Laboratorium 1: „Moduły jądra systemu Linux” (jedne zajęcia)

dr inż. Arkadiusz Chrobot

24 lutego 2020

Spis treści

Wprowadzenie **1**

1. Prosty moduł jądra **1**

2. Kompilacja modułu **3**

3. Obsługa modułów **4**

4. Parametry modułów jądra **5**

5. Funkcja `printk()` **7**

6. Typy danych **8**

7. Pomocne makra **9**

Zadania **10**

Wprowadzenie

Instrukcja zawiera informacje o sposobie tworzenia i kompilowania modułów jądra systemu Linux w wersji 4.4[1]. Opisano w niej także specyfikę tworzenia oprogramowania dla przestrzeni jądra systemu. Rozdział 1 zawiera kod źródłowy prostego modułu jądra, wraz z jego opisem. Rozdział 2 opisuje sposób kompilacji modułu jądra. Następny, 3 rozdział poświęcony jest poleceniom, które służą do obsługi modułów (ładowania, usuwania itd.). W rozdziale 4 zawarty jest opis sposobu użycia parametrów modułów jądra. Rozdział 5 zawiera więcej szczegółów na temat funkcji `printk()`. Następujący po nim rozdział 6 opisuje typy danych specyficzne dla modułów jądra Linuksa. W końcu rozdział 7 zawiera informacje na temat przydatnych makr, które zostały zdefiniowane przez programistów Linuksa. Instrukcja kończy się listą zadań do samodzielnego wykonania.

1. Prosty moduł jądra

Moduły jądra są plikami zawierającymi wykonywalny kod, który może być dynamicznie włączany do jądra. Innymi słowy są one odpowiednikami bibliotek łączonych dynamicznie, używanych przez programy przestrzeni użytkownika. Moduły używane są do zrealizowania mechanizmów rozszerzających funkcjonalność jądra. Najczęściej tymi mechanizmami są sterowniki urządzeń, ale mogą to być też filtry pakietów sieciowych, czy implementacje systemów plików. Kod prostego modułu, który generuje komunikat *Hello World* przy włączaniu i usuwaniu z jądra systemu prezentuje listing 1.

Listing 1: Prosty moduł jądra

```
1  #include<linux/module.h>
2
3  static int __init first_init(void)
4  {
5          printk(KERN_ALERT"Welcome\n");
6          return 0;
7  }
8
9  static void __exit first_exit(void)
10 {
```

[1] Instrukcje laboratoryjne tworzone były w oparciu o tę właśnie wersję jądra Linuksa, jednakże większość zawartych w nich informacji będzie prawdziwa również dla wersji wcześniejszych, od 2.6 do 4.4, a nawet późniejszych niż 4.4.

```
11          printk(KERN_ALERT"Good bye\n");
12  }
13
14  module_init(first_init);
15  module_exit(first_exit);
16  MODULE_LICENSE("GPL");
17  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
18  MODULE_DESCRIPTION("Another \"Hello World!\" kernel module :-)");
19  MODULE_VERSION("1.0");
```

Biblioteki podprogramów, które znane są programistom tworzącym aplikacje użytkowe nie mogą być stosowane na poziomie jądra systemu operacyjnego. Jądro systemu dysponuje własnymi bibliotekami, a zatem również własnym zestawem plików nagłówkowych. Moduł `module.h`, włączony w pierwszym wierszu kodu modułu, zawiera znaczniki, makra i inne elementy, które niezbędne są do utworzenia nawet najprostszego modułu jądra.

