Rozdział 1

Algorytm `mknod`

Autor: Marcin Stefaniak

Niniejszy rozdział zawiera opis algorytmu funkcji systemowej `mknod`, która służy do tworzenia plików różnorakiego rodzaju. Opis dotyczy systemu Linux, nr wersji jądra 2.2.17

1.1 Wprowadzenie

1.1.1 Działanie

Funkcja systemowa `mknod` umożliwia tworzenie plików różnego rodzaju:

- **specjalnych**, w tym **znakowych** i **blokowych**, (ang. *character special file*, *block special file*),
- **kolejek FIFO**, czyli **łącz nazwanych** (ang. *named pipe*)
- **gniazd** [1] (ang. *socket*),
- a także zwykłych plików.

W systemie Linux funkcja `mknod` nie potrafi utworzyć katalogu. Do tego celu służy odrębna funkcja systemowa `mkdir`.

Z wyjątkiem tworzenia kolejek FIFO, wyłącznie nadzorca (ang. *root*) może wykonać funkcję systemową `mknod`.

Tworzenie zwykłego pliku nie różni się *efektem* od wywołania funkcji systemowej `creat`, z jednym wyjątkiem — nie otwieramy utworzonego pliku.

1.1.2 Pliki specjalne

Pliki specjalne w systemie Linux reprezentują **urządzenia** (ang. *devices*). Tworząc plik specjalny należy określić **numer urządzenia**. Numer urządzenia składa się z dwóch liczb: **głównego**

[1] `man` mylnie podaje, że tego `mknod` nie potrafi

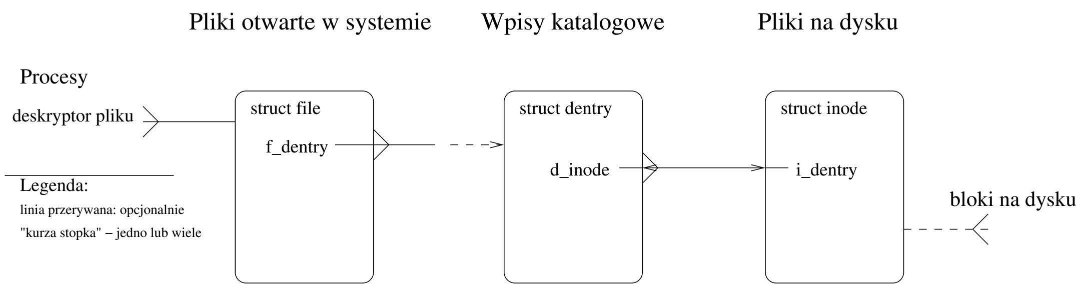
(ang. *major*) i **drugorzędnego** (ang. *minor*) numeru urządzenia. Numer główny określa rodzaj urządzenia, a numer podrzędny — które konkretnie urządzenie mamy na myśli.

Na przykład, partycja dysku twardego, na którym piszę to opracowanie, reprezentowana jest przez plik specjalny blokowy `/dev/hda7`, który ma główny numer 3 (taki sam jak dla innych partycji), a podrzędny 7 (specyficzny dla tej właśnie partycji).

Nowy plik specjalny tworzy się wówczas, gdy w systemie instalowany jest nowe urządzenie lub podprogram obsługi urządzenia. Funkcja systemowa `mknod` nie wiąże tworzonego pliku specjalnego z kodem obsługującym urządzenie o podanym numerze. Dzieje się to dopiero podczas próby otwarcia takiego pliku, a odbywa się to przy pomocy tzw. **tablic rozdzielczych**.

1.2 Struktury danych

Związki pojęciowe między istotnymi dla nas strukturami systemu plików obrazuje rysunek 1.1



Rysunek 1.1: schemat zależności w systemie plików

Główne struktury danych, na których dokonywane są operacje dla funkcji `mknod` to struktura `inode` i struktura `dentry`.

1.2.1 Struktura `inode`

Struktura `inode`[include/linux/fs.h] opisuje nam plik na dysku, w tym m.in.:

- bloki, które zajmuje na dysku,
- właściciela,
- prawa dostępu,
- rodzaj pliku
- rodzaj systemu plików
- data modyfikacji itp.

