Wykład 4
Moduły jądra i urządzenia blokowe

Co to jest moduł jądra ?

- Jest to fragment kodu jądra, który może być ładowany i usuwany z pamięci w trakcie pracy systemu.
  - Załadowany do pamięci moduł jest integralną częścią jądra, i ma dokładnie takie same uprawnienia jak jądro, np. Może spowodować załamanie się jądra
  - Moduł ma postać pliku wynikowego .o
- Moduł jest kompilowany niezależnie od jądra.
  - Jeżeli w module odwołujemy się do jakiegoś symbolu (funkcji lub zmiennej) jądra to nie znamy danej zmiennej lub funkcji.
  - Adresy te są generowane i wstawiane w kod podczas ładowania modułu.
  - Adresy te są przechowywane w tablicy symboli jądra ./kernel/ksyms.c (patrz makrodefinicje X).
  - Moduł nie może się odwoływać do symboli niezdefiniowanych w tej tablicy.

Co może być modułem

- Każdy podsystem jądra dla którego:
  - istnieje funkcja rejestrująca dany podsystem w jądrze.
  - pożądane aby istniała funkcja usuwająca podsystem z jądra.
  - pożądane aby istniał mechanizm pozwalający na zliczanie odniesień do modułu, co pozwala na sprawdzenie, czy moduł może być usunięty.
- Cztery najważniejsze podsystemy spełniające te warunki to:
  - Urządzenia znakowe.
  - Urządzenia blokowe.
  - Urządzenia sieciowe.
  - Systemy plików.
- Dla urządzeń znakowych oraz blokowych mamy funkcje *register_*dev* (rejestracja), *unregister_*dev* (usunięcie), a zliczanie odniesień możemy zaimplementować w metodach *open* i *release*.

init_module i clean_up

- Każdy moduł musi zawierać te dwie funkcje.
- `int init_module(void)` wywoływana jest po załadowaniu modułu.
  - jest odpowiednim miejscem do rejestracji podsystemu np. Urządzenia.
  - Wynik równy 0 oznacza sukces, ujemny błąd – moduł zostanie usunięty z pamięci
- `void cleanup_module(void)` wywoływana jest przed usunięciem modułu.
  - Jest to miejsce na wyrejestrowanie urządzenia, systemu plików.

Co jeszcze musi (powinien) zawierać moduł

- Dyrektywę #include <linux/module.h>
- Zliczanie odniesień należy zaimplementować przy pomocy makrodefinicji MOD_INC_USE_COUNT oraz MOD_DEC_USE_COUNT.
  - Pierwsza z nich zwiększa a druga zmniejsza licznik odniesień do modułu.
  - Moduł dla którego licznik odniesień > 0 nie zostanie usunięty z pamięci.
- Możliwe zastosowania:
  - Umieszczenie MOD_INC_USE_COUNT w metodzie open, a MOD_DEC_USE_COUNT w metodzie release sterownika.
  - Umieszczenie MOD_INC_USE_COUNT w *init_module*, co daje gwarancję, że moduł nigdy nie będzie usunięty z pamięci.

Zalety modułów

- Twórcy dystrybucji nie muszą kompilować ogromnego jądra ze wszytkimi możliwymi rodzajami sterowników i systemów plików.
  - Wystarczy małe jądro, z niezbędnymi podstawowymi sterownikami.
  - Pozostałe mogą być ładowane w razie potrzeby.
- Istnieje mechanizm (demon *kerneld*, polecenie *modprobe*, plik *conf.modules*) pozwalający na automatyczne ładowanie i usuwanie niezbędnych modułów.
  - Próba odwołania się do urządzenia o numerze nadrzędnym *n* powoduje załadowanie odpowiedniego modułu.
  - Jeżeli urządzenie nie jest używane przez 60 sekund, następuje usunięcie modułu.
- Moduły jądra bardzo ułatwiają życie studentom Wydziału Informatyki PB na przedmiocie Systemy Operacyjne 2.
  - Dotychczas po zmianie kodu niezbędne była kompilacja i instalacja jądra.
  - Jeżeli sterownik jest modułem to można go usunąć poleceniem *rmmod*, ponownie skompilować i natychmiast załadować poleceniem *insmod*, bez konieczności restartu systemu.

