Wykład 9
Systemy plików w Linuksie
część druga

Obiekt otwartego pliku

- Dlaczego, oprócz obiektu i-węzła, potrzebny jest obiekt otwartego pliku ?
  - i-węzeł reprezentuje plik lub katalog.
- Plik może być otwarty przez kilka procesów, lub wielokrotnie przez ten sam proces.
  - Z każdym otwarciem pliku związany jest tryb otwarcia oraz bieżąca pozycja pliku, wykorzystywana przez funkcje read oraz write.
  - Zmienne te muszą być przechowywane indywidualnie – w nowym obiekcie.
- Z obiektem otwartego pliku związane są metody, po raz pierwszy omówione przy omawianiu sterowników urządzeń.
- Otwarte pliki mogą być dziedziczone przez procesy potomne.
  - Dlatego też obiekt pliku musi implementować zliczanie referencji.
  - Operacje na otwartym pliku są niepodzielne => problem synchronizacji

Struktura struct file

```
struct file {
   mode_t f_mode; // tryb otwarcia pliku - kodowanie Linuksa
   loff_t f_pos;  // pozycja bieżąca
   unsigned short f_flags; // tryb otwarcia pliku - kodowanie open
   unsigned short f_count; // licznik odniesień.
   // Poniższe pola są związane z odczytem z wyprzedzeniem
   unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
   struct file *f_next, *f_prev;
   int f_owner;
   struct inode * f_inode;       // i-węzeł otwartego pliku
   struct file_operations * f_op; // tablica metod
   unsigned long f_version; // używane do synchronizacji
   void *private_data;
};
```

- private_data – do wykorzystania przez system plików lub sterownik urządzenia.
- f_mode, f_flags – uprawnienia (odczyt, zapis) z jakimi plik był otwarty zakodowane na dwa różne sposoby

Zarządzanie strukturami struct file

- Funkcja open zwraca deskryptor otwartego pliku (liczba całkowita). Jest on następnie wykorzystywany przez wszelkie funkcje operujące na tym pliku.
- W strukturze task_struct procesu zawarte jest pole

```
struct files_struct * files;
```

a struktura files pełni rolę tablicy deskryptorów plików:

```
struct files_struct {
        int count; // liczba procesów (wątków) współdzielących tablicę
        fd_set close_on_exec;
        fd_set open_fds; // maska bitowa wykorzystanych deskryptorów
        struct file * fd[NR_OPEN]; // 256 pozycji
};
```

- Deskryptor jest indeksem do tablicy fd.
- Maski bitowe pozwalają na znaczne przyspieszenie operacji (Na ogół wskaźnik zajmuje 32 bity
- Nowy proces (fork) dziedziczy otwarte deskryptory z pliku potomnego i współdzieli je z nim. Nowe pliki otwierane przez proces rodzicielski lub potomny nie są współdzielone.
- Nowy wątek (clone) współdzieli całą strukturę files_struct z wątkiem rodzicielskim.

Systemowa tablica struktur struct file - fs/file_table.c

- Struktury struct_file przechowywane są na dwukierunkowej liście cyklicznej z dowiązaniami zaimplementowanej przy pomocy pól f_next oraz f_prev.
- Pierwszy element tej listy to first_file, a jej rozmiar jest ograniczony przez NR_FILE (1024). Rozmiar listy przechowuje zmienna nr_files.
- Alokację nowej struktury przeprowadza funkcja get_empty_filp.
- Zwolnienie struktury to ustawienie pola count na zero.
- Zwolnienie deskryptora (numeru) to:
  - ustawienie wskaźnika w tablicy [fd] na NULL
  - oraz wyzerowanie maski bitowej

Metody obiektu otwartego pliku

```
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);
};
```