Dane ze struktury `inode` są zapisane na dysku (w **i-węźle** – rekordzie dyskowym). W momencie gdy system operuje na pliku, to dane z rekordu i-węzła są kopiowane do struktury `inode` i pozostają w pamięci jądra, dzięki czemu mogą być użyte ponownie. Taki mechanizm — pamięć podręczna (ang. *cache*) powszechnie stosowany jest przy pobieraniu danych z pamięci wolniejszej (dysk) do szybszej (pamięć operacyjna). Podobnie dzieje się np. z zawartością plików, która jest buforowana (służy do tego struktura `buffer_head`), jak też i ze strukturą katalogów (`dentry`)

Opis znaczenia istotnych dla nas tutaj pól struktury `inode` znajduje się w tablicy 1.1

Tablica 1.1: Wybrane pola struktury `inode`

Typ	**Nazwa**	**Znaczenie**
struct list_head	i_dentry	lista wpisów katalogowych, odnoszących się do danego i-węzła
unsigned long	i_ino	unikatowy numer i-węzła
unsigned int	i_count	licznik odwołań procesów do i-węzła
kdev_t	i_rdev	numer urządzenia (w przypadku pliku specjalnego)
umode_t	i_mode	określa typ i-węzła (czy jest katalogiem, plikiem specjalnym, itp.) oraz prawa dostępu
struct semaphore	i_sem	semafor binarny służący do blokowania i-węzła
struct inode_operations *	i_op	wskaźniki na funkcje charakterystyczne dla odpowiedniego systemu plików, operujące na danym i-węźle

1.2.2 Struktura `dentry`

Struktura `dentry`[include/linux/dcache.h] opisuje pojedynczy wpis katalogowy. Informacje o strukturze katalogów są zapisane na dysku w plikach zwanych **katalogami**, a postać tych plików zależy od rodzaju systemu plików. Informacje o strukturze katalogów są przechowywane podręcznie w pamięci jądra w strukturach `dentry`, prawie niezależnych od rodzaju systemu plików, co pozwala na ujednolicenie i przyspieszenie działań na strukturze katalogów (w szczególności, algorytmu rozbioru nazwy (ang. *name resolution*).

Struktury `dentry` tworzą drzewo (a raczej, uwzględniwszy dowiązania (ang. *link*), graf skierowany), na podobieństwo struktury katalogowej na dysku, bowiem odzwierciedlają jej część. W razie, gdy szukanego wpisu katalogowego nie ma w pamięci, to z dysku zostanie przeczytany odpowiedni katalog i stworzone zostaną potrzebne struktury `dentry`, przy czym w miarę potrzeb zwalniane będą nieużywane już takie struktury.

Opis znaczenia istotnych dla nas tutaj pól struktury `dentry` znajduje się w tablicy 1.2

Struktura `qstr` to nic innego jak napis wraz z jego wartością funkcji haszującej (zapamiętywaną dla większej efektywności).

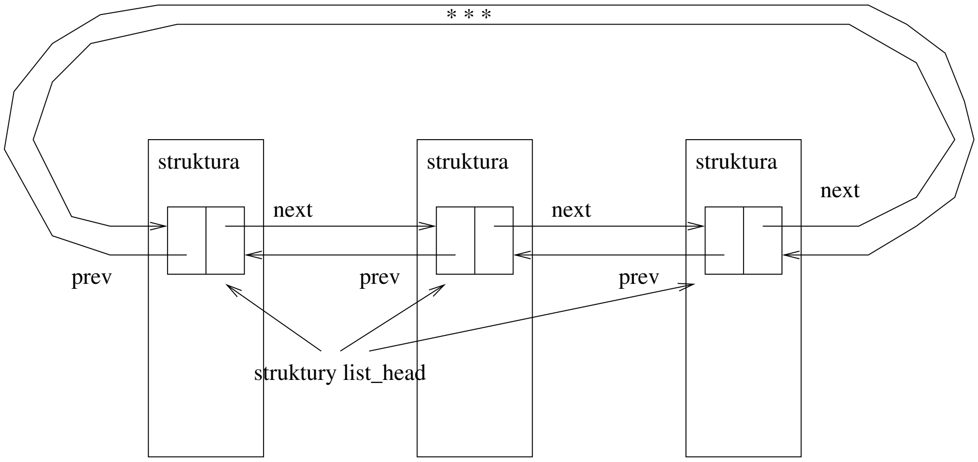
Dzięki licznikowi odwołań wiadomo, czy dany wpis katalogowy można zwolnić z pamięci.

