Rozdział 1

Funkcja open

1.1 Przeznaczenie funkcji `open()`, wiadomości ogólne

Aby skorzystać z pliku[1] proces użytkownika musi najpierw przygotować go do obsługi. W systemie Linux trzeba to robić zgodnie ze schematem przyjętym w warstwie "Interfejsu do systemu plików". Na początku zawsze trzeba wywołać funkcje systemowa *open()*, której zadaniem jest przygotowanie struktur (w jądrze i w procesie) pozwalających na traktowanie pliku, jako ciągu bajtów opatrzonego nazwą. Operację tą ze względu na tłumaczenie z angielskiego słowa *open* potocznie określa się jako "otworzenie pliku".

Z otwarciem pliku związany został sposób jego obsługi. Możemy np. zażądać przygotowania pliku jedynie do czytania lub do dopisywania na końcu i połączyć to z wymaganiem dostępu synchronicznego. Spowoduje to, iż wszystkie dalsze operacje na pliku będą wykonywane zgodnie z sposobem jego otwarcia i umożliwi sprawdzanie, które operacje są dozwolone, a które nie. Ponieważ po otwarciu pliku staje się on dostępny dla różnorakich operacji jest to odpowiedni moment na sprawdzenie czy użytkownik posiada odpowiednie prawa dostępu do pliku. Gdyby tak nie było otwieranie nie powinno się powieść.

Wartością zwracaną przez `open()` jest deskryptor pliku. W ramach jednego procesu identyfikuje on jednoznacznie jedno otwarcie pliku. Jest to informacja lokalna dla procesu, tzn. w różnych procesach temu samemu deskryptorowi mogą odpowiadać różne pliki. Wszystkie dalsze operacje proces przeprowadza za pomocą tego właśnie deskryptora.

1.2 Opis wykorzystywanych struktur danych

System operacyjny musi udostępniać funkcje systemowe do tworzenia, zapisywania, czytania, usuwania, skracania plików oraz zmiany położenia w pliku. Większość wymienionych informacji wymaga przeszukania katalogu w celu odnalezienia w nim wpisu skojarzonego z plikiem. Większość systemów w celu uniknięcia ciągłego przeszukiwania otwiera plik i zapisuje informacje o nim w tablicy otwartych plików. Potem wystarczy użyć indeksu tej tablicy, by wykonać jakąś operację plikową. W systemie operacyjnym Linux 2.2.17 przyjęto nieco bardziej skomplikowane rozwiązanie

[1] Przez plik rozumiemy tu obiektu udostępniany przez warstwę systemu plików: "Interfejs do systemu plików" identycznie obsługiwane są pliki dyskowe, łącza nienazwane, kolejki FIFO, pliki NFS, ...

dotyczące obsługi plików. Zamiast jednej tablicy wewnętrznej istnieją aż trzy poziomy takich tablic.

1.2.1 Procesy a system plików

Każdy plik jest otwierany przez konkretny proces. Wszelkie dane o procesie zebrane są w strukturze `task_struct`. Każdy proces może się do niej odwołać poprzez wskaźnik `current` (w rzeczywistości jest to system makr i funkcji asemblerowych). Pełna postać tej struktury znajduje się w pliku: `<linux/sched.h>`. Z punktu widzenia obsługi plików interesujące są tylko dwa spośród wielu pól:

```
struct task_struct {
  ...
  struct fs_struct    fs;         /* informacje o związkach procesu
                                     z systemem plików */
  struct files_struct files;      /* informacje o otwartych plikach procesu */
  ...
};
```

Uwaga: w strukturze `task_struct` istotna jest kolejność poszczególnych składników, gdyż niektóre spośród funkcji asemblerowych odwołują się do pól tej struktury poprzez adresy względne.

Zajmijmy się najpierw pierwszą z tych struktur, a mianowicie strukturą `fs_struct`:

```
struct fs_struct {
  atomic\_t      count;      /* zarezerwowana do dalszych rozszerzeń */
  int           umask;
  struct dentry  *root;
  struct dentry  *pwd;
};
```

Znaczenie poszczególnych pól jest następujące:

`int umask;`

Na tej zmiennej przechowywana jest maska praw nadawana plikom tworzonym przez dany proces. Na wartość tej zmiennej można wpływać za pośrednictwem wywołania systemowego umask.

`struct dentry * root;`

Jest to wskaźnik do struktury opisującej korzeń systemu plików dostępnego dla danego procesu. Używana jest ona do określania bezwzględnych ścieżek dostępu do plików, czyli takich, które rozpoczynają się od korzenia systemu plików. Warto dodać, iż użytkownik specjalny poprzez funkcję `chroot` może zmienić katalog, który opisywany jest przez tę strukturę. Pole to wykorzystywane jest przykładowo przy obsłudze Anonimowego FTP.

`struct dentry * pwd;`

Wskaźnik do struktury zawierającej dane o aktualnym katalogu. Używana jest ona do określania względnych ścieżek dostępu do plików, to jest ścieżek rozpoczynających się w katalogu bieżącym. Poprzez funkcję systemową `chdir` możliwa jest zmiana tego katalogu.

Druga struktura, tzn. `files_struct` jest nieco bardziej rozbudowana:

```
struct files_struct {
  atomic_t     count;
```

```
  int           max_fds;
  int           max_fdset;
  int           next_fd;
  struct file   **fd;
  fd_set        *close_on_exec;
  fd_set        *open_fds;
  fd_set        close_on_exec_init;
  fd_set        open_fds_init;
  struct file   *fd_array[NR_OPEN_DEFAULT];
};
```

Oto opis jej poszczególnych pól:

`atomic_t count;`

W polu tym zliczane są odwołania do tej struktury przez różne procesy.

`int max_fds;`

Określa rozmiar tablicy `fd`, czyli maksymalną liczbę deskryptorów plików, które proces może otworzyć. Pole to jest inicjowane wartością `NR_OPEN_DEFAULT`. Stała ta jest równa `BITS_PER_LONG`, czyli 32. W razie potrzeby rozmiar tablicy, a zatem i wartość tego pola, mogą być zwiększone (patrz opis tablicy `fd` poniżej).

`int max_fdset;`

Określa rozmiar zbiorów `close_on_exec` i `open_fds`. Jest zwiększana wraz z powiększeniem tablicy fd do rozmiaru przekraczającego wielkość tych zbiorów. Operacja ta jest realizowana przez funkcję `expand_fdset`.

`int next_fd;`

Jest to numer, od którego należy zacząć poszukiwanie wolnego deskryptora. Po inicjalizacji jest to numer pierwszego wolnego deskryptora, potem niekoniecznie. Będzie tak z pewnością w przypadku, gdy zamykamy deskryptor (funkcja close) o numerze mniejszym od wartości aktualnie przechowywanej w tym polu. Wówczas bowiem pole zostanie zaktualizowane przyjmując wartość równą zwalnianemu deskryptorowi. W sytuacji zaś gdy zajmowany jest deskryptor o numerze wskazywanym przez to pole (funkcja open), jego wartość zwiększana jest o jeden. Nie musi już zatem ono wskazywać wolnego deskryptora, a jedynie miejsce, od którego należy takowego zacząć szukać. Prawdą jest natomiast, iż jeżeli jest to numer wolnego deskryptora, to jest to najmniejszy wolny deskryptor. Innymi słowy wszystkie deskryptory o numerach mniejszych od `next_fd` są zajęte.

