Wykład 3
Sterownik urządzenia znakowego

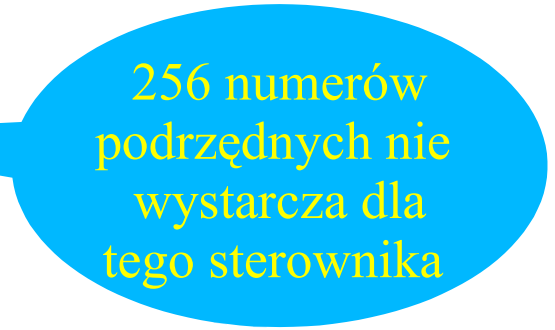
Urządzenia w Linuxie

- W systemie Linux występują trzy kategorie urządzeń:
- Urządzenia blokowe na których przechowywane są systemy plików. Przykładami są: stacja dysków elastycznych, dyski twarde (IDE lub SCSI), ale także RAM-dysk. Dane są przesyłane w blokach.
- Urządzenia znakowe, do których mogą być przesyłane pojedyncze bajty. Przykłady: terminale, port drukarki, mysz, karty dźwiękowe.
- Karty sieciowe w Linuksie stanowią trzeci typ urządzeń. Dane przesyłane są w pakietach.
- Każde urządzenie blokowe bądź znakowe ma dwa numery (0-255):
  - numer nadrzędny identyfikuje sterownik urządzenia.
  - numer podrzędny identyfikuje kolejne urządzenie obsługiwane przez sterownik.
  - Jeden sterownik obsługuje urządzenia o identycznym numerze nadrzędnym.
- Bezpośredni dostęp do urządzeń poprzez pliki specjalne umieszczone w katalogu /dev

Plik /proc/devices

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyp
  5 cua
  7 vcs
 10 misc

Block devices:
  2 fd
  3 ide0
 22 ide1
```



- System plików /proc zawiera informacje o systemie dostępne w postaci plików.
- Plik /proc/devices listuje aktywne sterowniki urządzeń, w postaci par <numer_nadrzędny nazwa_sterownika>

Fragment wyniku ls -l /dev

```
brw-rw----   1 root     floppy     2,   0 May 14  1996 fd0
brw-r-----   1 root     disk       3,   0 Apr 28  1995 hda
brw-r-----   1 root     disk       3,   1 Apr 28  1995 hda1
brw-r-----   1 root     disk       3,   2 Apr 28  1995 hda2
brw-r-----   1 root     disk       3,  64 Apr 28  1995 hdb
brw-r-----   1 root     disk       3,  65 Apr 28  1995 hdb1
crw--w----   1 root     tty        4,   1 Oct 19 13:39 tty1
crw--w----   1 root     tty        4,   2 Oct 19 13:46 tty2
crw--w----   1 root     tty        4,   3 Oct 19 13:46 tty3
```

- Jeden sterownik obsługuje wiele różnych urządzeń o różnych numerach podrzędnych
- Plik specjalny możemy założyć przy pomocy polecenia *mknod* np.

```
mknod /dev/moje c 60 0
```

Zakłada plik specjalny odpowiadający urządzeniu znakowemu o numerze nadrzędnym 60 i podrzędnym 0.

Wykorzystanie urządzeń znakowych z poziomu programu użytkownika

- Urządzenia to pliki, więc możemy na nich wykonywać większość wywołań systemowych na plikach, np.
  - int open(char *file,int flags): otwarcie, wynik jest deskryptorem pliku
  - int close(int fd): zamknięcie pliku o deskryptorze fd
  - int read(int fd, char *buf, size_t count): odczyt z urządzenia.
  - int write(int fd, char *buf, size_t count): zapis do urządzenia.
  - select: oczekiwanie na nadejście danych z urządzenia.
- Dodatkowo istnieje wywołanie systemowe `ioctl(fd,command,data)`, specyficzne dla urządzeń, pozwalające na zmianę i odczyt parametrów urządzenia.
  - Komenda określa numer parametru, oraz czy następuje odczyt zapis parametru.
  - Dane na ogół jest wskaźnikiem, często do jakiejś większej struktury
  - Np. dla karty dźwiękowej możemy zmieniać format dźwięku i prędkość próbkowania.

Implementacja wywołań systemowych

- Implementacja wywołań systemowych operujących na plikach znajduje się w podkatalogu ./fs
  - Nazwy funkcji realizujących poszczególne wywołania rozpoczynają się od sys. Np sys_open realizuje open, sys_read realizuje read.
- Szczególnie interesująca jest sys_open, ze względu na konieczność analizy ścieżek.
- Gdy otwieranym plikiem jest plik specjalny reprezentujący urządzenie, następuje przekazanie, sterowania dla procedury specyficznej dla danego urządzenia.
- Zatem sterownik powinien definiować swoje wersje operacji open, close, read, write, ioctl.
  - Nie każda operacja musi być zdefiniowana np. write dla urządzenia tylko dla odczytu.
- Adresy funkcji realizujących poszczególne operacje przechowywane są w strukturze `file_operations`.

Struktura file_operations