- Były omawiane przy okazji sterowników urządzeń znakowych. Dla systemu plików ważne operacje to:
  - readdir – odczyt katalogu.
  - map – zamapowanie fragmentu pliku w przestrzeń adresów procesu.
  - fsync – synchronizacja stanu pliku pomiędzy pamięcią a urządzeniem blokowym (zapisanie danych i meta-danych na urządzeniu)
  - check_media_change,revalidate – związane z wymiennymi nośnikami na urządzeniach blokowych

Uwagi na temat metod obiektu otwartego pliku

- Obiekt wykorzystywany jest również w przypadku katalogów.
- W momencie otwarcia pliku (funkcja systemowa open) tablica metod (pole f_op) jest inicjowana przy pomocy pola default_file_ops struktury i-node.
- Domyślna implementacja lseek zmienia pozycje pliku. read/write powinny również zmieniać pozycję o ilość przeczytanych/zapisanych bajtów.
- W przypadku gdy obiekt i-węzła ma zaimplementowaną metodę bmap tłumaczącą numer bloku w pliku na numer bloku na urządzeniu możliwe jest wykorzystanie gotowych funkcji korzystających z tej metody. Są to:
  - generic_file_read dla metody read w obiekcie pliku.
  - generic_file_mmap dla metody mmap w obiekcie pliku.
  - generic_readpage dla metody readpage obiektu i-węzła.

System plików ifs

- Skrót od ***Idiotic File System***.
- Składa się z katalogu głównego w którym rezyduje plik o nazwie "ala.txt" o zawartości "Ala ma kota".
  - Wszystkie dane przechowywane są w pamięci.
  - Nie wymaga urządzenia.
- Jest systemem tylko do odczytu – co ***znacznie*** upraszcza jego kod.
  - W szczególności trywialna jest synchronizacja.
- Implementacja mieści się w jednym pliku ifs.c, zaimplementowany jako moduł.
- Wykorzystuje różne metody i-węzła i otwartego obiektu pliku w przypadku plików zwykłych i katalogów.

Niezbędne dane

```
// Dane jedynego pliku w systemie
static char *file="Ala ma kota\n";
// Rozmiar tego pliku
#define FILESIZE 12

// Nazwa jedynego pliku w systemie
static char *filename="ala.txt";
// Długość tej nazwy
#define NAMESIZE 7


// numer i-węzła jedynego katalogu
#define ROOT 0

// numer i-węzła jedynego pliku
#define PLIK 1

// "magiczna liczba" identyfikująca system plików ifs
#define IFS_SUPER_MAGIC 0xabcd4321
```

- Przy tak prymitywnym systemie plików struktury danych są praktycznie nieistniejące. Poza tym nie chciałem za bardzo pomagać Państwu przy realizacji projektów !!!

Inicjalizacja i zwolnienie modułu

```
// 0 po "ifs" oznacza, że system nie wymaga urządzenia blokowego
struct file_system_type ifs_type = {ifs_read_super,"ifs",0,NULL };

int init_module()
{
   if (register_filesystem(&ifs_type)<0) {
      printk("Cannot register ifs filesystem\n");
      return -1;
   }
   return 0;
}

void cleanup_module()
{
   if (unregister_filesystem(&ifs_type)<0)
      printk("Cannot unregister ifs filesystem\n");
}
```

- System nie wymaga urządzenia więc montujemy go przy pomocy polecenia:

  mount none <katalog> -t ifs

Odczyt i zwolnienie super-bloku

```
struct super_block *ifs_read_super(struct super_block *s,void *data,
    int silent) {
   lock_super(s);
   s->s_blocksize = 1024;  s->s_blocksize_bits = 10;
   s->s_magic = IFS_SUPER_MAGIC;
   s->s_op = &ifs_sops;
   s->s_flags |= MS_RDONLY; // system plików tylko do odczytu
   unlock_super(s);
   // pobierz i-węzeł katalogu głównego
   if (!(s->s_mounted=iget(s,ROOT))) {
      s->s_dev = 0;
      printk("get root inode failed\n");
      return NULL;
   }
   MOD_INC_USE_COUNT; // zwiększ licznik odniesień modułu
   return s;
}

void ifs_put_super(struct super_block *sb) {
   iput(sb->s_mounted); // zwolnij i-węzeł katalogu głównego
   lock_super(sb);
   sb->s_dev = 0; // wymagane przez VFS, ustaw nr urządzenia na zero
   unlock_super(sb);
   MOD_DEC_USE_COUNT;
}
```

- Obiekt super bloku implementuje metody read_inode, put_super.