`struct file ** fd;`

Jest to tablica wskaźników do struktur plików (`file`) indeksowana numerami deskryptorów. Deskryptory są niewielkimi liczbami całkowitymi, zwracanymi przez funkcję open lub create. Wszystkie pozostałe funkcje odwołują się następnie do tego pliku poprzez deskryptory. Używając deskryptora jako indeksu tablicy fd uzyskujemy zatem wskaźnik do struktury pliku, który umożliwia dostęp do samego pliku otwartego przez dany proces. Deskryptor pliku jest zawsze mniejszy od opisanej wyżej zmiennej `max_fds`. Wielkość tablicy fd jest zmieniana dynamicznie. Początkowo fd jest adresem tablicy `fd_array`, a jej rozmiar tablicy jest równy `NR_OPEN_DEFAULT`. W sytuacji jednak gdy proces potrzebuje dostępu do większej liczby deskryptorów, tablica jest powiększana - za pierwszym razem do 256 pozycji, następnie do rozmiaru jednej strony (czyli do `PAGE_SIZE / sizeof(struct file *)` pozycji). Każde kolejne powiększenie powoduje dwukrotny wzrost liczby pozycji w tablicy fd. Liczba to jest ograniczona przez stałą `NR_OPEN`, której domyślna wartość to 1024*1024 (czyli dwa do potęgi dwudziestej). Należy jednak zwrócić uwagę, iż wspomniana stała jest na ogół większa od maksymalnej liczby wszystkich otwartych plików w systemie (`NR_FILE = 4096`). Oprócz tego trzeba uwzględnić pliki zarezerwowane dla roota (`NR_RESERVED_FILES = 10`) oraz otwarte przez inne procesy. Maksymalna wielkość tablicy fs zatem będzie z pewnością dużo mniejsza. Podane stałe mogą być oczywiście różne w różnych systemach. Operacja powiększania tablicy fs jest realizowana przez funkcję `expand_fd_array`.

`fd_set * close_on_exec;`
Pole to określa zbiór deskryptorów plików, które mają zostać zamknięte w momencie wywołania przez proces funkcji systemowej exec. Zbiór ten jest zapisany jako mapa bitowa, tzn. deskryptor o numerze n należy do zbioru wtedy i tylko wtedy, gdy n-ty bit jest ustawiony. Początkowo jest to adres pola `close_on_exec_init`, ale powiększenie zbioru tych deskryptorów spowoduje zmianę adresu, gdyż jest to związane z alokacją pamięci na nowy zbiór. Wartość zbioru `close_on_exec_init` pozostanie bez zmian.

`fd_set * open_fds;`
To pole również określa zbiór deskryptorów plików procesu. Zbiór ten składa się jednak, w odróżnieniu od zbioru powyżej, z wszystkich otwartych przez dany proces deskryptorów. Pole jest inicjalizowane adresem pola `open_fds_init`.

`fd_set close_on_exec_init;`
Jest to początkowy zbiór deskryptorów plików procesu, które zostaną zamknięte w momencie wywołania przez proces funkcji exec.

`fd_set open_fds_init;`
Zbiór wszystkich deskryptorów plików otwartych przez proces w momencie jego utworzenia.

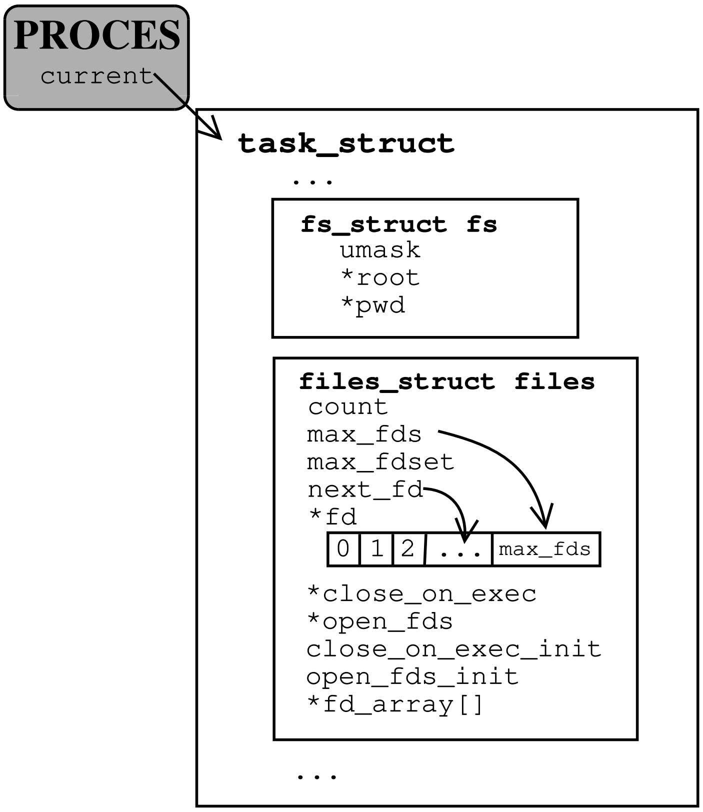
`struct file * fd_array[NR_OPEN_DEFAULT];`
Jest to podobnie do pola fs tablica wskaźników do struktur plików indeksowana numerami deskryptorów plików. Obie struktury na początku są identyczne (mają te same adresy), różnica polega na tym, iż rozmiar struktury fs jest potem zmieniany dynamicznie.

Definicję obu struktur można obejrzeć w pliku: `<linux/sched.h>`. Schemat powiązań procesów z obsługą plików jest przedstawiony na rysunku 1.1. Warto jeszcze dodać, iż obie struktury moga być współdzielone przez kilka procesów. Taka sytuacja powoduje, iż zmiany dotyczące deskryptorów plików (otwieranie, zamykanie) wprowadzone przez jeden proces znajdują swoje odzwierciedlenie w tablicach deskryptorów pozostałych procesów.

1.2.2 Procesowa tablica plików

W tablicy deskryptorów otwartych przez proces plików znajdują się wskaźniki do struktur plików. Struktura ta jest zdefiniowana w pliku: `<linux/fs.h>` i ma postać:

```
struct file {
  struct file     *f_next, **f_pprev;
                  /* wskaźniki do sąsiednich elementów na liście */
  struct dentry   *f_dentry;      /* wskaźnik do struktury dentry danego pliku */
  struct file_operations  *f_op;
                  /* dopuszczalne operacje plikowe,
 np. read, write, open */
  mode_t          f_mode;         /* prawa dostępu */
  loff_t          f_pos;          /* aktualna pozycja w pliku */
```



Rysunek 1.1: Podstawowy schemat powiązań procesów z obsługą plików

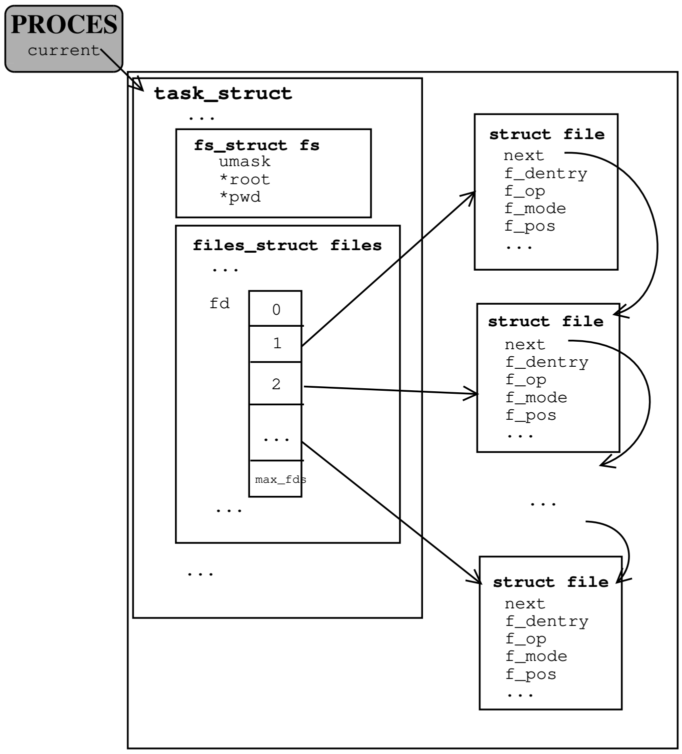
```
  unsigned int    f_count;        /* licznik odwołań */
  unsigned int    f_flags;        /* tryb dostępu (flagi funkcji open) */
  unsigned long   f_reada, f_ramax, f_raend, f_ralen, f_rawin;
                  /* wartości dotyczące czytania z wyprzedzeniem */
  struct fown_struct  f_owner;    /* dane dotyczące procesu-właściciela */
  unsigned int    f_uid, f_gid;
  int             f_error;
  unsigned long   f_version;      /* zarządzanie wersjami DCache */
  void            *private_data;  /* potrzebne m.in. dla sterownika tty */
};
```