Szczegóły implementacji pól `d_alias` i `i_dentry` są tematem zadania 2.

1.2.3 Struktura `list_head`

Warto zwrócić uwagę na strukturę `list_head`[include/linux/list.h], używaną w wielu fragmentach jądra systemu, przy użyciu której implementowane są dwukierunkowe listy cykliczne (rysunek 1.2). Jej definicja to

```
struct list_head {
  struct list_head *prev, *next
}
```



Rysunek 1.2: lista cykliczna w linuksie

Są tam też zdefiniowane specjalne makra do operowania na tych listach, przede wszystkim do uzyskania z wskaźnika na strukturę `list_head` wskaźnika na strukturę macierzystą.

W starszych wersjach jądra takie listy były realizowane przez pary wskaźników należących bezpośrednio do danej struktury. Dzięki wprowadzeniu struktury `list_head` kod jądra stał się bardziej jednolity, zwarty i przejrzysty.

Tablica 1.2: Wybrane pola struktury `dentry`

Typ	**Nazwa**	**Znaczenie**
int	d_count	licznik odwołań procesów do danego wpisu
struct inode *	d_inode	wskaźnik na i-węzeł, do którego dany wpis się odnosi
struct dentry *	d_parent	wskaźnik na wpis reprezentujący katalog macierzysty
struct list_head	d_alias	lista wpisów katalogowych, które odnoszą się do tego samego i-węzła, co dany
struct qstr	d_name	nazwa wpisu

Szczegóły implementacji struktury listy są tematem zadania 1.

1.3 Algorytmy

Zanim opiszemy algortym `mknod`, trzeba przybliżyć istotną funkcję z której ten algorytm korzysta — funkcję `lookup_dentry`.

1.3.1 Algorytm `lookup_dentry`

Funkcja `lookup_dentry`[fs/namei.c] służy do znalezienia (bądź stworzenia nowego) wpisu katalogowego o zadanej nazwie. Korzysta z niej wiele funkcji systemowych, oprócz `mknod` także `open`, `mkdir`, `link` i `unlink`.

Głównym zadaniem tej funkcji jest realizacja algorytmu rozbioru nazw. Dla przykładu, przypuśćmy że użytkownik chce skorzystać z plików o nazwach:

```
so/referat/linux-src/linux/fs/namei.c
../alg_mknod.tex1
/mnt/cdrom/linux/linux-2.2.17.tgz
```

Każdą z tych długich nazw trzeba zanalizować – rozbić na pojedyncze nazwy wpisów katalogowych, określić gdzie „zaczyna się” nazwa (katalog roboczy albo główny), i „wchodzić” do kolejnych katalogów z podanej ścieżki. Po drodze mogą się przydarzyć punkty montażu (ang. *mount point*), jak też i symboliczne dowiązania (ang. *symbolic link*), które kierują nas do innego miejsca w strukturze katalogowej.

Algorytm wydaje się jasny i oczywisty dla użytkownika systemu UNIX i podobnych, ale jest wystarczająco skomplikowany, aby implementacja zajęła ponad 200 linii kodu.

W niniejszym opracowaniu pominięto szereg istotnych niuansów, jak np. sposób znajdowania wpisu w linii 17., który wykorzystuje fakt trzymania w pamięci ostatnio używanych wpisów katalogowych. Do dysku sięgamy w ostateczności, gdy w pamięci nie znajdziemy szukanego wpisu. Nadmienić należy, że na czas odczytania informacji z katalogu zajmowany jest semafor w jego strukturze `inode`. Interesujący przykład przeplotu zdarzeń bez takiego zabezpieczenia, który kończy się blokadą, przedstawiony jest przy opisie algorytmu `link` w [Bach].

Jeśli podczas „przechodzenia” przez dowiązanie symboliczne natrafimy na ciąg kolejnych pięciu dowiązań symbolicznych, to następuje błąd zapętlenia.

Interpretacja znaku tyldy na początku ścieżki jako katalogu domowego użytkownika odbywa się na poziomie powłoki poleceń (np. `bash`).

