną pracę.

`void flush_scheduled_work(void)` - funkcja, która opróżnia domyślą kolejkę prac. Wymusza ona wykonanie wszystkich prac zaszeregowanych do tej kolejki, przed jej wywołaniem i wstrzymuje działanie do czasu, gdy to nastąpi. Jej wykonanie może być kosztowne, podobnie jak w przypadku `flush_workqueue()`, a w przypadku złożonych prac nawet niebezpieczne.

2.3. Przykład

Listing 2 zawiera kod źródłowy modułu jądra, który ilustruje sposób zarządzania i użycia kolejki prac.

Listing 2: Przykładowy moduł używający kolejki prac

```
#include<linux/module.h>
#include<linux/workqueue.h>

static struct workqueue_struct *queue;

static void normal_work_handler(struct work_struct *work)
{
        pr_info("Hi! I'm handler of normal work!\n");
}

static void delayed_work_handler(struct work_struct *work)
{
        pr_info("Hi! I'm handler of delayed work!\n");
}

static DECLARE_WORK(normal_work, normal_work_handler);
static DECLARE_DELAYED_WORK(delayed_work, delayed_work_handler);

static int __init workqueue_module_init(void)
{
        queue = create_singlethread_workqueue("works");
        if(IS_ERR(queue)) {
                pr_alert("[workqueue_module] Error creating a work queue: %ld\n",PTR_ERR(queue));
                return -ENOMEM;
        }

        if(!queue_work(queue,&normal_work))
                pr_info("The normal work was already queued!\n");
        if(!queue_delayed_work(queue,&delayed_work,10*HZ))
```

```
30                 pr_info("The delayed work was already queued!\n");
31
32         return 0;
33 }
34
35 static void __exit workqueue_module_exit(void)
36 {
37         if(!IS_ERR(queue)) {
38                 if(cancel_work_sync(&normal_work))
39                         pr_info("The normal work has not been done yet!\n");
40                 if(cancel_delayed_work_sync(&delayed_work))
41                         pr_info("The delayed work has not been done yet!\n");
42                 destroy_workqueue(queue);
43         }
44 }
45
46 module_init(workqueue_module_init);
47 module_exit(workqueue_module_exit);
48 MODULE_LICENSE("GPL");
49 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
50 MODULE_DESCRIPTION("A module demonstrating the use of work queues.");
51 MODULE_VERSION("1.0");
```

W wierszu nr 2 włączono do modułu plik nagłówkowy `linux/workqueue.h`. Wiersz nr 4 zawiera deklarację wskaźnika na kolejkę prac. W wierszach 6-9 zdefiniowano funkcję `normal_work_handler()`, która zawiera kod do wykonania w ramach odłożonej pracy. W przypadku tej funkcji jest to umieszczenie w buforze jądra odpowiedniego komunikatu. Podobnie w wierszach 11-14 zdefiniowano funkcję `delayed_work_handler()` zawierającą kod do wykonania w ramach opóźnionej pracy. Tak jak jej odpowiedniczka dla odłożonej pracy, umieszcza ona w buforze jądra odpowiedni komunikat. W wierszu nr 16 zadeklarowana i zainicjowana jest struktura typu `struct work_struct` o nazwie `normal_work`. W wyniku inicjacji zostaje ona powiązana z funkcją `normal_work_handler()`. W wierszu nr 17 zadeklarowano i zainicjowano strukturę typu `struct delayed_work` o nazwie `delayed_work`. W rezultacie inicjacji jest ona stowarzyszona z funkcją `delayed_work_handler()`.

W konstruktorze modułu tworzona jest kolejka prac o nazwie „works”, obsługiwana przez pojedynczy wątek roboczy (wiersz nr 21). Jej adres jest zapisywany we wskaźniku `queue`. Jeśli nie udałoby się stworzyć tej kolejki, to konstruktor modułu umieści w buforze jądra odpowiedni komunikat i zakończy działanie sygnalizując błąd (wiersze 22-25). W wierszu nr 27 do utworzonej kolejki prac szeregowana jest praca odłożona. Jeśli byłaby ona wcześniej zaszeregowana, to konstruktor umieści w buforze jądra informujący o tym komunikat (wiersz nr 28). Podobne działania są wykonywane w wierszach nr 29 i nr 30 kodu źródłowego modułu, ale tym razem dotyczą one pracy opóźnionej. Opóźnienie w tym przypadku wynosi 10 sekund.

W destruktorze, w wierszu nr 38 anulowana jest praca odłożona. Jeśli to anulowanie się powiedzie, to będzie to oznaczało, że praca była wcześniej zaszeregowana, ale nie została jeszcze wykonana i w wierszu nr 39 destruktor umieści w buforze jądra komunikat o tym informujący. Podobne czynności są wykonywane dla pracy opóźnionej w wierszach nr 40 i 41 kodu źródłowego modułu. W wierszu nr 42 kolejka prac jest usuwana. Moduł może wykonać tę ostatnią czynność w sposób bezpieczny, gdyż wykonanie instrukcji ze wcześniej opisywanych wierszy gwarantuje, że kolejka prac jest pusta w chwili jej usuwania.

Działanie modułu wygodnie jest śledzić przy pomocy polecenia `dmesg -w -d`. Proszę zwrócić uwagę na komunikaty pojawiające się w buforze jądra, kiedy moduł zostanie usunięty przed upływem 10 sekund od momentu jego załadowania i po upływie tego czasu.

Zdania

1. [2 punkty] Zaprezentuj działanie makra `DECLARE_TASKLET_DISABLED` oraz funkcji `tasklet_enable()`, `tasklet_disable()` i `tasklet_disable_nosync()`.

2. [4 punktów] Stwórz tasklet automatycznie powtarzalny, tzn. taki, który po wykonaniu ponownie sam się zaszereguje do ponownego wykonania.

3. [6 punktów] Korzystając z wiedzy o listach, wątkach i mechanizmach synchronizacji stwórz własną implementację mechanizmu kolejek prac, ale bez obsługi prac opóźnionych i z tylko jednym wątkiem roboczym.

4. [2 punkty] Sprawdź, czy możliwe jest korzystanie z mechanizmu kolejek prac w module, który nie jest dostępny na licencji GPL.

5. [4 punktów] Stwórz automatycznie powtarzalną pracę opóźnioną, tzn. taką, która po wykonaniu sama się zaszereguje do ponownego wykonania.

6. [6 punktów] Napisz moduł, który będzie korzystał z taskletów i tworzył plik w katalogu `/proc` zawierający statystyki na temat tego ile taskletów i jakiego priorytetu zostało w ramach modułu utworzonych, zaszeregowanych i wykonanych lub oczekuje na wykonanie. Struktury reprezentujące tasklety powinny być tworzone i usuwane automatycznie z użyciem alokatora plastrowego.

7. [2 punkty] Zmodyfikuj moduł z listingu 2 tak, aby korzystał ze standardowej kolejki prac.

8. [4 punkty] Zmodyfikuj moduł z listingu 2 tak, aby zamiast z marka `create_singlethread_workqueue` korzystał z omówionej na wykładzie funkcji `alloc_workqueue()`. Utworzona kolejka powinna być kolejką niepowiązaną z konkretnym procesorem.