Wszystkie te struktury plików razem tworzą tablicę plików otwartych w procesie. Właściwie jest to dwukierunkowa lista cykliczna korzystająca ze wskaźników `f_next` i `f_pprev`, ale potocznie nazywana jest ona procesową tablicą plików. Teraz można już uzupełnić schemat z rysunku pierwszego o struktury plików (patrz rysunek 1.2). Należy zauważyć, iż wspomniana w dozwolonych operacjach funkcja open, nie jest tą samą funkcją, której algorytm zostanie omówiony. Dowodem na to niech będzie wyraźna różnica w przyjmowanych argumentach obu funkcji. Interesująca nas funkcja open jest funkcją systemową i zajmuje się wszelkimi niezbędnymi czynnościami dotyczącymi zwykłych plików, zatem druga z funkcji może być użyteczna tylko dla programów obsługi urządzeń.
Druga uwaga dotyczy licznika odwołań. Proces może wielokrotnie otwierać ten sam plik i każdej takiej operacji odpowiadać będą różne deskryptory oraz struktury pliku. Istnieją jednak funkcje jak `dup`, które poprzez kopiowanie deskryptorów zwiększają liczbę odwołań do danego pliku.

1.2.3 Struktura katalogów

W systemie Linux pliki zorganizowane są w drzewiastą strukturę katalogów. Właściwie należało by mówić o grafie acyklicznym, gdyż poprzez linki w strukturze tej mogą istnieć powiązania między plikami. Po części dla uproszczenia, a po części ze względu na tradycję rozpatruje się jednak drzewo katalogów. Katalogi też są plikami, z tą różnicą, że zawierają one informacje o tym jak odnaleźć inne pliki. Nazwą ścieżkową do pliku (dowolnego) nazwiemy napis, który opisuje drogę prowadzącą do tego pliku w strukturze katalogów, umożliwiając w ten sposób jego identyfikację. Struktura pliku zawiera w sobie wskaźnik do struktury dentry (pole `f_dentry`), zdefiniowanej w pliku: `<linux/dcache.h>`.

```
struct dentry {
  int               d_count;      /* licznik odwołań */
  unsigned int      d_flags;
  struct inode      *d_inode;     /* struktura inode pliku opisywanego
                    przez tę strukturę */
  struct dentry     *d_parent;    /* katalog nadrzędny, tzn. katalog, do którego
                    należy plik zwykły
lub nadkatalog w przypadku katalogu */
  struct dentry     *d_mounts;    /* korzeń drzewa katalogów zamontowanego
                    w danym
katalogu systemu plików */
  struct dentry     *d_covers;
```



Rysunek 1.2: Rozszerzony schemat powiązań procesów z obsługą plików

```
  struct list_head  d_hash;       /* lista haszująca, do której
                    należy dany plik */
  struct list_head  d_lru;        /* lista nieużywanych pozycji plikowych LRU,
                    struktura jest dołączona do tej listy, gdy d_count = 0 */
  struct list_head  d_child;      /* lista dzieci naszego ojca,
                    tzn. lista braci */
  struct list_head  d_subdirs;    /* lista dzieci danego pliku-katalogu */
  struct list_head  d_alias;      /* lista aliasów dla d_inode */
  struct qstr       d_name;       /* dane dotyczące nazwy pliku */
  unsigned long     d_time;       /* używane przez d_revalidate */
  struct dentry_operations  *d_op;   /* operacje dozwolone dla tego typu pliku */
  struct super_block  *d_sb;      /* korzeń drzewa katalogów,
                    do którego należy dany plik */
  unsigned long     d_reftime;    /* czas ostatniego odwołania */
  void              *d_fsdata;    /* dane dotyczące konkretnego systemu plików */
  unsigned char     d_iname[DNAME_INLINE_LEN];
};
```

Dane dotyczące nazwy pliku są zapisane w strukturze:

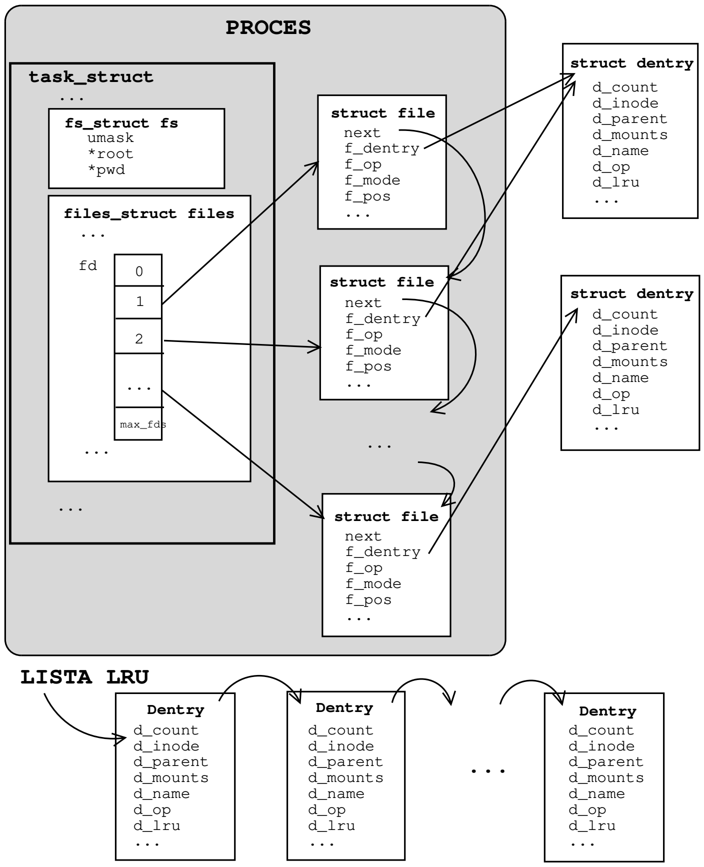
```
struct qstr {
  const unsigned   char * name;    /* nazwa pliku */
  unsigned int     len;            /* długość nazwy pliku */
  unsigned int     hash;           /* zmienna zależna od nazwy i jej długości */
};
```

Wszystkie używane aktualnie struktury umieszczone są na listach haszujących. Przyspiesza to procedurę szukania, gdyż nie trzeba za każdym razem przeglądać wszystkich struktur w systemie (potencjalnie). Nieużywane struktury zaś umieszczane są na liście LRU (ang. *Least Recently Used*). Struktura dentry jest uznawana za nieużywaną, jeżeli jej licznik odwołań jest równy zeru. Oznacza to, iż w przypadku pliku zwykłego nie jest on otwarty przez żaden proces, w przypadku katalogu zaś, iż nie jest otwarty żaden plik w tym katalogu, ani w żadnym z jego podkatalogów.