Jak zaimplementować sterownik ring jako moduł

- Dodać dyrektywę #include<linux/module.h>
- Do kodu funkcji *ring_open*, najlepiej po wywołaniu down, dodać linię:

```
MOD_INC_USE_COUNT;
```

  a do kodu *ring_release* linię:

```
MOD_DEC_USE_COUNT;
```

- Dodać dwie funkcje:

```
int init_module()
{
        return ring_init();

}

void cleanup_module() {
        unregister_chrdev(RING_MAJOR,"ring");
}
```

Kompilacja, załadowanie i usunięcie modułu

- Kompilacja (oczywiście proponuję stworzyć plik Makefile)

```
gcc -D__KERNEL__ -DMODULE -O2 -c ring.c
```

  - Jądro musi być poprawnie skonfigurowane (make menuconfig)
  - Powstanie plik ring.o
  - Jeżeli ktoś nie lubi każdorazowego wpisywania opcji kompilatora, to można zdefiniować skrypt lub (lepiej) plik Makefile i użyć make.

- Załadowanie modułu:

```
insmod ring
```

- Lista wszystkich modułów, wraz z licznikami odwołań:

`lsmod` (lub obejrzeć zawartość pliku /proc/modules)

- Usuniecie modułu:

```
rmmod ring
```

Przekazywanie parametrów modułom

- Parametry modułów powinny być takie same jak nazwy jak zmienne w module.
- insmod abc=0x40 def=jola.
  - Muszą istnieć zmienne abc i def.
- Algorytm obróbki parametrów jest następujący:
  - Jeżeli pierwszy znak wartości parametru jest cyfrą parametr jest traktowany jako zmienna typu int.