1.3.2 Algortym `mknod`

Algorytm `mknod` zaimplementowany jest przez funkcje `sys_mknod` oraz `do_mknod`, część zależna od rodzaju systemu plików zawarta jest w funkcji `<fs>_mknod`, gdzie `<fs>` to oznaczenie systemu plików.

Dla ustalenia uwagi opiszemy jego wariant dla popularnego (w Linuxie) systemu plików [2] *ext2*.

[2] od wersji jądra 2.4 można użyć specjalnego systemu plików dla urządzeń *devfs*, podobnie jak *procfs* jest systemem plików dla procesów

Algorytm 1 `lookup_dentry`

Dane wejściowe: `name` – nazwa do rozwinięcia
`struct dentry * base` – alternatywny katalog początkowy (zamiast bazowego)
`lookup_flags` – tryb przeszukiwania (proste/złożone nazwy, czy interesuje nas znalezienie katalogu, czy

Wynik: `struct dentry *` oznaczający szukany przez nas wpis katalogowy

```
jeśli name zaczyna się od '/' to
   base := katalog główny
w przeciwnym przypadku
   jeśli base jest NULL to
      base := katalog bieżący
   koniec jeśli
koniec jeśli
jeśli name jest pusta to
   zwróć base i zakończ
koniec jeśli
wykonuj w pętli
   jeśli brak uprawnienia do wykonania katalogu base to
      wyjdź z błędem
   koniec jeśli
   odetnij od name początkowy kawałek aż do następnego '/'
   i usuń z reszty name początkowe '/'
   spróbuj znaleźć wpis katalogowy odpowiadający temu początkowemu kawałkowi
   jeśli koniec nazwy to
      zwróć znaleziony wpis, albo wpis „pusty", tzn. bez odpowiadającej mu struktury inode
      (w komentarzach do kodu jądra takie wpisy określane są jako negative – ujemne)
   koniec jeśli
   jeśli znaleziono wpis to
      jeśli koniec nazwy to
         zwróć ten wpis i zakończ
      koniec jeśli
      „przejdź" przez ten wpis, zmianiając odpowiednio base
      (to się wykonuje specyficznie dla wpisu, inaczej dla zwykłych katalogów, inaczej dla dowiązań symbolicznych, inaczej dla punktów montażu)
      jeśli nie udało się przejść to
         wyjdź z błędęm
      koniec jeśli
   w przeciwnym przypadku
      brak pliku w środku ścieżki - wyjdź z błędem
   koniec jeśli
koniec pętli
```

Jak już podałem we wprowadzeniu, funkcja systemowa `mknod` potrafi też tworzyć zwykłe pliki, niemal zupełnie tak jak to czyni funkcja systemowa `open/creat`. Jednakowoż przepływ sterowania w tych funkcjach dla przypadku tworzenia nowego zwykłego pliku pozostaje do końca niezależny (choć korzystają z wspólnych podprocedur, np. `lookup_dentry`). Trochę mi się to nie podoba, bo przez to mniej czytelna jest filozofia funkcjonowania systemu; z drugiej strony, rzadko się używa funkcji `mknod` do tworzenia zwykłych plików i jest to właściwie nieistotny dodatek (czyżby zaszłość historyczna?).

Algorytm `sys_mknod`

Funkcja `sys_mknod`[fs/namei.c] jest wykonywana w trybie jądra, gdy tylko użytkownik wywoła po swojej stronie funkcję `mknod`.

Algorytm 2 `sys_mknod`

Dane wejściowe: `filename` – nazwa pliku do utworzenia
`mode` – rodzaj pliku i prawa dostępu
`dev_t dev` – numer urządzenia
Wynik: tworzy nowy plik danego rodzaju albo zwraca błąd

zajmij jądro (`lock_kernel()`)
jeśli plik ma być katalogiem, lub też nie jesteśmy nadzorcą a plik nie jest kolejką FIFO **to**
  zwróć błąd
koniec jeśli
skopiuj nazwę pliku do pamięci jądra
(zabezpieczenie przed błędem braku strony)
wywołaj funkcję `sys_mknod`
zwolnij pobraną strukturę `dentry`, zwolnij nazwę pliku
zwolnij jądro (`unlock_kernel()`)

Algorytm `do_mknod`

Funkcja `do_mknod`[fs/namei.c] jest wywoływana przez funkcję `sys_mknod`.