W kodzie modułu zdefiniowano dwie funkcje. Obie są funkcjami statycznymi (proszę zwrócić uwagę na słowo kluczowe `static` w ich nagłówkach), co oznacza, że ich nazwy są widoczne jedynie wewnątrz modułu. Większość funkcji w module powinna być definiowana w ten sposób, aby nazwy ich funkcji nie powodowały kolizji z nazwami innych funkcji jądra. Wyjątkiem są funkcje, które chcemy udostępnić na zewnątrz modułu, tak aby mogły z nich korzystać inne moduły. Temat ten będzie opisany w rozdziale 7 . Funkcje odpowiedzialne za inicjację i usuwanie modułu należy wskazać kompilatorowi za pomocą, odpowiednio, makra `module_init` i `module_exit` (wiersze 14 i 15). Obie, jako argumenty wywołania, przyjmują nazwy funkcji, które mogą być dowolnymi, dozwolonymi przez reguły języka C nazwami. Funkcja inicjująca musi zwracać wartość typu `int`, która oznacza kod jej zakończenia. Poprawne zakończenie sygnalizowane jest zerem, a niepoprawne liczbą ujemną. Nie posiada ona żadnych parametrów. W jej prototypie można użyć znacznika `__init`, który informuje jądro, że ta funkcja będzie używana tylko na początku działania modułu i tylko raz. Dzięki temu może ono zwolnić pamięć przeznaczoną na tę funkcję tuż po jej wykonaniu. Funkcja usuwająca jest wbrew nazwie wywoływana przed usunięciem modułu z jądra i jej zadaniem jest „posprzątanie" po module. Może to oznaczać wykonanie różnych czynności, w zależności od tego jaką rolę pełnił moduł w jądrze, np. jeśli jest to sterownik urządzenia, to jego funkcja usuwająca może wyłączać to urządzenie. Podobnie jak funkcja inicjująca, funkcja usuwająca nie posiada żadnych parametrów, ale w przeciwieństwie do tej pierwszej nie zwraca także żadnej wartości. Może ona być oznaczona znacznikiem `__exit` (wiersz 9), który informuje jądro, że funkcja ta będzie używana tylko w wypadku, kiedy moduł będzie usuwany z jądra. Ma to znaczenie w sytuacji gdy moduł jest na stałe włączany do jądra, tj. na etapie jego kompilacji, lub wówczas, gdy jądro skompilowane jest z opcją, która wyłącza usuwanie modułów. W takich przypadkach funkcja może zostać pominięta przy włączaniu modułu do jądra.

Zasadniczym działaniem obu funkcji w module z listingu 1 jest umieszczenie komunikatu w buforze jądra przy pomocy funkcji `printk()`. Ta funkcja podobna jest w użyciu do funkcji `printf()`, jednak aby obejrzeć efekt jej działania należy użyć polecenia `dmesg` lub obejrzeć zawartość odpowiedniego pliku znajdującego się w katalogu `/var/log`. W zależności od dystrybucji Linuksa ten plik będzie nazywał się `kern.log` lub `messages`. W obu przypadkach jest to plik tekstowy. Niektóre dystrybucje Linuksa dodatkowo domyślnie wypisują zawartość bufora jądra na konsolę (ekran monitora w większości przypadków). W innych można takie zachowanie włączyć za pomocą wspomnianego wyżej polecenia `dmesg`. Warto również zwrócić uwagę, że łańcuch znaków przekazany do wywołania funkcji `printk()` poprzedzony jest stałą `KERN_ALERT`. Decyduje ona o tym jaki priorytet będzie miała wiadomość umieszczana w buforze jądra. Opis funkcji `printk()` oraz związanych z nią poleceń znajduje się w dalszej części instrukcji 5.

Wiersze 16-19 kodu modułu zawierają wywołania makr, które określają licencję na jakiej moduł jest dostępny, a także zawierają podstawowy opis modułu. Jeśli makro `MODULE_LICENSE` nie zostanie użyta, lub zostanie wywołana z ciągiem znaków oznaczającym licencję nie należącą do licencji wolnych, to jądro systemu po włączeniu moduł wygeneruje komunikat o tym, że zostało „skażone" modułem własnościowym. Taka informacja będzie zawarta w każdym komunikacie typu *kernel oops*, aby poinformować osoby badające przyczyny awarii jądra, że może ona mieć swoje źródła w kodzie, do którego dostęp jest jedynie w formie binarnej[2]. Pozostałe makra mogą zostać pominięte bez większych konsekwencji, jednakże ich

[2]Taka sytuacja oznacza najczęściej, że programiści jądra Linuksa nie podejmą się ustalenia przyczyny awarii.

użycie ułatwia określenie administratorom systemu do czego dany moduł jest przydatny i jak go użyć.