```
struct file_operations {
 int (*lseek) (struct inode *, struct file *, off_t, int);
 int (*read) (struct inode *, struct file *, char *, int);
 int (*write) (struct inode *, struct file *, const char *, int);
 int (*readdir) (struct inode *, struct file *, void *, filldir_t);
 int (*select) (struct inode *, struct file *, int, select_table *);
 int(*ioctl)(struct inode*,struct file*,unsigned int,unsigned long);
 int(*mmap)(struct inode *, struct file *, struct vm_area_struct *);
 int (*open) (struct inode *, struct file *);
 void (*release) (struct inode *, struct file *);
 int (*fsync) (struct inode *, struct file *);
 int (*fasync) (struct inode *, struct file *, int);
 int (*check_media_change) (kdev_t dev);
 int (*revalidate) (kdev_t dev);
};
```

- Zdefiniowana, ponownie jak dalsze struktury omawiane na wykładzie, w pliku include/linux/fs.h
- Struktura zawiera wskaźniki do funkcji realizujących poszczególne operacje. Najważniejsze to read, write, open, oraz release (realizujące zamknięcie urządzenia).

Rejestracja urządzenia w systemie

- Zadeklaruj zmienną typu `struct file_operations`.
- Ustaw pola w strukturze na adresy funkcji implementujących poszczególne operacje.
- Dla urządzeń znakowych wywołaj funkcję:

```
int register_chrdev(unsigned major, char *name, struct
  file_operations *fops);
```

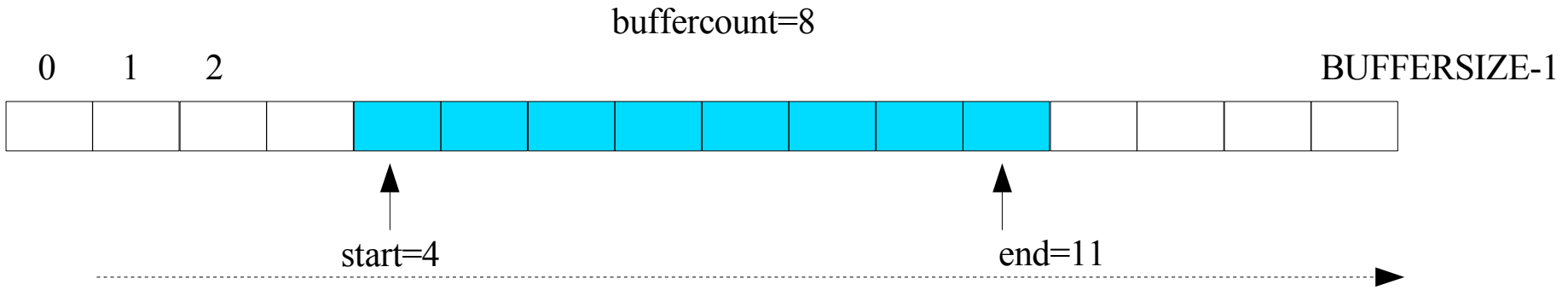
  gdzie `major` jest numerem nadrzędnym urządzenia, `name` jego nazwą (pojawi się w pliku /proc/devices), a wskaźnikiem do zadeklarowanej przez Ciebie zmiennej.

- Od tego momentu wszystkie operacje na plikach specjalnych o numerze `major` będą kierowane do sterownika.
- Istnieje też funkcja `unregister_chrdev` usuwająca sterownik (wykorzystywana gdy sterownik jest zaimplementowany jako moduł jądra).
- Urządzenia blokowe mają funkcje `register_blkdev` oraz `unregister_blkdev`.

Struktury inode oraz file

- `struct inode` reprezentuje plik, `struct file` zawiera dodatkowe informacje o otwartym pliku np. bieżącą pozycję, identyfikator procesu który otworzył plik itc.
- Przekazywane są dla większości funkcji sterownika, Dla naszych potrzeb ważne są następujące pola.
  - `kdev_t i_rdev;` w strukturze `inode`. Koduje ono numer nadrzędny i podrzędny urządzenia. Jeżeli mamy argument funkcji zadeklarowany w postaci `struct inode *inode`, to `MAJOR(inode->i_rdev)` określa numer nadrzędny, a `MINOR(inode->i_rdev)` numer podrzędny urządzenia.
  - `f_pos` w strukturze file reprezentującą bieżącą pozycję w otwartym pliku.

Przykładowe urządzenie znakowe: ring.c



- Urządzenie „akademickie”
- Jest to bufor cykliczny w postaci tablicy o rozmiarze BUFFERSIZE bajtów. Operacja write zapisuje bajty do bufora na pozycji end, read odczytuje dane z pozycji start. Zmienna buffercount przechowuje liczbę bajtów w buforze.
- W następujących sytuacjach: (a) bufor pełny przy zapisie do urządzenia (b) bufor pusty przy odczycie z urządzenia i urządzenie jest otwarte co najmniej dwa razy następuje uśpienie procesu wykonującego operację read albo write, odpowiednio na kolejce (a) read_queue (b) write_queue.
- Tak naprawdę mamy do czynienia z problemem producent-konsument.

Zmienne globalne sterownika