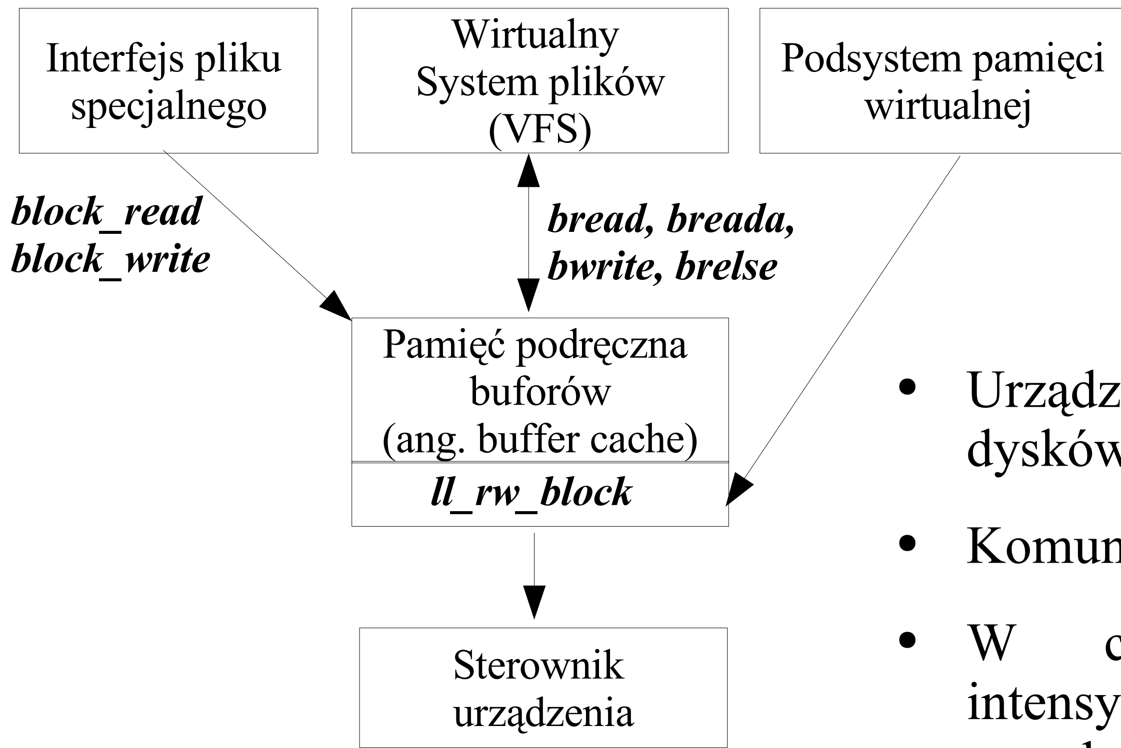
    ```
    int abc;
    ```

  - W przeciwnym wypadku jest traktowany jako łańcuch znaków

    ```
    char *def;
    ```

- Wg powyższej metody możemy zainicjalizować każdą zmienną globalną modułu, nie są też weryfikowane typy.
  - Co się stanie jeżeli omyłkowo zainicjalizujemy def jako jako int, a później na ślepo potraktujemy jako char * ?
  - Zostało to poprawione w wersji 2.2.x

Urządzenia blokowe w systemie Linux



Podsystemy jądra nie komunikują się bezpośrednio ze sterownikiem, a poprzez pamięć podręczną !!!

- Urządzenia blokowe to dyski twarde, stacje dysków, stacje CD i DVD.
- Komunikują się blokami (sektorami)
- W celu zwiększenia wydajności Linux intensywanie wykorzystują pamięć podręczną, przechowującą w pamięciu RAM najczęściej używane bloki.
  - Ponowny odczyt tego samego bloku może być wykonany z pamięci podręcznej.
  - Zmodyfikowane bloki są zapisywane z opóźnieniem, (write back) dzięki czemu kilkakrotna modyfikacja tego samego bloku nie wiąże się z kilkakrotnym zapisem.

Rejestracja urządzenia blokowego

- W strukturze file_operations jako adresy funkcji read i write należy podać standardowe funkcje *block_read* i *block_write*.
  - Funkcje te komunikują się z podsystemem pamięci podręcznej, a ten poprzez funkcję *ll_rw_block* ze sterownikiem urządzenia.
- Interfejs pomiędzy *ll_rw_block* a sterownikiem nie jest elegancki. Do jego deklaracji wykorzystuje się szereg tablic globalnych indeksowanych numerem nadrzędnym urządzenia (MAX_BLKDEV=64), plik ./include/blkdev.h
  - int *blk_size[MAX_BLKDEV] rozmiary urządzeń w blokach (drugi indeks to numer podrzędnu urządzenia)
  - int *blksize_size[MAX_BLKDEV] rozmiary bloków w bajtach (1024 jeżeli NULL)
  - int *hardesct_size[MAX_BLKDEV] rozmiary sektorów w bajtach (512 jeżeli NULL);
  - struct blk_dev_struct blk_dev[MAX_BLKDEV] adresy procedur strategii i kolejki żądań, patrz dalszy slajd

Struktura blk_dev_struct

```
struct blk_dev_struct {
        void (*request_fn)(void);
        struct request * current_request;
        struct request   plug;
        struct tq_struct plug_tq;
};
```

- Interesują nas dwa pola: *request_fn* oraz *current_request*.
- *request_fn* to adres procedury strategii. Jest ona wywoływana przez podsystem pamięci podręcznej (ll_rw_block), kiedy potrzebuje on zapisać lub odczytać bloki (obsłużyć żądania; request=żądanie)
  - Bardziej precyzjnie: Wołana gdy kolejka żadań jest pusta, a nowe właśnie zostało dodane.
  - To pole należy ustawić przy rejestracji urządzenia.
  - Procedura strategii jest asynchroniczna, powrót z procedury nie musi (ale może) nastąpić dopiero po obsłudze wszystkich zgłoszeń.
- *current_request* to wskaźnik na pierwszy element w kolejce żądań przekazanej procedurze strategii. Sama procedura strategii nie ma argumentów.