2. Kompilacja modułu

Jądro posiada własny system kompilacji, o nazwie *kbuild*, który wykorzystywany jest także do budowania modułów. Więcej na jego temat można przeczytać w dokumentacji jądra umieszczonej w katalogu `/usr/src/linux/Documentation/kbuild/`. Kompilacja modułu jądra wymaga zainstalowania plików źródłowych wersji jądra używanej w systemie oraz użycia narzędzia `make`. Zachowanie polecenia `make` można najprościej określić tworząc odpowiedni plik konfiguracyjny. Zawartość takiego pliku pokazuje listing 2. Jego treść wydaje się być skomplikowana, ale można ją potraktować jak szablon, który można wielokrotnie wykorzystać, zmieniając tylko i wyłącznie jego pierwszy wiersz.

Listing 2: Plik Makefile do kompilacji przykładowego modułu jądra

```
1       obj-m := first.o
2       KERNELDIR = /lib/modules/$(shell uname -r)/build
3
4       default:
5               $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
6
7       clean:
8               $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

Plik konfiguracyjny pokazany na listingu 2 zakłada, że kod modułu z listingu 1 zapisany jest w pliku `first.c`. Aby go skompilować wystarczy w pierwszej linii pliku `Makefile` umieścić jego nazwę z rozszerzeniem `.o`. Zapis `obj-m` oznacza, że plik wynikowy będzie modułem, a nie zostanie włączony do skompilowanej postaci jądra systemu.

Zmienna `KERNELDIR`, zawiera ścieżkę dostępu do katalogu gdzie zawarte są nagłówki jądra. Zapis `$(shell uname -r)` oznacza, że zostanie wywołane polecenie powłoki `uname` z opcją `-r` celem ustalenia wersji jądra. Łańcuch znaków odpowiadający tej wersji jest również nazwą katalogu, który zawierającego istniejące moduły jądra, wraz z dowiązaniem do katalogu nagłówków (`build`).

Po wydaniu polecenia `make` domyślnie wykonywana jest reguła `default`. Zmienna `MAKE` zawiera nazwę programu `make`, zmienna `PWD` zawiera katalog bieżący, w którym powinien znajdować się kod źródłowy kompilowanego modułu.

Reguła `clean` pozwala na usunięcie plików powstałych podczas kompilowania modułu, włącznie z plikiem zawierającym skompilowany moduł. Aby ją uruchomić należy polecenie `make` wywołać następująco: `make clean`

Jeśli chcielibyśmy skompilować moduł o innej nazwie, to wystarczy zmienić wspomnianą nazwę w pierwszym wierszu. Jeśli chcemy skompilować moduł składający się z kilku plików źródłowych, to musimy dodać kilka dodatkowych wierszy. Przykładowo, jeśli kod modułu jest zawarty w plikach o nazwach `first.c` i `second.c`, a chcemy go nazwać `third`, to na początku pliku `Makefile` należy umieścić następujący zapis:

```
obj-m := third.o
third-objs := first.o second.o
```

Listingi 3 i 4 zawierają przykład kodu źródłowego modułu podzielonego na dwa pliki, a listing 5 treść pliku `Makefile`, który pozwala go skompilować i zapisać wynik tej kompilacji w pliku `hello.ko`.

Listing 3: Plik zawierający pierwszą część kodu źródłowego przykładowego modułu jądra

```
1   #include<linux/module.h>
2
3   static int __init first_init(void)
4   {
5           printk(KERN_ALERT"Welcome!");
6           return 0;
```

```
}

module_init(first_init);
```

Listing 4: Plik zawierający drugą część kodu źródłowego przykładowego modułu jądra

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

static void __exit first_exit(void)
{
        printk(KERN_ALERT"Good bye!");
}

module_exit(first_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_DESCRIPTION("\"Hello world\" module in two parts.");
MODULE_VERSION("1.0");
```