```
#define BUFFERSIZE 1024

static char *buffer;
int buffercount;
int start,end;
int usecount;
struct semaphore sem=MUTEX;
struct wait_queue *read_queue,*write_queue;
```

- `static char *buffer` jest wskaźnikiem na bufor (bufor jest dynamicznie alokowany).
- `int usecount;` zlicza liczbę otwartych deskryptoru plików do tego urządzenia.
- `struct semaphore sem` jest semaforem niezbędnym do realizacji wzajemnego wykluczania przy dostępnie do krytycznych.
- `struct wait_queue *read_queue,*write_queue` to kolejki procesów uśpionych odpowiednio przy operacji odczytu i zapisu.

Operacja open

```
int ring_open(struct inode *inode,struct file *file)
{
  down(&sem);
  usecount++;
  if (usecount == 1) {

    // kmalloc moze uspic proces - uwaga na synchronizacje
    buffer=kmalloc(BUFFERSIZE,GFP_KERNEL);
    buffercount=start=end=0;
  }
  up(&sem);
  return 0;
}
```

- Pamięć dla bufora przydzielona jest przy pierwszym otwarciu pliku.
- Funkcja kmalloc może uśpić bieżący proces (gdy brak pamięci w systemie, trzeba przesłać pewne ramki do obszaru wymiany)
  - Zatem możliwy jest następujący scenariusz: Proces A wykonuje open => kmalloc uspia => Proces B wykonuje open =>Proces B wykonuje read => Oops bufor nie został przydzielony !
  - W celu uniknięcia tego scenariusza wykorzystujemy semafor do zagwarantowania wzajemnego wykluczania operacji open
  - Pytenie: czy nie grozi to blokadą? Odp: nie grozi (***dlaczego ?***) .

Rejestracja sterownika

```
struct file_operations ring_ops = {
   read: ring_read, write:ring_write,
   open:ring_open, release:ring_release
};

#define RING_MAJOR 60

void ring_init(void)
{
   init_waitqueue(&write_queue);
   init_waitqueue(&read_queue);
   usecount=0;
   register_chrdev(RING_MAJOR,"ring",&ring_ops);
    printk("Ring device initialized\n");
}
```

- Sposób inicjalizacji struktury jest charakterystyczny dla kompilatora gcc (nie jest częścią standardu ANSI C).
- Plik ring.c znajduje się w podkatalogu drivers/char/
- Funkcja `ring_init` jest wołana z pliku mem.c z funkcji mem.c z funkcji `chr_dev_init`.

Operacja release

```
void ring_release(struct inode *inode,struct file *file)
{
   usecount--;
   if (usecount==0)
      kfree(buffer);
}
```

- Jeżeli licznik otwartych plików osiągnie zero, zwalniamy pamięć bufora.

Sygnały

- W systemach Uniksowych każdy proces może otrzymać jeden z 32 sygnałów. Każdy z nich ma swój numer (0-31). Sygnał może być wysłany przez inny proces, lub też może zostać spowodowany przez bieżący proces (naruszenie mechanizmów ochrony, dzielenie przez zero, pułapka przy debugowaniu, sygnał alarmu - zgłaszany cyklicznie.
- Możliwe reakcje na sygnał: (a) zakończenie procesu (b) zakończenie procesu z zapisem jego obrazu w pamięci (ang. core) na dysk (c) zignorowanie (d) wykonanie handlera (pewnej funkcji w procesie).
  - Proces może programować swoje reakcje na sygnał, z wyjątkiem sygnału SIGKILL (nr 9), który zawsze powoduje przerwanie procesu.
- Problem: proces został uśpiony w oczekiwaniu na znak z klawiatury, znaku nie ma, a proces otrzymał sygnał
  - Należy usypiać proces w stanie TASK_INTERRUPTIBLE (interruptible_sleep_on) a nie TASK_UNINTERRUPTIBLE.
  - po obudzeniu sprawdzamy wyrażenie current->signal & ~current->blocked. Jeżeli jest różne od zera, to znaczy że został zgłoszony sygnał i należy jak najszybciej powrócić z operacji read albo write.

Operacja write – z błędem