We wcześniejszych wersjach systemu Linux struktury dentry używane były jako podręczna pamięć buforowa nazw katalogów. Takie podejście było jednak sprzeczne z ideą Wirtualnego Systemu Plików, w którym przyjęto zasadę, iż należy ukryć szczegóły implementacyjne za warstwą programową. Do tej pory katalogi miały uprzywilejowaną pozycję w stosunku do innych plików, teraz zaś są traktowane na równi z nimi. Każdy rodzaj pliku ma charakterystyczne dla siebie operacje (pole `d_op`), które mogą (choć nie muszą) odróżniać go od innych typów plików. Schemat powiązań struktur dentry ze strukturami danych przedstawionymi w poprzednich punktach można obejrzeć na rysunku 1.3.

1.2.4 Ogólnosystemowa tablica plików

Dzięki strukturom dentry i operacjach na nich zdefiniowanych istnieje możliwość dostępu do i-węzłów. I-węzeł (ang. *inode*) jest strukturą, w której przechowuje się większość informacji o konkretnym pliku na dysku. Aby móc się posługiwać i-węzłami jądro musi wczytać je do pamięci. Poza skopiowaniem i-węzła z urządzenia blokowego zapamiętywanych jest też szereg dodatkowych informacji. Ostatecznie struktura i-węzła w pamięci systemu Linux jest następująca:



Rysunek 1.3: Struktury danych dotyczące plików

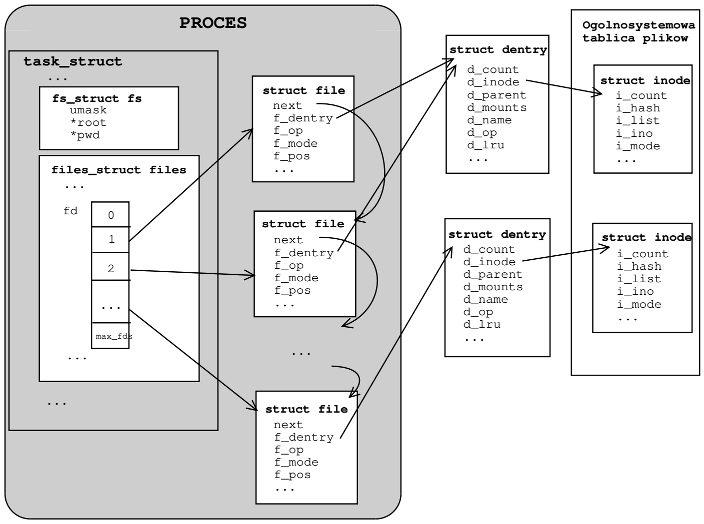
```
struct inode {
  struct list_head  i_hash;       /* lista haszująca, do której
                    należy dany i-węzeł */
  struct list_head  i_list;
  struct list_head  i_dentry;
  unsigned long     i_ino;        /* numer struktury inode */
  unsigned int      i_count;      /* licznik odwołań */
  kdev_t            i_dev;        /* numer urządzenia plikowego */
  umode_t           i_mode;       /* typ pliku i prawa dostępu */
  nlink_t           i_nlink;      /* liczba sztywnych łączy */
  uid_t             i_uid;        /* właściciel */
  gid_t             i_gid;        /* właściciel */
  kdev_t            i_rdev;       /* urządzenie w przypadku
                    urządzenia plikowego */
  off_t             i_size;       /* rozmiar pliku */
  time_t            i_atime;      /* czas ostatniego dostępu */
  time_t            i_mtime;      /* czas ostatniej modyfikacji */
  time_t            i_ctime;      /* czas utworzenia */
  unsigned long     i_blksize;    /* rozmiar bloku */
  unsigned long     i_blocks;     /* liczba bloków */
  unsigned long     i_version;    /* zarządzanie wersjami DCache */
  unsigned long     i_nrpages;
  struct semaphore  i_sem;        /* kontrola dostępu */
  struct semaphore  i_atomic_write;
  struct inode_operations  i_op;  /* dopuszczalne operacje na strukturze inode */
  struct super_block  i_sb;       /* blok specjalny */
  struct wait_queue  i_wait;      /* kolejka do oczekiwania */
  struct file_lock  *i_flock;     /* blokady plików */
  struct vm_area_struct  *i_mmap;  /* obszary pamięci */
  struct page       *i_pages;
  struct dquot      *i_dquot[MAXQUOTAS];
  unsigned long     i_state;      /* stan i-węzła */
  unsigned int      i_flags;      /* flagi = i_sb->s_flags */
  unsigned char     i_pipe;       /* struktura reprezentuje potok */
  unsigned char     i_sock;       /* struktura reprezentuje gniazdo */
  int               i_writecount;  /* liczba uprawnień do zapisu */
  unsigned int      i_attr_flags;
  __u32             i_generation;
  union {
    struct pipe_inode_info   pipe_i;
    struct minix_inode_info  minix_i;
    struct ext2_inode_info   ext2_i;
    ...
  } u;                            /* informacje zależne od systemu plików */
};
```

Warto podkreślić, iż w i-węźle nie przechowuje się ścieżki dostępu i nazwy pliku (w Linuxie plik może mieć więcej niż jedną nazwę). Struktury inode są zapisane w tablicy liniowej na urzadzeniu blokowym (np. twardym dysku). W pamięci natomiast są one przechowywane w tablicy haszującej, znanej pod nazwą ogólnosystemowej tablicy plików. Każdy plik ma jednoznacznie przypisany i-węzeł, zaś każdy i-węzeł ma niepowtarzalny numer w ramach danego systemu plików. Znając zatem system plików oraz numer i-węzła można zidentyfikować plik, a zarazem kolejkę haszującą, w której powinien się znaleźć. Zauważmy ponadto, iż dane zawarte w pliku nie są zapamiętywane w i-węźle - są to dwie oddzielne rzeczy. W i-węźle zapisane jest tylko (i aż) położenie pliku na dysku lub innym urządzeniu blokowym (właściwie położenie w pamięci) oraz inne informacje niezależne od procesów.
Operacja otwarcia pliku przez jakiś proces powoduje przyporządkowanie wolnego deskryptora nowo utworzonej strukturze plikowej w procesowej tablicy otwartych plików. W strukturze tej znajduje się wskaźnik do odpowiedniej pozycji w ogólnosystemowej tablicy struktur dentry, a przez to do ogólnosystemowej tablicy plików. W przypadku gdy plik ten jest już otwarty przez jakiś proces (być może ten sam), pociąga to za sobą tylko i wyłącznie zwiększenie licznika odwołań (`i_count`). Analogiczne liczniki mają zarówno struktury dentry jak i każdy i-węzeł. Operacja zamknięcia zmniejsza ten licznik, a gdy nastąpi jego wyzerowanie, plik przestaje być potrzebny i jego wpis może zostać usunięty z tablicy otwartych plików. Pełna definicja i-węzła znajduje się w pliku: `<linux/fs.h>` .