Listing 5: Plik Makefile z instrukcjami kompilującymi kod źródłowy modułu podzielonego na dwa pliki

```
obj-m := hello.o
hello-objs := first.o second.o

KERNELDIR = /lib/modules/$(shell uname -r)/build

default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

W przypadku gdy za pomocą jednego pliku konfiguracyjnego chcemy skompilować wiele modułów, to pierwszy wiersz należy zamienić na zapis podobny do tego:

```
obj-m += mod1.o
obj-m += mod2.o
obj-m += mod3.o
```

W powyższym przykładzie zostaną skompilowane trzy modułu, których kto źródłowy jest zawarty w plikach o nazwach `mod1.c`, `mod2.c` i `mod3.c`. Jeżeli w trakcie kompilacji nie zostaną znalezione błędy w kodzie źródłowym modułu, to w jej wyniku otrzymamy plik o rozszerzeniu *.ko*, który będzie zawierał kod wynikowy modułu.

3. Obsługa modułów

Skompilowany moduł może zostać włączony (załadowany) do jądra systemu przez administratora systemu (użytkownika root) lub użytkownika posiadającego odpowiednie uprawnienia. Powłoka systemowa dostarcza kilku poleceń, które umożliwiają wykonanie tej pracy oraz innych czynności związanych z zarządzaniem modułami jądra.

Najprostszym poleceniem służącym do załadowania modułu od jądra jest `insmod`. Jedynym wymaganym argumentem wywołania tego polecenia jest nazwa, wraz z rozszerzeniem, pliku zawierającego skompilowany moduł. Przykładowo, dla opisanego w rozdziale 1 moduł wywołanie `insmod` może mieć następującą postać:

```
insmod first.ko
```

Opcjonalnie, jeśli moduł przyjmuje argumenty wywołania, to ich wartości mogą zostać wymienione po nazwie modułu w poleceniu `insmod`.

Innym poleceniem umożliwiającym załadowanie modułu do jądra jest `modprobe`. W porównaniu do `insmod`, `modprobe` ładuje dodatkowo wszystkie inne moduły powiązane z tym, który został wymieniony w linii poleceń. Załóżmy, że moduł `first.ko` jest w katalogu bieżącym. Składnia ładowania tego modułu za pomocą polecenia `modprobe` jest następująca:

```
modprobe ./first.ko
```

Polecenie `modprobe` może służyć także do wykonywania innych czynności związanych z obsługą modułów. Szczegóły działania tego polecenia można znaleźć w podręczniku systemowym (`man modprobe`).

Listę modułów załadowanych do jądra systemu wyświetla polecenie `lsmod`. Dodatkowo podaje ono ilość pamięci zajmowanej przez każdy moduł oraz ile innych modułów korzysta z funkcjonalności dostarczanej przez dany moduł. To polecenie nie przyjmuje żadnych argumentów wywołania. Jego działanie polega na sformatowaniu i wyświetleniu informacji zawartych w pliku (`/proc/modules`).

Informacje o module zapisanym w pliku z rozszerzeniem „.ko". wyświetla na ekran polecenie `modinfo`. Aby uzyskać informacje o module `first.ko` należy to polecenie wywołać następująco:

```
modinfo first.ko
```

Domyślnie na ekran wyświetlane są wszystkie te informacje, które autor modułu umieścił w nim przy pomocy odpowiednich makr. Opcje tego polecenia są opisane w podręczniku systemowym: `man modinfo`.

Usunięcie modułu z pamięci dokonywane jest za pomocą polecenia `rmmod`. Aby usunąć moduł należy w wierszu polecenia `rmmod` podać jego nazwę, np.:

```
rmmod first
```

Opcje tego polecenia opisane są w podręczniku systemowym: `man rmmod`.

4. Parametry modułów jądra

Parametry modułu, są to zmienne, którym można nadać wartość podczas ładowania do pamięci. W module, którego kod jest zamieszczony w listingu 6 są trzy takie zmienne typu `int`. Najpierw są one deklarowane jako zmienne statyczne, które mają określoną wartość początkową. Następnie każda z nich jest zmieniana w parametr za pomocą makra `module_param`, która przyjmuje trzy parametry: nazwę zmiennej, nazwę typu zmiennej oraz liczbę określającą prawa dostępu do tej zmiennej - w przypadku opisywanego modułu jest to prawo do odczytu i zapisu dla właściciela i prawo do odczytu dla pozostałych użytkowników. Wartość parametrów może być modyfikowana w kodzie modułu, tak jak to ma miejsce w destruktorze przykładowego modułu. Makrodefinicja `MODULE_PARM_DESC` służy do opisywania parametrów modułu. Przyjmuje dwa argumenty - nazwę parametru i ciąg znaków stanowiący jego opis, który można zobaczyć po skompilowaniu modułu i użyciu polecenia `modinfo` na pliku wynikowym.

Listing 6: Prosty moduł jądra z parametrami

```
1  #include<linux/module.h>
2
3  static int foo = 1;
4  static int bar = 2;
5  static int baz = 3;
6
7  module_param(foo,int,0644);
8  MODULE_PARM_DESC(foo,"parameter");
9  module_param(bar,int,0644);
10 MODULE_PARM_DESC(bar,"parameter");
11 module_param(baz,int,0644);
12 MODULE_PARM_DESC(baz,"parameter");
13
14 static int __init parammod_init(void)
15 {
16         printk(KERN_ALERT "Module parameters:\n");
17
```

```
        printk(KERN_ALERT "foo value: %d\n",foo);
        printk(KERN_ALERT "bar value: %d\n",bar);
        printk(KERN_ALERT "baz value: %d\n",baz);

        return 0;
}

static void __exit parammod_exit(void)
{
        printk(KERN_ALERT "Module parameters:\n");

        foo++; bar++; baz++;

        printk(KERN_ALERT "foo value: %d\n",foo);
        printk(KERN_ALERT "bar value: %d\n",bar);
        printk(KERN_ALERT "baz value: %d\n",baz);

}

module_init(parammod_init);
module_exit(parammod_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_DESCRIPTION("Kernel module with parametrers.");
MODULE_VERSION("1.0");
```

Moduł 6 może zostać załadowany w podobny sposób, jak poprzednio opisany:

```
insmod ./parmmod
```

Wówczas parametry `foo`, `bar` i `baz` będą miały taką wartość, jaka została im nadana w miejscu deklaracji. W trakcie usuwania modułu z pamięci zostanie ona jedynie zwiększona o jeden. Można również załadować ten moduł tak, aby nadać parametrom inną wartość. Przykładowo:

```
insmod ./parmmod foo=3 bar=4 baz=5
```

zmieni wartości początkowe wszystkich trzech parametrów, a polecenie:

```
insmod ./parmmod foo=5
```

nada wartość początkową `5` wyłącznie parametrowi `foo`.

Parametr może być także ciągiem znaków lub tablicą. Aby utworzyć parameter, przez który może być przekazany ciąg znaków należy najpierw zadeklarować tablicę elementów typu `char`, a następnie posłużyć się makrem `module_param_string`, które przyjmuje cztery argumenty. Pierwszym jest nazwa parametru, drugim nazwa wspomnianej tablicy, trzecim liczba określająca maksymalną liczbę znaków, jaką może pomieścić ta tablica, a czwartym liczba ósemkowa definiująca prawa dostępu do parametru. W przypadku parametru tablicowego należy użyć makra `module_param_array` zamiast `module_param`. Przyjmuje ono również cztery argumenty: nazwę tablicy, typ elementów tablicy, adres zmiennej typu `int`, w której makro zapisze ile zostało modułowi dostarczonych wartości dla elementów tablicy podczas jego ładowania, a ostatni argument jest liczbą ósemkową definiującą prawa dostępu do parametru.

Listing 7 zawiera kod modułu, do którego przez parametry można przekazać ciąg znaków i wartości tablicy.

Listing 7: Moduł jądra z parametrami będącymi ciągiem znaków i tablicą

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

#define ARRAY_NUMBER_OF_ELEMENTS 10
#define STRING_NUMBER_OF_ELEMENTS 40

static char foo[STRING_NUMBER_OF_ELEMENTS];
```

```
7  static int array[ARRAY_NUMBER_OF_ELEMENTS];
8  static int number_of_elements = 0;
9
10 module_param_string(foostring,foo,STRING_NUMBER_OF_ELEMENTS,0644);
11 MODULE_PARM_DESC(foo,"A char array parameter");
12 module_param_array(array,int,&number_of_elements,0644);
13 MODULE_PARM_DESC(array, "An array parameter");
14
15 static int __init parammod_init(void)
16 {
17         int i;
18         printk(KERN_ALERT"Module parameter:\n");
19
20         printk(KERN_ALERT"foo value: %s\n",foo);
21
22         printk(KERN_ALERT"array values: ");
23         for(i=0;i<number_of_elements;i++)
24                 printk(KERN_CONT"%d\n",array[i]);
25
26         return 0;
27 }
28
29 static void __exit parammod_exit(void)
30 {
31         int i;
32         printk(KERN_ALERT "Module parameter:\n");
33
34         printk(KERN_ALERT "foo value: %s\n",foo);
35
36         printk(KERN_ALERT"array values: ");
37         for(i=0;i<number_of_elements;i++)
38                 printk(KERN_CONT"%d\n",array[i]);
39 }
40
41 module_init(parammod_init);
42 module_exit(parammod_exit);
43
44 MODULE_LICENSE("GPL");
45 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
46 MODULE_DESCRIPTION("Kernel module with char array (string) and array parametrers.");
47 MODULE_VERSION("1.0");
```