Tablice metod obiektów pliku oraz i-węzła dla plików zwykłych oraz katalogów

```
static struct file_operations ifs_file_file_ops = {
   read: ifs_read };

struct inode_operations ifs_file_inode_ops = {
   default_file_ops: &ifs_file_file_ops };
```

- Dla zwykłego pliku obiekt i-węzła nie implementuje żadnych metod. Wykorzystywane jest jedynie pole default_file_ops w celu inicjalizacji metod obiektu pliku. Obiekt pliku implementuje jedynie metodę read.

```
struct file_operations ifs_dir_file_ops = {
   readdir: ifs_readdir };

struct inode_operations ifs_dir_inode_ops = {
   default_file_ops: &ifs_dir_file_ops,
   lookup: ifs_lookup };
```

- W przypadku katalogu implementowane są metody lookup (dla i-węzła) i readdir (dla pliku).
- Tabela metod dla i-węzła jest inicjalizowana w metodzie read_inode obiektu superbloku.

Metoda read_inode

```
void ifs_read_inode(struct inode * inode)
{
   inode->i_op = NULL;
   inode->i_uid = inode->i_gid = 0; // właściciel root
   inode->i_size = 0;
   // ustaw wszystkie czasy na czas bieżący
   inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
   inode->i_blocks = 0; inode->i_blksize = 1024;
   // Mamy tylko dwa i-węzły w całym systemie plików
   switch (inode->i_ino) {
      case ROOT:
         inode->i_op=&ifs_dir_inode_ops; // tablica metod
         inode->i_nlink=2;
         // Katalog + każdy może czytać + każdy możę uruchomić
         inode->i_mode=S_IFDIR | S_IRUGO | S_IXUGO;
         break;
      case PLIK:
         inode->i_mode=S_IFREG | S_IRUGO;
         // Zwykły plik + każdy może czytać
         inode->i_op=&ifs_file_inode_ops; // tablica metod
         inode->i_nlink=1;
         inode->i_size=FILESIZE;
         inode->u.generic_ip=(void *)file; // wskazuje na dane pliku
         break;
   }
}
```

Metoda statfs

```
void ifs_statfs(struct super_block *sb, struct statfs *buf, int
   bufsiz){
    struct statfs tmp;

    tmp.f_type = IFS_SUPER_MAGIC;
    tmp.f_bsize = 1024;
    tmp.f_blocks = 0;
    tmp.f_bfree = 0;
    tmp.f_bavail = 0;
    tmp.f_files = 0;
    tmp.f_ffree = 0;
    tmp.f_namelen = 14;
    memcpy_tofs(buf, &tmp, bufsiz);
}
```

- Metoda zwraca informacje statystyczne o systemie plików.
- Funkcja memcpy_tofs kopiuje dane z pamięci jądra do pamięci procesu.
  - Do kopiowanie w przeciwną stronę możemy użyć funkcji memcpy_fromfs
- W funkcji sys_statfs implementującej wywołanie systemowe wykonywana jest najpierw weryfikacja poprawności adresu (verify_area).