1.2.5 Podsumowanie

Każdy wpis w procesowej tablicy plików wskazuje pośrednio (przez strukturę dentry) na ogólnosystemową tablicę plików. W procesowej tablicy plików przechowywane są aktualne pozycje w plikach, uaktualniane po każdej operacji czytania i pisania. Gdyby w tablicy deskryptorów trzymać bezpośrednie wskaźniki do i-węzłów, to ta pozycja musiałaby być przechowywana w i-węźle. Jednak plik może być otworzony przez więcej niż jeden proces, bądź nawet przez ten sam proces, ale wielokrotnie, więc dla każdego pliku musiałoby istnieć kilka i-węzłów w pamięci. To natomiast nie byłoby dobrym pomysłem za względów ekonomicznych w przypadku, gdyby wszystkie i-węzły odwzorowywały plik w tę samą przestrzeń adresową. W przeciwnym wypadku zaś mogłoby to w spowodować, iż system nie wiedziałby, dane z którego i-węzła zapisać na dysk. W celu ograniczenia operacji dyskowych bowiem, system VFS nie zapisuje i-węzłów natychmiast, lecz przechowuje je w pamięci podręcznej. W rezultacie zawartość dysku byłaby uzależniona od kolejności operacji zapisu, co w oczywisty sposób jest niedopuszczalne. Dlatego w Linuxie przyjęto rozwiązanie, w którym do pamiętania aktualnej pozycji w pliku używa się struktury pliku.
Na zakończenie tego rozdziału wypunktujmy kilka spostrzeżeń:

- Katalog od zwykłego pliku odróżnia pole w strukturze dentry lub inode.
- Każdy proces ma swoją tablicę deskryptorów.
- Każdy deskryptor wskazuje na jakąś strukturę pliku (file).
- Każda struktura pliku jest powiązana ze strukturą dentry.
- Na podstawie struktury dentry można wyznaczyć i-węzeł w pamięci (inode).
- Każdy aktywny plik (w tym używany katalog) ma i-węzeł w pamięci.



Rysunek 1.4: Powiązania pomiędzy strukturami danych dotyczącymi plików

- I-węzeł w pamięci tłumaczy się jednoznacznie na i-węzeł na urządzeniu blokowym.

Powiązania między opisanymi strukturami obrazuje rysunek 1.4.

1.3 Sposób wywołania i parametry

Jak wszystkie wywołania systemowe, wywołanie `open()` opiera się na przejściu z Trybu użytkownika do Trybu systemowego. Zrealizowane jest to za pomocą, specjalnie do celu tego zarezerwowanego przerwania 0x80. Użytkownik wywołuje funkcję biblioteczną `open()`, która jako reguła wygenerowana z makr `_syscall` w `<asm-i386/unistd.h>` zapisuje swe argumenty i numer wywołania systemowego w określonych rejestrach i wyzwala przerwanie 0x80. Gdy procedura obsługi przerwania zwróci sterowanie, odczytywane są odpowiednie rejestry z wartości zwrotną i kończone jest wykonywanie funkcji bibliotecznej. Definicje tablicy `_sys_call_table` z numerami wywołań systemowych znajdują się w pliku `arch/i386/kernel/entry.S`. Wpis na piątym miejscu świadczy o tym, iż za realizacje wywołania `open()` odpowiada funkcja

`int sys_open(const char * filename, int flags, int mode)` ,

której kod znajduje się w pliku `<open.c>`.

- `filename` - ścieżka dostępu do pliku; jeżeli zaczyna się od '/' oznacza ścieżkę pełną, jeżeli nie traktowana jest jako kontynuacja ścieżki bieżącej
- `flags` - określają tryb otworzenia pliku, tzn.
  - `O_RDONLY` - otwarcie pliku tylko do czytania,
  - `O_WRONLY` - otwarcie pliku tylko do pisania,
  - `O_RDWR` - otwarcie pliku zarówno do czytania, jak i do pisania.

Flagi te mogą być modyfikowane bitową operacją `OR` przy pomocy następujących flag:

- `O_CREAT` - jeżeli plik nie istnieje, to zostanie utworzony,
- `O_EXCL` - jeżeli plik istnieje i ustawiona jest flaga `O_CREAT`, to zwraca błąd,
- `O_NOCTTY` - jeżeli `pathname` odnosi się do terminala, to nie stanie się on terminalem sterującym, nawet jeżeli takowego proces nie posiada,
- `O_TRUNC` - jeżeli plik już istnieje, to zostanie obcięty,
- `O_APPEND` - tuż po otworzeniu oraz przed każdą operacją pisania wskaźnik pokazujący na aktualną pozycję w pliku będzie ustawiany na końcu pliku, czyli innymi słowy plik zostanie otworzony do dopisywania. Uwaga: system plików NFS nie wspomaga tej flagi,
- `O_NBLOCK / O_NDELAY` - nieblokujące otworzenie pliku, tzn. każda operacja wykorzystująca przydzielony deskryptor do pliku będzie kończyła się błędem jeżeli nie będzie mogła być zrealizowana od razu (np. trzeba będzie czekać na zdjęcie blokady z pliku),
- `O_SYNC` - otworzenie pliku do synchronicznych operacji wejścia-wyjścia, tzn. dowolne operacje pisania będą blokowały proces aż do momentu wykonania fizycznego zapisu na urządzeniu docelowym.

Więcej informacji na temat tych i innych flag można znaleźć np. używając polecenia `man open`.

- `mode` - wykorzystywane jedynie z flagą O_CREAT, wpp. ignorowane; razem z atrybutem umask procesu określa prawa dostępu do pliku `~umask & mode`. Dozwolone są przy tym następujące stałe symboliczne:
  - `S_IRWXU` - 00700 właściciel ma wszystkie prawa: do czytania, pisania i wykonywania,
  - `S_IRUSR (S_IREAD)` - 00400 właściciel ma prawo do czytania,
  - `S_IWUSR (S_IWRITE)` - 00200 właściciel ma prawo do pisania,
  - `S_IXUSR (S_IEXEC)` - 00100 właściciel ma prawo do wykonywania,
  - `S_IRWXG` - 00070 grupa ma wszystkie prawa: do czytania, pisania i wykonywania,
  - `S_IRGRP` - 00040 grupa ma prawo do czytania,
  - `S_IWGRP` - 00020 grupa ma prawo do pisania,
  - `S_IXGRP` - 00010 grupa ma prawo do wykonywania,
  - `S_IRWXO` - 00007 wszyscy mają wszystkie prawa: do czytania, pisania i wykonywania,
  - `S_IROTH` - 00004 wszyscy mają prawo do czytania,
  - `S_IWOTH` - 00002 wszyscy mają prawo do pisania,
  - `S_IXOTH` - 00001 wszyscy mają prawo do wykonywania.

1.4 Zwracane wartości i możliwe błędy

Funkcja `open` zwraca deskryptor pliku w przypadku pomyślnego wykonania operacji otwarcia pliku, w przypadku wystąpienia błędu zaś zwraca -1. Kod błędów można odczytać ze zmiennej `errno`. Oto opis niektórych z mogących się pojawić podczas wykonywania operacji otwierania pliku:

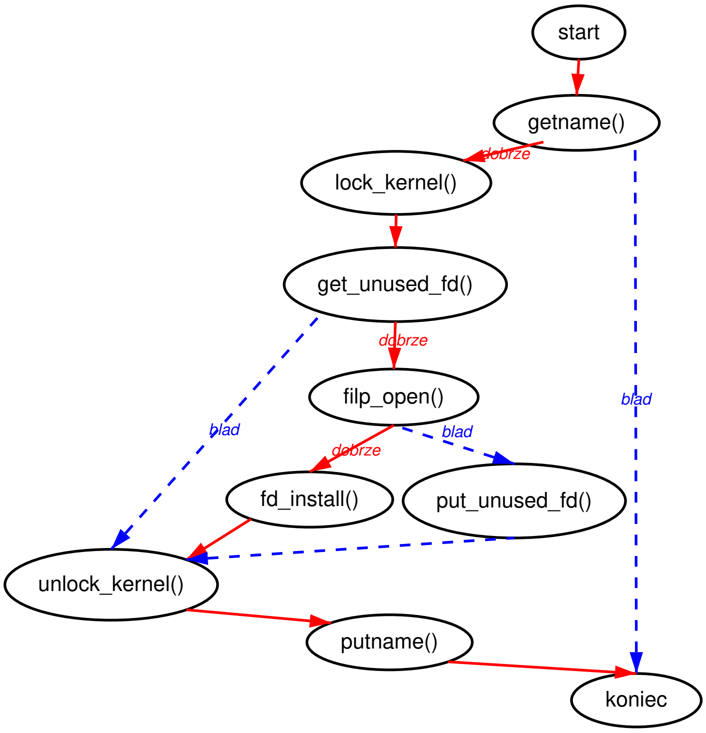
- `EEXIST` - plik podany w `pathname` już istnieje, a ustawione były flagi O_CREAT i O_EXECL,
- `EISDIR` - próbowano otworzyć katalog do pisania,
- `EACCES` - dostęp do pliku był zabroniony lub jeden z katalogów w `pathname` nie miał ustawionego dostępu do przeszukiwania, lub też katalog, w którym plik miał zostać utworzony nie ma prawa do zapisu,
- `ENAMETOOLONG` - ścieżka dostepu `pathname` była zbyt długa,
- `ENOENT` - jeden z katalogów użyty w `pathname` nie istniał lub był nieprawidłowym dowiązaniem,
- `EROFS` - próba otwarcia do pisania pliku, który należy do systemu plików read-only,
- `ENOSPC` - brak wolnego miejsca na urządzeniu do utworzenia nowego pliku,
- `ENOMEM` - niewystarczająca ilość pamięci dostępnej jądru systemu,
- `ENFILE` - osiągnięto maksymalną liczbę otwartych plików w systemie.

Więcej informacji na temat tych i innych kodów błędu można znaleźć np. używając polecenia `man open`.

1.5 Algorytm

We wstępie warto zauważyć, iż jak to zazwyczaj bywa, duża część całości kodu odpowiada za obsługę błędów. Od jądra systemu wymagamy niezawodności, więc wszystkie sytuacje wyjątkowe muszą tu być obsłużone ze szczególną uwagą. Ponieważ nad kodem jądra pracuje wielu programistów i prawdopodobnie każdy z nich ma swoje przyzwyczajenia co do sposobów przekazywania błędów, został narzucony sposób globalny. Opiera się on na spostrzeżeniu, iż wskaźniki jądra mają miejsce na nadmiarową informacje. Dlatego stworzono interfejs składający się z trzech funkcji zdefiniowanych w `<fs.h>` :

- `ERR_PTR(err)` - zamienia kod błędu na wskaźnik
- `IS_ERR(ptr)` - sprawdza czy wskaźnik przenosi informacje o kodzie błędu, czy adres w pamięci
- `PTR_ERR(ptr)` - odczytuje kod błędu zapisany we wskaźniku



Rysunek 1.5: funkcja `sys_open()`

Tablica 1.1: wartości dla sys_open() i filp_open()

00	tylko do odczytu
01	tylko do zapisu
10	do odczytu i zapisu
11	tryb specjalny

Tablica 1.2: wartości dla open_namei() i follow_link()

00	nie potrzeba żadnego pozwolenia
01	potrzebne pozwolenie do odczytu
10	potrzebne pozwolenie do zapisu
11	potrzebne pozwolenia i do odczytu i do zapisu

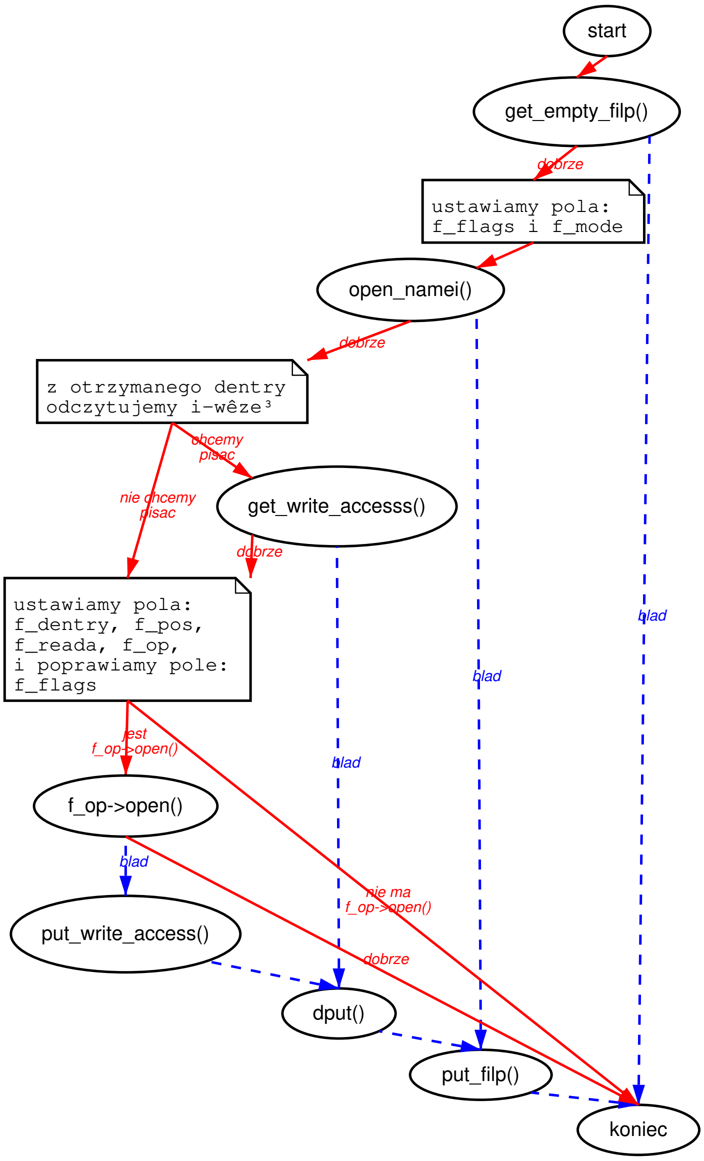
Dzięki ujednoliceniu obsługi błędów za pomocą funkcji definiowanych w oddzielnej bibliotece rozwiązano również ewentualne problemy przy przenoszeniu systemu na inne architekury, z inną przestrzenią adresową.

Pierwszą czynnością algorytmu (patrz rysunek 1.5) jest skopiowanie parametru filename do obszaru pamięci jądra. Odpowiada za to funkcja getname() z <namei.c>. Od teraz mamy pewność, iż nie powstaną żadne sytuacje wyścigowe związane ze "swapowaniem" pamięci procesu zawierającej filename w trakcie wykonywania algorytmu. Przy okazji z powodów praktycznych sprawdzane jest również ograniczenie na długość ścieżki (<PAGE_SIZE). Następnie zawłaszczany jest dostęp do jądra lock_kernel() i za pomocą funkcji get_unused_fd() z <open.c> rezerwowany jest pierwszy nieużywany deskryptor w tablicy current->files->fd[]. Jeżli w tablicy nie ma już miejsca, a nie przekroczyliśmy jeszcze limitu otwartych plików (current->rlim[RLIMIT_NOFILE].rlim_cur), tablica się rozrośnie. Najważniejsza część algorytmu wykonuje się w funkcji filp_open() (opisanej dalej), która zwraca wskaźnik do struct file odpowiadającej otwieranemu plikowi. Jeżeli wszystko się powiodło to struktura podczepiana jest pod wcześniej zarezerwowany deskryptor fd_install(), oddawany jest dostęp do jądra unlock_kernel() i zwalniana pamięć zarezerwowana na filename.

```
struct file *filp_open(const char * filename, int flags, int mode) ,
```