Aby nadać wartości parametrom `foo` i `array` skompilowany moduł może zostać załadowany np. tak:

```
insmod ./parmod2.ko foo="Informatyka" array=1,2,3,4,5,6
```

5. Funkcja `printk()`

Funkcja `printk()` jak wspominano w rozdziale 1 jest odpowiedniczką funkcji `printf()` jednak nie wypisuje ona informacji bezpośrednio na ekran, lecz umieszcza je w buforze cyklicznym jądra, skąd w zależności od konfiguracji systemu trafiają one do odpowiednich plików. Zawartość tego bufora można wyświetlić na ekranie za pomocą polecenia `dmesg`. Przydatną opcją tego programu jest `-c`, która powoduje, że polecenie to wypisuje zawartość bufora i czyści go. Nowe komunikaty jądra będą zatem trafiać do pustego bufora i łatwiej będzie je znaleźć. Więcej opcji programu `dmesg` można poznać czytając jego stronę podręcznika: `man dmesg`.

Jak zostało to wyjaśnione w rozdziale 1 ciąg zawierający informację, która ma być umieszczona

w buforze jądra przez `printk()` jest poprzedzany stałą określającą poziom logowania, czyli priorytet tej informacji. **Miedzy tą stałą, a wspominanym ciągiem nie może występować żaden inny znak, poza spacją, a nawet ona nie jest obowiązkowa.** Aby ułatwić posługiwanie się funkcją `printk()` programiści jądra zdefiniowali szereg funkcji, o krótkich nazwach, które wywołują `printk()` z odpowiednim poziomem logowania. Tabela 1 zawiera opis stałych poziomów logowania oraz wykaz nazw wspominanych funkcji.

Tabela 1: Stałe logowania i aliasy dla `printk()` (na podstawie `http://elinux.org/Debugging_by_printing`)

Nazwa	Opis	Alias
`KERN_EMERG`	Najwyższy poziom logowania dla wyjątków, które mogą spowodować załamanie systemu lub brak stabilności.	`pr_emerg()`
`KERN_ALERT`	Poziom logowania dla zdarzeń, które wymagają natychmiastowej uwagi użytkownika.	`pr_alert()`
`KERN_CRIT`	Poziom logowania dla zdarzeń krytycznych związanych z awarią oprogramowania lub sprzętu.	`pr_crit()`
`KERN_ERR`	Poziom logowania dla błędów, najczęściej używany przez sterowniki urządzeń do raportowania problemów ze sprzętem.	`pr_err()`
`KERN_NOTICE`	Poziom logowania dla zdarzeń, np. związanych z bezpieczeństwem, które zazwyczaj nie niosą poważnych konsekwencji, ale są warte odnotowania.	`pr_notice()`
`KERN_INFO`	Poziom logowania dla zwykłych informacji.	`pr_info()`
`KERN_DEBUG`	Poziom logowania dla informacji związanych z debugowaniem.	`pr_debug()`
`KERN_DEFAULT`	Domyślny poziom logowania.	
`KERN_CONT`	Jeśli ciąg znaków w poprzednim wywołaniu `printk()` nie kończył się znakiem nowego wiersza (`\n`), to w następnym wywołaniu należy użyć tego poziomu logowania.	`pr_cont()`