Metoda lookup

```
int ifs_lookup(struct inode * dir, const char * name, int len,
struct inode ** result)
{
   struct super_block * sb=dir->i_sb;
   *result = NULL;
   if (!dir || !S_ISDIR(dir->i_mode))
      return -ENOENT;
   // Obsługa nazw "." oraz ".."
   if (!len || (name[0] == '.' && (len == 1 ||
                        (name[1] == '.' && len == 2)))) {
      *result = dir;
      return 0;
   }
   // Uwaga: każda metoda i-węzła musi wywołać iput na i-węźle !!!
   iput(dir);
   // Nasz jedyny plik
   if ((len==NAMESIZE) && !memcmp(name,filename,NAMESIZE)) {
      *result=iget(sb,PLIK);
      return 0;
   }
   return -ENOENT; // Nie znaleziono dowiązania
}
```

- Lookup poszukuje dowiązania o nazwie name (len jest długością nazwy i zwraca w result i-węzeł odpowiadający temu dowiązaniu)
- Uwaga na istotne uproszczenie: w przypadku wyszukiwania katalogu nadrzędnego zwracany jest katalog w którym prowadzimy wyszukiwanie. Jest to poprawne tylko w przypadku głównych katalogów systemu plików, które obsługiwane są specjalnie !!!

Metoda readdir

```
int ifs_readdir(struct inode * inode, struct file * file,
void * dirent, filldir_t filldir)
{
   int quit=0;
   if (!inode || !S_ISDIR(inode->i_mode))  return -EBADF;
   while(quit>=0) {
      switch((int)file->f_pos) {
          case 0: quit=filldir(dirent,".",1,0,inode->i_ino); break;
          // Uwaga: poniżej poprawne tylko dla głównego katalogu !!!
          case 1: quit=filldir(dirent,"..",2,1,inode->i_ino); break;
          case 2: quit=filldir(dirent,filename,NAMESIZE,2,PLIK); break;
      }
      file->f_pos++;
      if (file->f_pos>2) quit=-1;
  }
  return 0;
}
```

- readdir odczytuje katalog od bieżącej pozycji (dla katalogów pole f_pos jest pozycją w katalogu, a nie numerem bajtu).
- Jądro przekazuje adres funkcji int filldir(void *dirent,char *name,int len,int pos, int ino). Metoda readdir musi użyć tej funkcji do wypełnienia katalogu.
  - Wypełniamy od bieżącej pozycji (f_pos)
  - Jeżeli filldir zwróci wartość <0 to należy opuścić readdir
  - Uwaga !!! Ponownie pozycja ".." zwraca numer i-węzła dla samego siebie !!! Jest to poprawne ponieważ jedyny katalog w systemie plików jest jednocześnie katalogiem głównym.

Metoda read – odczyt danych z pliku

```
int ifs_read(struct inode * inode, struct file * file,char * buf, int
   count)
{
   int maxcount=inode->i_size - file->f_pos;
   if (count<0)
      return -EINVAL;
   // Czy nie chcemy odczytać za dużo bajtów ?
   if (count>maxcount)
      count=maxcount;
   if (count>0) {
      memcpy_tofs(buf,(char *)inode->u.generic_ip+file->f_pos,count);
      file->f_pos+=count;
   }
   return count;
}
```

- Przypomnienie: w read_inode ustawiliśmy pole u.generic_ip na bufor z danymi pliku.