Funkcja ta zdefiniowana jest w pliku <open.c>, a jej parametry są analogiczne do parametrów sys_open(). Należy jednak pamiętać, iż sys_open() i filp_open() rozpoznają dwa młodsze bity parametru flags zgodnie z Tablicą 1.1, a funkcje z niższych warstw takie jak open_namei() i follow_link() używają, nie tylko według autorów jądra, bardziej zdroworozsądkowych wartości (patrz Tablica 1.2)

Na początek znajdowana jest wolna struct file lub jeżeli nie przekroczono jeszcze limitu max_files z <file_table.c> alokowana jest nowa. Root ma tutaj specjalne przywileje. Zarezerwowana jest dla niego dodatkowa struktura na wypadek zakorkowania się systemu i potrzeby podjęcia działań prewencyjnych. Następnie wypełnia się pola f_flags i f_mode i za pomocą opisanej dalej funkcji open_namei() zdefiniowanej w <namei.c> przygotowywana jest struct dentry i przeprowadzane właściwe otwieranie pliku. Teraz wystarczy jedynie, w razie potrzeby,



Rysunek 1.6: funkcja `sys_open()`

otrzymać prawo do pisania get_write_access() z <namei.c> (mogło się zdarzyć, że była blokada VM_DENYWRITE) dla dentry->d_inode, wypełnić pozostałe pola : f_dentry, f_pos, f_reada, f_op i f_flags oraz wywołać funkcję file->open() (operacja ta zajmuje się czynnościami właściwymi dla danego typu pliku). Jeśli plik, który zosał otwarty, jest plikiem urządzenia znakowego, wywoływana jest funkcja chardev_open(), która z koleji modyfikuje operacje plikowe zgodnie z numerem nadrzędnym (ang. *major*) urządzenia.

```
int chrdev_open(struct inode * inode, struct file * filp)
{
  int i;
  i = MAJOR(inode->i_rdev);
  if (i >= MAX_CHRDEV || !chrdevs[i].fops)
    return -ENODEV;
  filp->f_op = chrdevs[i].fops;
  if (filp->f_op->open)
    return filp->f_op->open(inode,filp);
  return 0;
}
```

Operacje plikowe dla programów obsługi urządzeń przechowywane są w tablicy chrdevs[], gdzie zostały umieszczone przez fukcję register_chrdev() (patrz [1] rodział 7) podczas inicjowania programu obsługi. Funkcja open() programu obsługi urządzenia zapewnia dodanie dalszych operacji plikowych zgodnie z numerem podrzędnym (ang. *minor*) urządzenia.

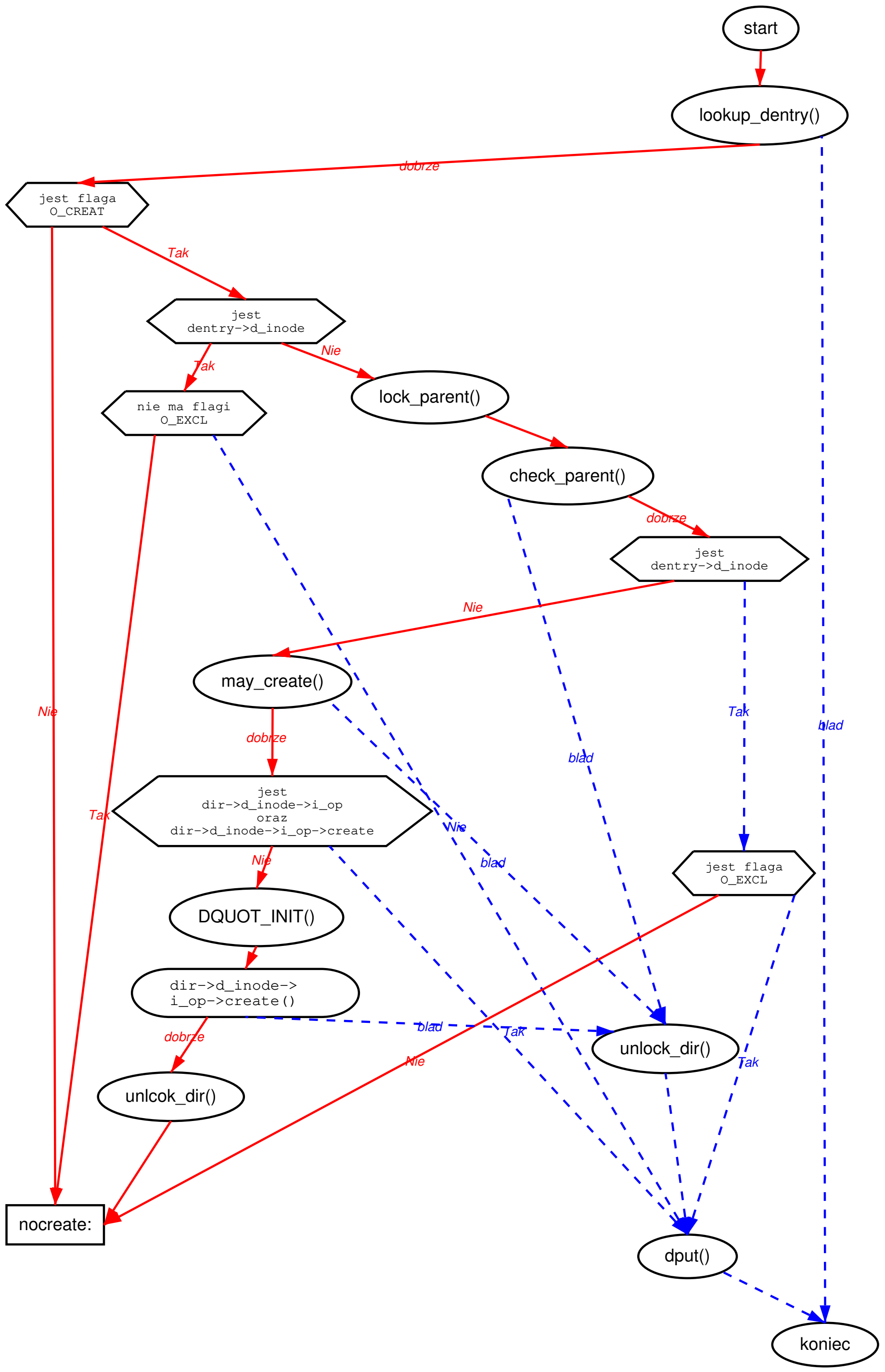
struct dentry *open_namei(const char * pathname, int flag, int mode) , Największa część testów i pracy wykonuje się w funkcji open_namei() (patrz rys. 1.7 i 1.8). Po pierwsze na podstawie scieżki odnajdywana jest struktura dentry katalogu, w którym otwieramy plik. Odpowiada za to funkcja lookup_dentry() z <namei.c>. Przechodzi ona po kolei wszystkie katalogi ze scieżki, zaczynając od current->fs->root lub current->fs->pwd, w zależności od tego czy pathname zaczynało się od '/' czy nie. Jeżli po drodze wystąpiły jakieś linki rozwija się je za pomocą do_follow_link(). Oczywiście proces żądający takiej operacji musi mieć prawa do wykonywania dla wszystkich odwiedzanych katalogów.

Ze względu na to, że zarówno lookup_dentry() jak i do_follow_link() korzystają z funkcji dget() typ plik (normalny, plik urządzenia itp.) jest już od tego momentu znany. Z tego względu operacje na strukturach dentry zawierają w polu default_file_ops operacje plikowe odpowiednie dla danego typu pliku.