W większości przykładowych modułów jądra zaprezentowanych w tej instrukcji używany jest poziom `KERN_ALERT`, aby informacje pochodzące od tych modułów były jak najszybciej umieszczane w cyklicznym buforze jądra.

6. Typy danych

Tworząc moduły jądra można posługiwać się wszystkimi typami podstawowymi zdefiniowanymi w języku C. Ponieważ jednak standard języka C nie określa precyzyjnie wielkości tych typów, a jądro systemu i co za tym idzie, jego moduły muszą działać tak samo na różnych platformach sprzętowych, to programiści Linuksa opracowali specjalne podstawowe typy danych, których wielkość jest niezależna od platformy sprzętowej i zawsze taka sama. Dostępne są one po włączeniu nagłówka `linux/types.h` do kodu modułu jądra. Ich opis zamieszczony jest w tabeli 2.

Tabela 2: Podstawowe typy danych dla jądra

Nazwa	Opis
`s8`	Liczba całkowita, ośmiobitowa.
`u8`	Liczba naturalna, ośmiobitowa.
`s16`	Liczba całkowita, szesnastobitowa.
`u16`	Liczba naturalna, szesnastobitowa.
`s32`	Liczba całkowita, trzydziestodwubitowa.
`u32`	Liczba naturalna, trzydziestodwubitowa.
`s64`	Liczba całkowita, sześćdziesięcioczterobitowa.
`u64`	Liczba naturalna, sześćdziesięcioczterobitowa.

Choć typy zmiennoprzecinkowe, takie jak `float` i `double` mogą być użyte w kodzie źródłowym jądra, to jednak operatory dla nich nie są zdefiniowane. Ich działanie wiąże się z użyciem jednostki

zmiennoprzecinkowej (FPU), której obsługa jest z punktu widzenia jądra dosyć kosztowna. Poza tym, arytmetyka zmiennoprzecinkowa w większości przypadków nie jest potrzebna w przestrzeni jądra. Gdyby zaszła jednak konieczność jej użycia, to wyrażenia zawierające zmienne i operatory zmiennoprzecinkowe należy mieścić miedzy wywołaniami funkcji `kernel_fpu_begin()` i `kernel_fpu_end()`, oraz dodatkowo dodać opcję kompilacji `-mhard-float`.

Jądro Linuksa dysponuje również odpowiednikami funkcji do obsługi ciągów znaków, które są znane z przestrzeni użytkownika. Nazywają się one tak samo i przyjmują takie same argumenty. Różnica polega na tym, że aby ich użyć należy do kodu modułu włączyć plika nagłówkowy `linux/string.h`, a nie `string.h`.

W tym samym pliku nagłówkowym co funkcja `printk()` zadeklarowano także funkcję `sprintf()`, która działa tak samo jak jej odpowiedniczka z przestrzeni użytkownika.

7. Pomocne makra

Jeśli komunikat wypisywany przez `printk()` dotyczy wyjątku czasu wykonania, to lokalizację jego przyczyny może ułatwić użycie makr `__LINE__` oraz `__FILE__`.