```
int ring_write(struct inode *inode,struct file *file,const char *pB,int count) {
   int i;
   for(i=0;i<count;i++) {
      while (buffercount==BUFFERSIZE) {
         interruptible_sleep_on(&write_queue);
         if (current->signal & ~current->blocked) {
            if (i==0)
               return -ERESTARTSYS;
            return i;
         }
      }
      down(&sem);
      buffer[end]=get_user(pB+i);
      buffercount++;
      end++;
      if (end==BUFFERSIZE)
         end=0;
      up(&sem);
      wake_up(&read_queue);
   }
   return count;
}
```

Usypiaj proces tak długo jak bufor jest pełny

Jeżeli proces został obudzony przez sygnał, natychmiast powróć z funkcji

Makro get_user odczytuje zmienną o długości 8, 16 lub 32 bitów z pamięci procesu – *może uśpić !!!*

Obudź procesy wykonujące operacje read które zostały uśpione bo zastały pusty bufor.

Zmienna pB wskazuje na obszar pamięci w przestrzeni procesu (nie jądra). W systemie Linux 2.0.x w kodzie jądra nie możemy tak po prostu odwołać się do pamięci procesu (w nowszych wersjach jądra możemy). Należy użyć specjalnych makr get_user/put_user.

Gdzie tkwi błąd ?

```
int ring_write(struct inode *inode,struct file *file,const char *pB,int count) {
   int i;
   for(i=0;i<count;i++) {
      while (buffercount==BUFFERSIZE) {
         interruptible_sleep_on(&write_queue);
         if (current->signal & ~current->blocked) {
            if (i==0)
               return -ERESTARTSYS;
            return i;
         }
      }
      down(&sem);
      buffer[end]=get_user(pB+i);
      buffercount++;
      end++;
      if (end==BUFFERSIZE)
         end=0;
      up(&sem);
      wake_up(&read_queue);
   }
   return count;
}
```

Czekamy, aż w buforze będzie wolny conajmniej jeden znak`

Potencjalne przełączenie kontekstu !!!

Scenariusz prowadzący do błędu: Proces A czeka => zwalnia się miejsce na jeden znak=>A wchodzi do semafora=>Przełączenie kontekstu do B=>B nie musi czekać w pętli while (***miejsce ciągle wolne !!!***)=>B wstrzymany na semaforze=>Przełączenie kontekstu do A=>A wstawia znak (***bufor pełny !!!***) i wychodzi z semafora=>przełączenie kontekstu do B=>B wchodzi do semafora i wstawia znak (***do pełnego bufora – katastrofa !!!***)

A jak go poprawić ?

- Analogiczny problem występuje w operacji read
- Problem związany jest z faktem, że oczekiwanie na zwolnienie miejsca w buforze jest również częścią sekcji krytycznej.
- Narzucające się przesunięcie operacji down, przed pętlę while doprowadzi do blokady
  - Ponieważ proces czekający na zwolnienie bufora wstrzyma operację read innych procesów.
- Rozwiązanie=> przenieść instrukcję mogącą przełączyć kontekst poza sekcję krytyczna !!!
  - get_user przed pętlę while (w operacji write)
  - put_user po modyfikacji bufora (w operacji read)
- Wykorzystujemy fakt niewywłaszczalności jądra.
- W takim przypadku semafory są niepotrzebne => można z nich zrezygnować !!!

Przykład dla operacji write

```
int ring_write(struct inode *inode,struct file *file,const char *pB,int count) {
   int i; char tmp;
   for(i=0;i<count;i++) {
      tmp=get_user(pB+i);
      while (buffercount==BUFFERSIZE) {
         interruptible_sleep_on(&write_queue);
         if (current->signal & ~current->blocked) {
            if (i==0)
               return -ERESTARTSYS;
            return i;
         }
      }
      buffer[end]=tmp;
      buffercount++;
      end++;
      if (end==BUFFERSIZE)
         end=0;
      wake_up(&read_queue);
   }
   return count;
}
```

To niczym nie grozi

Brak możliwości przełączenia kontekstu

Ciekawostka: W jądrze 2.2.x (i późniejszych) w wersji wieloprocesorowej powyższy kod jest błędny (dwa procesy mogą jednocześnie zostać obudzone na dwóch różnych procesorach.

Mam nadzieję, że Ci z Państwa, którzy skarżyli się że *„na wykładzie wykorzystuje się takie przestarzałe jądro"*, teraz zrozumieli nasz wybór.

Operacja read