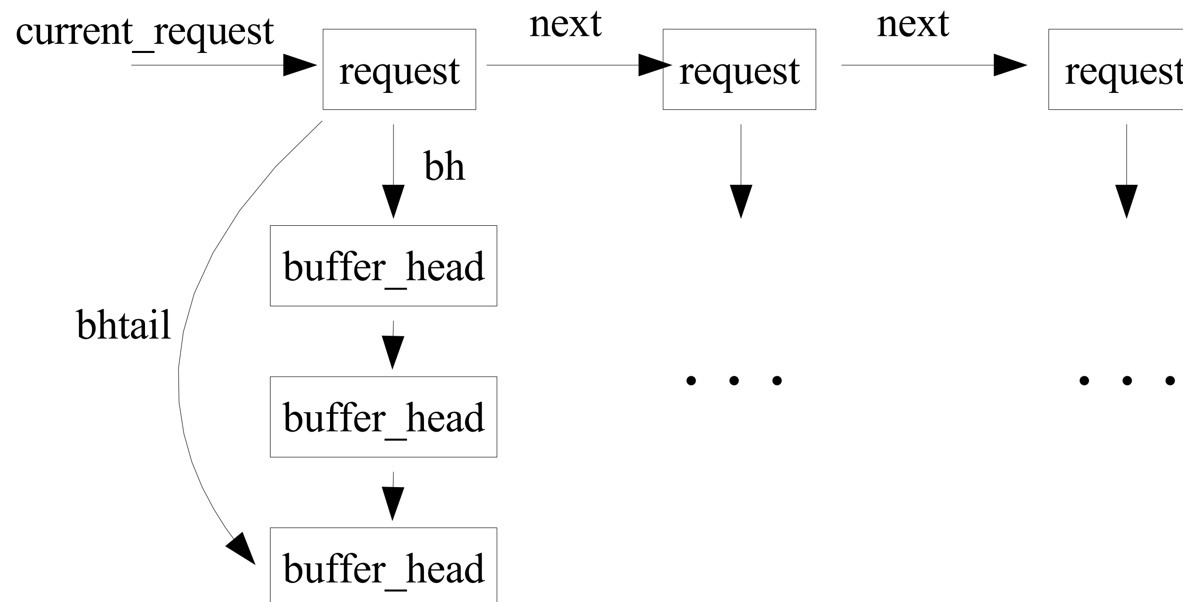
Struktura request

```
struct request {
        volatile int rq_status;
#define RQ_INACTIVE             (-1)
#define RQ_ACTIVE               1
        kdev_t rq_dev;
        int cmd;                /* READ or WRITE */
        int errors;
        unsigned long sector;
        unsigned long nr_sectors;
        unsigned long current_nr_sectors;
        char * buffer;
        struct semaphore * sem;
        struct buffer_head * bh;
        struct buffer_head * bhtail;
        struct request * next;
};
```

- Znaczenie ważniejszych pól:
  - cmd: READ żądanie odczytu, WRITE zapisu
  - sector: pierwszy sektor, nr_sectors: liczba sektorów.
  - buffer: tu należy umieścić (READ) ew. stąd odczytać dane.
  - next: wskaźnik na następne żądanie w kolejce
- W systemie są tylko 64 instancje struktury request przechowywane w tablicy
  - Gdy brakuje żądań, procesy usypiane są na kolejce *wait_for_request*.

Ale to nie wszystko

- Pojedyncze żądanie może dotyczyć wielu buforów rozproszonych po pamięci (Ale żądanie ***zawsze*** związane jest ze zwartym obszarem dysku).
  - Niektóre (bardzo zaawansowane) urządzenia wyposażone są w kontroler DMA typu *scatter-gather*, będący w stanie przesłać w jednej operacji wiele bloków rozproszonych w pamięci.
- Każdy bufor jest reprezentowany przez strukturę *buffer_head*.
  - Pole bh wskazuje na początek listy buforów związanych z żądaniem.
  - Pole bhtail wskazuje na koniec listy buforów.



Struktura buffer_head

```
struct buffer_head {
   unsigned long b_blocknr;          /* block number */
   kdev_t b_dev;                     /* device (B_FREE = free) */
   kdev_t b_rdev;                    /* Real device */
   unsigned long   b_rsector;        /* Real buffer location on disk */
   struct buffer_head * b_next;      /* Hash queue list */
   struct buffer_head * b_this_page; /* circular list of buffers in one page */
   unsigned long b_state;            /* buffer state bitmap (see above) */
   struct buffer_head * b_next_free;
   unsigned int b_count;             /* users using this block */
   unsigned long b_size;             /* block size */
   char * b_data;                    /* pointer to data block (1024 bytes) */
   unsigned int b_list;              /* List that this buffer appears */
   unsigned long b_flushtime;
   unsigned long b_lru_time;
   struct wait_queue * b_wait;
   struct buffer_head * b_prev;
   struct buffer_head * b_prev_free;
   struct buffer_head * b_reqnext;
};
```