Algorytm `ext2_mknod`

Funkcja `ext2_mknod`[fs/ext2/namei.c] wywoływana jest przez funkcję `do_mknod`.

Funkcja `ext2_new_inode()`[fs/ext2/ialloc.c] znajduje wolne miejsce na nowy i-węzeł wedle prostej strategii: szuka pierwszego wolnego począwszy od bloku i-węzła katalogu macierzystego (algorytm przydziału dla nowego katalogu jest bardziej wyrafinowany, lecz funkcja `mknod` nie tworzy katalogów).

W linii 10 następuje zaznaczenie zmian jako „brudne”, żeby system je zapisał na dysk. I tu należy się wyjaśnienie — dotyczy to *tylko* utworzenia nowego i-węzła i zmian w katalogu macierzystym. Te zmiany są trwałe, natomiast to, co się dzieje z treścią nowo stworzonego pliku, zależy już poźniej od jego typu, a konkretnie, od przypisanych mu operacji i-węzła. W przypadku zwykłego pliku to, co doń zapisujemy będzie zapisywana na dysk, w przypadku kolejki FIFO nie, w przypadku pliku specjalnego — wszystko może się zdarzyć.

Algorytm 3 `do_mknod`

Dane wejściowe: `filename` – nazwa pliku do utworzenia
`mode` – rodzaj pliku i prawa dostępu
`dev_t dev` – numer urządzenia

Wynik: tworzy nowy plik danego rodzaju, zwraca wpis katalogowy lub błąd

zmień prawa dostępu (`mode`) zgodnie z wartością `umask` procesu
znajdź wpis katalogowy odpowiadający danej nazwie pliku (`lookup_dentry()`)
jeśli nie udało się znaleźć wpisu **to**
  błąd
koniec jeśli
zajmij katalog macierzysty tego wpisu (`lock_parent()`)
jeśli brak uprawnień żeby stworzyć nowy plik w tym katalogu, lub taki wpis już istnieje (`may_create()`) **to**
  błąd
koniec jeśli
jeśli w strukturze `inode` dla tego katalogu jest operacja `mknod` **to**
  uwzględnij nowy plik w limitach (ang. *quota*), o ile kompilujemy jądro z obsługą limitów
  wywołaj operację `mknod` (w naszym przypadku będzie to `ext2_mknod`)
w przeciwnym przypadku
  błąd
koniec jeśli
zwolnij katalog macierzysty (`unlock_dir()`)
zwróć wpis katalogowy stworzonego pliku

Algorytm 4 `ext2_mknod`

Dane wejściowe: `struct inode *dir` – i-węzeł katalogu macierzystego
  `struct dentry *dentry` – nowy, pusty wpis katalogwy
  `mode` – rodzaj pliku i prawa dostępu
  `dev_t dev` – numer urządzenia

Wynik: w systemie plików ext2 tworzy nowy plik danego rodzaju, zwraca kod błędu

1: stwórz (allokuj) nowy `inode` (i-węzeł) (`ext2_new_inode()`)
2: dodaj do katalogu `dir` wpis określający ten nowy i-węzeł
3: ustaw właściciela, tryb i operacje dla nowego i-węzła
4: **jeśli** ma to być kolejka FIFO **to**
5:   zainicjuj kolejkę (`initfifo()`)
6: **koniec jeśli**
7: **jeśli** ma to być plik specjalny **to**
8:   ustaw odpowiedni numer urządzenia tego i-węzła
9: **koniec jeśli**
10: zaznacz ten i-węzeł „brudny” (`mark_inode_dirty()`)
  podobnie oznacz „brudny” bufor, otrzymany w linii 1 (`mark_buffer_dirty()`).
11: **jeśli** system plików jest synchronizowany **to**
12:   natychmiast zapisz informacje o zmianach
13: **koniec jeśli**
14: ustanów odpowiednie wskaźniki między strukturami `dentry` i `inode` nowo utworzonego pliku (`d_instantiate()`)
15: zwolnij użyty bufor i ewentualnie `inode`

Inaczej jest w przypadku tworzenia łącza nienazwanego (`pipe`), które różni się od łącza nazwanego tym, że nie przynależy nigdzie w strukturze katalogowej systemu plików. Tym samym struktura `inode` łącza nienazwanego nie powinna nigdy zostać zapisana na dysk. Aby to zapewnić, stosuję się chytrą metodę: ustawia się w tej strukutrze znacznik „brudny", ale *nie* wpisuje się jej na listę struktur `inode` przeznaczonych do zapisu na dysk. System nigdy później sam nie doda jej do tej listy, gdyż będzie mu się zdawało, że już tam jest.