```
int ring_read(struct inode *inode,struct file *file,char *pB,int count)
{
   int i; char tmp;
   for(i=0;i<count;i++) {
      while (buffercount==0) {
         if (usecount==1)
            return i;
         interruptible_sleep_on(&read_queue);
         if (current->signal & ~current->blocked) {
            if (i==0)
               return -ERESTARTSYS;
            return i;
         }
      }
      tmp=buffer[start];
      start++;
      if (start==BUFFERSIZE)
         start=0;
      buffercount--;
      wake_up(&write_queue);
      put_user(tmp,pB+i);
   }
   return count;
}
```

Usypiaj w oczekiwaniu na pojawienie się znaków w buforze, ale zakończ read jeżeli plik otwarty był tylko raz

Brak możliwości przełączenia kontekstu

read zwracające zero oznacza koniec pliku – wykorzystywane przez polecenie cat. (Spowoduje wyjście pliku, jeżeli nikt nie otworzył go do pisania).

Przetestowanie sterownika

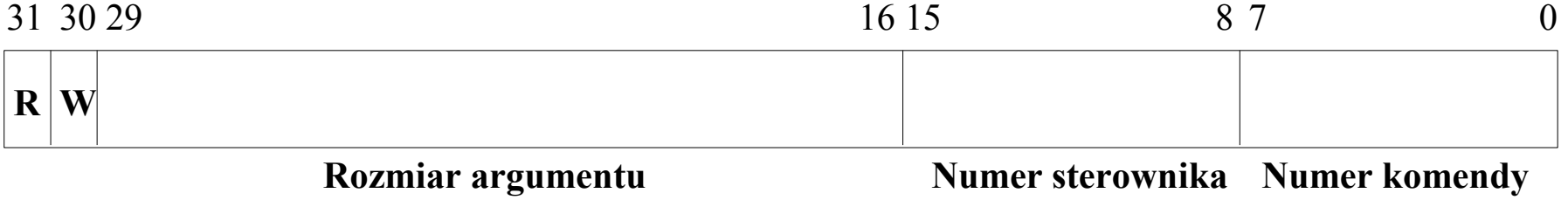
- mknod /dev/ring c 60 0 (tylko raz, jako root)
- Na jednym terminalu :
  - cat plik.txt > /dev/ring.
  - plik.txt powinien mieć co najmniej 1024 bajty, aby uśpić proces
- Na drugim terminalu:
  - cat /dev/ring

Praca domowa

- Spróbuj przeanalizować kod funkcji sys_read, sys_write, sys_close, sys_ioctl znajdujący się w podkatalogu fs.
- Jakie może być przeznaczenie funkcji jądra `verify_area` ?
  - Podczas jednego z poprzednich wykładów wspomniałem o pewnym niebezpieczeństwie.
- Jakie są konsekwencje, dla sterowników urządzeń, braku wywołania tej funkcji w przypadku sys_ioctl ?
  - Wskazówka: obejrzyj obsługę operacji ioctl w przypadku sterownika karty dźwiękowej drivers/sound/soundcard.c
- Praca nie jest obowiązkowa, ale brak jej wykonania może wyjść na jaw przy realizacji projektu, nieuchronie spowoduje obniżenie oceny.

Wywołanie systemowe ioctl

- `int ioctl(pid,command,data);`
- command powinno być kodowane w specjalny sposób, zalecana konwencja to:



- Kodowanie numeru (nadrzędnego) sterownika nie pozwala na wydania przez pomyłkę polecenie nie tego urządzeniu.
- Pola R i W oznaczają odczyt bądź zapis.
- Istnieją gotowe makra do zakodowania poleceń, w pliku /include/linux/ioctl.h
  - _IOW(c,d,t) – zapis do urządzenia c, numer polecenia d, typ danych t.
  - _IOR(c,d,t) – odczyt z urządzenia c, numer polecenia d, typ danych t.
  - _IOWR(c,d,t) – zapis i odczyt.
  - _IO(c,d) – komenda nie ma parametrów. (np. reset urządzenia)
- Przykład: _IOW(60,1,int) - polecenie o numerze 1 dla sterownika o numerze nadrzędnym 60, dana typu int.
- Jeżeli następuje odczyt, lub sizeof(dane)>4, jako dane musimy użyć wskaźnika