- Najważniejsze pola to b_state, b_data, b_count, b_rsector, b_size. b_state to kombinacja logicznych flag:
  - BH_Uptodate bufor zwiera ważne dane
  - BH_Locked bufor zablokowany w pamięci

Co musi zrobić procedura strategii

- Co najmniej rozpocząć realizację pierwszego żądania.
  - Jest to asynchroniczna obsługa żądań, przy której powrót następuje natychmiast po zainicjalizowaniu żądania.
  - Dalsze przetwarzanie następuje (na ogół w procedurze obsługi przerwania) gdy żądanie lub jego fragment zostanie obsłużone.
  - Dlatego procesura strategii jest wywoływana z zablokowanymi przerwaniami
  - W międzyczasie podsystem pamięci podręcznej może modyfikować kolejkę żądań (dodanie nowych żądań, a nawet rozszerzenie istniejących).
- Procedura strategii może mieć też charakter synchroniczny. Tak jest np. W przypadku starych driverów CD-ROM.
  - Powrót następuje dopiero po wykonaniu wszystkich żądań. (Wszystkie żądania – czyli jedno w tym wypadku)
- Wymagania dla sterownika:
  - Dla każdego obsłużonego (odczytanego albo zapisanego) bufora w żądaniu należy wykasować bit BH_ Locked, ustawić bit BH_Uptodate, oraz obudzić procesy czekające na obsłużenie żądania związanego z buforem.
  - Żądania całkowicie spełnione usunąć z listy, ustalić status na RQ_INACTIVE, obudzić procesy czekające na kolejce na odczyt bloku oraz na kolejce *wait_for_request*.

Ułatwienia dla sterowników urządzeń blokowych – plik /include/linux/blk.h

- Przed włączeniem dyrektywą #include, należy przy pomocy #define zdefiniować makrodefinicję MAJOR_NR jako numer nadrzędnu urządzenia.
- Definiowane są (między innymi) dwie makrodefinicje:
  - CURRENT wskaźnik na pierwsze żądanie w kolejce do urządzenia.
  - INIT_REQUEST; inicjalizacja żądania powrót jeżeli kolejka pusta.
- Definiowana jest również (bardzo) pożyteczna funkcja *end_request(int uptodate)*. Może być ona wywoływana przy zakończeniu transmisji każdego bufora związanego z żądaniem.
  - Paremtr uptodate równy 1 oznacza poprawną transmisję bufora, a 0 błędną.
- Algorytm tej funkcji jest następujący (zakładając brak błędu).
  - Jeżeli przesłany bufor nie był ostani w żądaniu, to ustaw się na następnym buforze modyfikując pola *bh*, *buffer*, *sector*, *current_nr_sectors* (nr_sectors powinno być zmieniane przez sterownik) w strukturze request.
  - Jeżeli przesłany bufor był ostatni to usuń to żądanie z kolejki i ustaw jego stan na RQ_INACTIVE.
  - Przy okazji wykonywane są wszystkie modyfikacje stanu bufora i żądania, budzenie procesów.

Synchroniczna procedura strategii z wykorzystaniem end_request

1. INIT_REQUEST
2. Prześlij blok używając pól *buffer*, *current_nr_sectors*, *sector* w strukturze request wskazywanej przez CURRENT.
3. Zmniejsz zmienną nr_sectors o liczbę przesłanych sektorów.
4. end_request(1).
5. skocz do pkt 1.

Praca domowa

1. Przeanalizuj pliki /include/linux/blk.h (kod funkcji end_request), /include/linux/blkdev.h.
2. Przeanalizuj kod jakiegoś rzeczywistego sterownika urządzenia blokowego. (katalogi /drivers/block oraz /drivers/cdrom)