API modułów jądra dosyć często zmienia się z wersji na wersję. Z pomocą w pisaniu kodu modułu niezależnego od wersji przychodzą makra `LINUX_VERSION_CODE` i `KERNEL_VERSION`. Pierwsze z nich zwraca liczbę, która reprezentuje numer wersji jądra, dla której moduł został skompilowany. Nie przyjmuje ono żadnych argumentów. Drugie makro przyjmuje trzy argumenty określające numer wersji i przekształca je w pojedynczą liczbę, takiego formatu, jak ta zwracana przez `LINUX_VERSION_CODE`, czyli numer wersji jądra zapisany w szesnastkowym kodzie liczbowym. Dzięki temu te dwie liczby można porównać ze sobą. Oba makra zdefiniowane są w pliku nagłówkowym `linux/version.h`. Nietypowe ich zastosowanie przedstawia listing 8.

Listing 8: Przykład wykorzystania makr do zarządzania wersją

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

static int __init kernelinfo_init(void)
{
        if(LINUX_VERSION_CODE>KERNEL_VERSION(2,6,0))
                printk(KERN_ALERT "This is not kernel 2.6.0.\
                        It's number in hexdecimal is: %x",LINUX_VERSION_CODE);
        return 0;
}


static void __exit kernelinfo_exit(void)
{
        if(LINUX_VERSION_CODE>KERNEL_VERSION(2,6,0))
                printk(KERN_ALERT "This is not kernel 2.6.0.\
                        It's number in hexdecimal is: %x",LINUX_VERSION_CODE);
}

module_init(kernelinfo_init);
module_exit(kernelinfo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_DESCRIPTION("Kernel module that informs about kernel version");
MODULE_VERSION("1.0");
```

W module konstruktor i destruktor używają opisanych wcześniej makr do ustalenia, czy zostały uruchomione z jądrem w wersji wyższej niż *2.6.0*. Jeśli tak, to za wywołują `printk()`, która umieszcza w bufo-

rze jądra odpowiedni komunikat, zawierający wartość prawdziwego numeru wersji jądra zapisanego w kodzie heksadecymalnym. Nietypowość tego przykładu polega na tym, że zazwyczaj `LINUX_VERSION_CODE` i `KERNEL_VERSION` są używane wraz z instrukcjami preprocesora takimi jak `#ifndef` lub `#ifdef`, nie zaś bezpośrednio w kodzie jądra.

Jeśli tworzymy pliki nagłówkowe, w których mieszamy kod dla przestrzeni jądra z kodem dla przestrzeni użytkownika, to do ich rozdzielenia może być przydatne makro `__KERNEL__`. Używa się go np. w instrukcji `#ifdef` celem wskazania kompilatorowi, który kod ma być traktowany jako kod jądra.

Makra `EXPORT_SYMBOL` i `EXPORT_SYMBOL_GPL` są używane do udostępniania zmiennych i funkcji zdefiniowanych w pliku nagłówkowym lub pliku z kodem źródłowym modułu (w tym wypadku deklaracja zmiennej powinna być poprzedzona słowem kluczowym `extern`), innym modułom. Przyjmują one tylko jeden argument, którym jest nazwa zmiennej lub funkcji. Różnica w ich działaniu polega na tym, że nazwy udostępniane za pomocą `EXPORT_SYMBOL_GPL` są dostępne jedynie dla modułów na licencji GPL.

Zdania

1. `[3 punkty]` Zbadaj za pomocą operatora `sizeof` rozmiary typów danych opisanych w tabeli 2.
2. `[5 punktów]` Sprawdź działanie funkcji operujących na ciągach znaków w module jądra.
3. `[7 punktów]` Napisz moduły, które zademonstrują użycie makr `EXPORT_SYMBOL` i `EXPORT_SYMBOL_GPL`.
4. `[3 punkty]` Napisz moduł jądra, w którym przypiszesz zmiennej typu `double` wartość `0.1*0.1`. Sprawdź, czy ten moduł się kompiluje. Znajdź wytłumaczenie dla wyniku tego eksperymentu.
5. `[5 punktów]` Napisz moduł jądra, w którym zademonstrujesz działanie makr `__FILE__` i `__LINE__`.
6. `[7 punktów]` Napisz funkcję, która w zależności od tego, czy zostanie włączona do zwykłego programu, czy do modułu jądra, to wypisze stosowny komunikat, o tym w której przestrzeni została wywołana. Skorzystaj z makra `__KERNEL__`.