1.4 Zadania

1. a) Dlaczego warto używać list dwukierunkowych, cyklicznych?
   b) Jak używa się struktury `list_head` (napisz makro)

2. W linii 14 algorytmu `ext2_mknod` następuje „wstawienie" nowo stworzonej struktury `inode` do, również nowej, struktury `dentry`. Oto kod funkcji `d_instantiate`[fs/dcache.c]

```
void d_instantiate(struct dentry *entry, struct inode * inode)
{
  if (inode)
    list_add(&entry->d_alias, &inode->i_dentry);
  entry->d_inode = inode;
}
```

   Funkcja `list_add`[include/linux/list.h]:

```
static __inline__ void __list_add(struct list_head * new,
                                  struct list_head * prev,
                                  struct list_head * next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}

/*
 * Insert a new entry after the specified head..
 */
static __inline__ void list_add(struct list_head *new,
                                struct list_head *head)
{
  __list_add(new, head, head->next);
}
```

   Co właściwie się tu dzieje?

1.5 Rozwiązania zadań

1. a) Są proste w implementacji, jak też i efektywne.
   b) W pliku `include/linux/list.h` jest zdefiniowane takie oto makro:

   ```
   #define list_entry(ptr, type, member) \
     ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
   ```

   Można go użyć w taki sposób:

   ```
     struct inode * inode = ... ;
     struct list_head * plist_head = inode->i_list.next;
     struct inode * inode_next = list_entry(p_list_head, struct inode, i_list);
   ```

   Makro `list_entry` zwraca wskaźnik na strukturę macierzystą danej struktury `list_head`, pod warunkiem, że podamy nazwę pola tej listy (`member`).

2. Oprócz ustawienia wskaźnika na `inode` w strukturze `dentry`, co jest raczej jasne, element listy `entry->d_alias` wstawiany jest do listy `inode->i_dentry`. Tutaj mamy nowo utworzony `inode` i `dentry`, więc te listy są puste (tj. wskazują na samych siebie), a zatem wykonanie `list_add` spowoduje złączenie tych elementów. Ta funkcja jest wywoływana wtedy, gdy stworzone zostjaje nowe `dentry`, np. przy funkcji systemowej `link`. Może zachodzić taka sytuacja, że na jeden `inode` przypada wiele `dentry`.

   Wniosek stąd taki, że obie te listy, to tak naprawde ta sama lista, zawierająca zawsze dokładnie jeden `inode` i jeden lub więcej `dentry`, które należy utożsamić.

   Wydaje się dziwne, że jedna lista zawiera w sobie elementy różnych typów, zważywszy na poprzednie zadanie. Jak w takiej sytuacji można ją przeglądać? Na szczęście wiemy, że jest tylko jeden `inode`, a dla każdego `dentry` możemy łatwo wyznaczyć ten `inode`; i możemy obejść wszystkie elementy począwszy od niego — wtedy będziemy pewni, że przechodzimy przez `dentry`, za wyjątkiem pierwszego/ostatniego elementu.

   Oczywiście, ta metoda nie działa dla związku łączącego wiele struktur jednego rodzaju z wieloma strukturami drugiego rodzaju; albo dla związku łączącego więcej niż dwa typy struktur.

Literatura

[Bach] Bach J. M.: *Budowa systemu operacyjnego UNIX.*

[źródła jądra] źródła jądra systemu operacyjnego Linux wersja 2.2.17, w tym:

[include/linux/fs.h]

[include/linux/dcache.h]

[include/linux/list.h]

[fs/namei.c]

[fs/dcache.c]

[fs/ext2/namei.c]

[fs/ext2/ialloc.c]