Przykład procedury synchronicznej – urządzenie simpleblk

- Jest to prosty RAM-dysk, który swoje dane przechowuje w pamięci RAM.
  - Zapisywane bloki wędrują do ciągłego bufora pamięci, odczytywane kopiowane są stamtąd.
  - Przesunięcie w tym buforze = numer_sektora*rozmiar_bloku.
  - Marnowana jest pamięć: dane w pamięci buforowej i w pamięci urządzenia. Z tego powodu Linuks implementuje RAM-dysk w sprytniejszy sposób (z właśnie z wykorzystaniem pamięci buforowej).
- `kmalloc` ma ograniczenie do 128KB. Do alokacji/dealokacji dużego bufora wykorzystujemy funkcje `vmalloc/vfree`. Nie mają one ograniczenia kmalloc, ale nie gwarantują że przydzielony obszar zajmie ciągły blok pamięci fizycznej (chodzi o stronicowanie). Wyklucza to użycia takiego bufora dla celów transmisji DMA.
- Procedura strategii jest synchroniczna => w kolejce nigdy nie przebywa więcej niż jedno żądanie.

simpleblk – procedura strategii

```
#define SECTORSIZE 512
void simple_request() {
   char *start;
   int bytes;

   // Petla nieskonczona - bo INIT_REQUEST zawiera return


   while (1) {       // Przesunięcie w buforze
      INIT_REQUEST;
      start=buffer+CURRENT->sector*SECTORSIZE;
      bytes=CURRENT->current_nr_sectors*SECTORSIZE;
      if (CURRENT->sector*SECTORSIZE+bytes > size) {
         printk(DEVICE_NAME ": buffer overrun\n");
         end_request(0);
      } else {
      if (CURRENT->cmd==WRITE) {
         memcpy(start,CURRENT->buffer,bytes);
      } else if (CURRENT->cmd==READ) {
         memcpy(CURRENT->buffer,start,bytes);
      } else  panic(DEVICE_NAME ": unknown command");
      CURRENT->nr_sectors-=CURRENT->current_nr_sectors;
      end_request(1); // Przejście do kolejnego elementu listy list
   }
}
```

simpleblk – zmienne globalne, open,release,ioctl

```
// Jeden „długi” bufor na RAM_DYSK
static char *buffer;
// rozmiar tego bufora w megabajtach
int size=1; // użytkownik ładując moduł może zmienić wartość zmiennej

// Rozmiar urządzenia w blokach
int nblocks;
// Standardowe funkcje block_read i block_write pozwalają na korzystanie
// z urządzenia jako ze zwykłego pliku
struct file_operations simple_ops = { read: block_read, write:block_write,
open:simple_open, release:simple_release};
```

- `simple_open/simple_release` – wywołanie `MOD_INC_USECOUNT` oraz `MOD_DEC_USECOUNT`. Należałoby jeszcze sprawdzać numer podrzędny i zwracać kod błędu w przypadku numeru innego niż zero.
- Niektóre systemy plików wymagają implementacji polecenia ioctl BLKGETSIZE zapisującego pod adresem będącym trzecim argumentem funkcji ioctl rozmiar urządzenia w sektorach 512 bajtowych (np. mkfs.fat tworzący system plików FAT)

Simpleblk - init_module

```
int init_module() {
   size*=(1024*1024);
   // alokacja może się nie powieść
   buffer=vmalloc(size);
   if (buffer==NULL) {
      printk(DEVICE_NAME ": Buffer allocation failed\n");
      return -1;
   }
   // rejestracja też nie
   if (register_blkdev(MAJOR_NR,DEVICE_NAME,&simple_ops)<0) {
      printk(DEVICE_NAME ": Device registration failed\n");
      vfree(buffer);
      return -1;
   }
   nblocks=size/1024;
   blk_size[MAJOR_NR]=&nblocks; // ustawiamy tablicę rozmiarów
   blk_dev[MAJOR_NR].request_fn=simple_request; // procedura strategii
   printk(DEVICE_NAME ": Initialization successful\n");
   printk(DEVICE_NAME ": %d blocks\n",nblocks);
   return 0;
}
```

simpleblk – test urządzenia

insmod simpleblk.o size=16   // RAM-dysk na 16MB

mknod /root/sblk b 50 0   // Utworzenie pliku specjalnego

mkfs.ext2 /root/sblk   // Utwórz system plików na urządzeniu

mkdir /root/mnt   // Utwórz punkt montowania

mount /root/sblk /root/mnt   // Zamontuj system plików

Teraz używamy systemu plików pod /root/mnt, i np. polecenia df

umount /root/mnt   // Odmontowanie systemu plików

// Ponowne zamontowanie, RAM-dysk przechowuje dane do momentu usunięcia modułu z pamięci

mount /root/sblk /root/mnt