System plików ext2 - historia

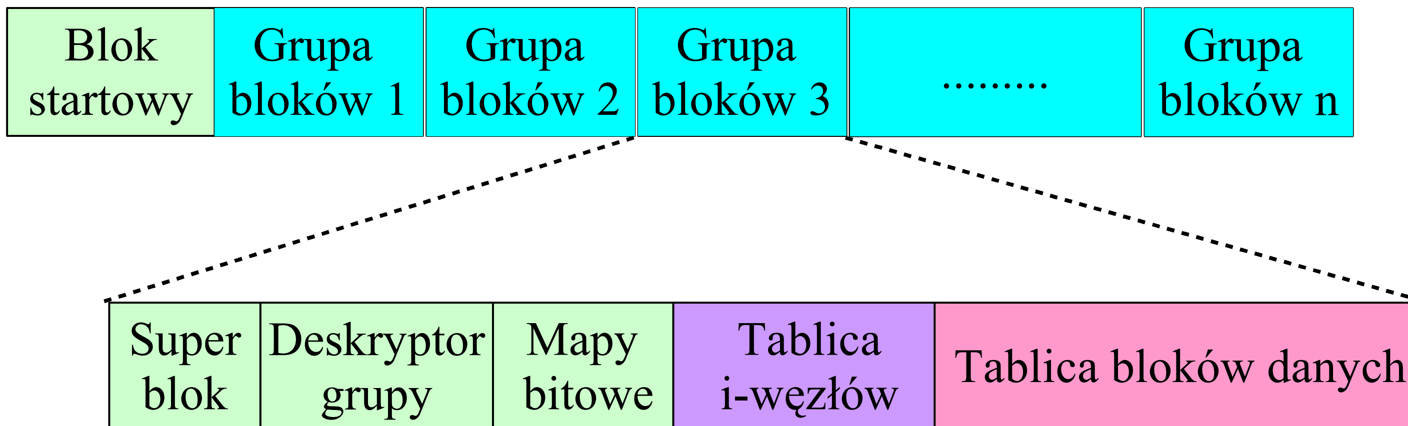
- Linuks został napisany na komputerze używającym dydaktycznego systemu operacyjnego Minix (A. S. Tanendbaum). I początkowo mógł korzystać tylko z tego systemu plików.
- Oczywiste wady systemu minix doprowadziły do opracowania systemu ext. Ext był jednak wolniejszy od minix.
- Następnie opracowano system plików xia (F. Xia) szybszy od ext i oparty na minix.
- System ext2 (R. Card) wywodzi się z ext. Jest standardowym systemem dla Linuxa 2.0.x.

	Minix	Ext	Xia	Ext2
Maks. rozmiar systemu plików	64MB	2GB	2GB	4TB
Maks. rozmiar pliku	64MB	2GB	64MB	2GB
Maks. Długość nazwy	14	255	248	255

Optymalizacje w systemie ext2 rozmieszczenie struktur na dysku

- Klasyczny system Unix

- System ext2



- Rozmieszczenie bloków, wzorowane na systemie ffs z Berkeley, pozwala na zmniejszenie odległości na dysku pomiędzy i-węzłem, jego blokami danych, oraz mapami bitowymi (alokacja nowych i-węzłów)

Optymalizacje w systemie ext2

- Alokuj nowe bloki danych:
  - O ile możliwe w odległości 32 bloków od bloku docelowego, jeżeli nie to
  - W obrębie tej samej grupy bloków, w której znajduje się i-węzeł, jeżeli nie to
  - W innych grupach bloków.
- Przydzielaj bloki danych z wyprzedzeniem. Przydzielanych jest maksymalnie 8 dodatkowych bloków. W momencie zamknięcia pliku niewykorzystane bloki są zwalniane.
- Zapisuj dane i metadane w kolejności minimalizującej prawdopodobieństwo awarii systemu.

Postać i-węzła w systemie ext2

0	Typ/uprawnienia	UID	Rozmiar
8	Czas dostępu		Czas utworzenia
16	Czas modyfikacji		Czas usunięcia
24	GID	Liczba dowiązań	Liczba blokówvdanych
32	Atrybuty plików		Zarezerwowane
40	12 bloków bezpośrednich		
88	Blok pośredni - 1 poziom		Blok pośredni - 2 poziom
96	Blok pośredni - 3 poziom		Wersja pliku
104	ACL pliku		ACL katalogu
112	Adres fragmentu		Zarezerwowane
128	Zarezerwowane		

- Uwaga na różnice: i-węzeł ext2, to coś zupełnie innego niż i-węzeł VFS (nazywany na tym wykładzie po prostu i-węzłem)

Pozycja katalogu w systemie ext2

Numer i-węzła	Długość pozycji	Długość nazwy	Nazwa

- Długość pozycji obejmuje wszystkie pola i jest zaokrąglana w górę do wielokrotności 4.
  - Umożliwia szybkie przejście do najbliższej pozycji