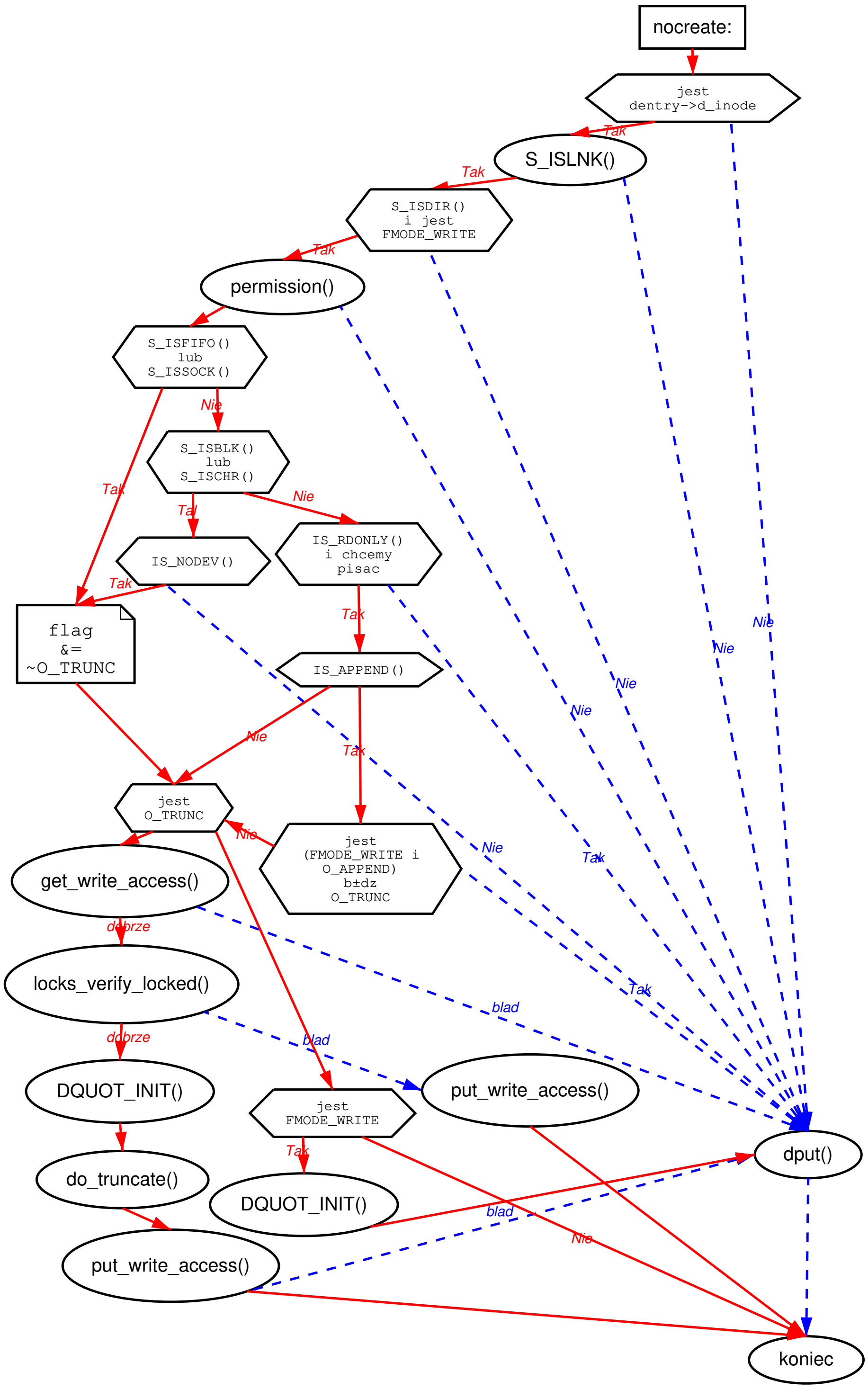
Teraz jeżeli była flaga O_CREAT i pozwala na to O_EXCL musimy najpierw plik utworzyć. Żeby modyfikować katalog (tworzyć w nim nowy plik) trzeba zabezpieczyć dostęp do jego katalogu nadrzędnego lock_parent() z <namei.c>. Jeżeli w międzyczasie nikt naszego katalogu nie usunął i nie utworzył pliku o tej samej nazwie, możemy wykonać dir->d_inode->i_op->create(), oczywiście przy założeniu, że mamy do tego wszystkie prawa may_create().

Gdy jesteśmy już pewni, że plik istnieje sprawdzamy, czy nie jest linkiem symbolicznym bądź katalogiem (jeżeli chcemy pisać) i wykonujemy następujące testy :

- Sprawdzamy prawa dostępu do struktury dentry. Wykorzystywana jest do tego funkcja permission(), która wykonuje swe zadanie na podstawie flag z (Tablicy 1.2). Jeśli w implementacji systemu plików zdefiniowana jest operacja permission() używana jest właśnie ona.



Rysunek 1.7: funkcja `open_namei()` część pierwsza (do `nocreat:`)



Rysunek 1.8: funkcja `open_namei()` część druga (od `nocreat:`)

- Jeżeli jest to plik urządzenia, do którego dostęp został zbroniony poprzez zamontowanie z opcją MS_NODEV, zostanie zwrócony błąd EACCES.
- Jeżeli plik nie jest plikiem urządzenia, a proces usiłuje uzyskać prawa zapisu do systemu plików przeznaczonego tylko do odczytu, próba ta kończy się niepowodzeniem i zwracany jest błąd EROFS.
- Jeżeli proces próbuje uzyskać prawa zapisu do pliku przeznaczonego "tylko do dopisywania", a nie jest aktywna flaga O_APPEND, funkcja open_namei() przerywana jest z błędem EPERM.
- Jeżeli aktywna jest flaga O_TRUNC, funkcja open_namei() wywołuje funkcję get_write_access() (o której była mowa wcześniej), aby sprawdzić, czy plik nie został zamapowany z włączoną flagą VM_DENYWRITE. Za pomocą funkcji do_truncate() rozmiar pliku ustawiany jest na zero i zmniejszana jest liczba pisarzy dla pliku (put_write_access()).

Gdy wszystkie powyższe testy się powiodły posiadamy już gotową strukturę dentry, którą możemy zwrócić jako wynik działania funkcji open_namei().

Rozdział 2

Funkcja creat

Ponieważ zgodnie z założeniami projektantów warstwa "Interfejsu do systemu plików" miała umożliwiać wykonywanie wszystkich podstawowych operacji na plikach, została przewidziana funkcja `creat()`, która ma umożliwiać utworzenie nowego pliku. Jednak ze względu na ilość możliwych flag dla funkcji `open()`, okazuje się, iż ten sam efekt można osiągnąć wywołując `open()` z flagą `O_CREAT | O_WRONLY | O_TRUNC`. Tak właśnie dzieje się to Linuxie 2.2.17. `creat()` pozostało ponieważ tworzenie, niezależnie od tego czy można je osiągnąć w jakiś inny sposób, jest operacją elementarną na plikach.

2.1 Sposób wywołania i parametry

(Patrz rozdział 1.3)

2.2 Zwracane wartości i możliwe błędy

(Patrz rozdział 1.4)

Rozdział 3

Zadania

1. Zadanie 1
   Podać sytuacje, w których większe są od jedności:
   - licznik odwołań `count` w strukturze `files_struct`,
   - licznik odwołań `f_count` w strukturze `file`,
   - licznik odwołań `d_count` w strukturze `dentry`.
2. Zadania 2
   Opisz sytuacje, w których :
   - Nie udało się nam otworzyć pliku do pisania, gdy mamy wszystkie wymagane do tego uprawnienia i innemu procesowi z takimi samymi uprawnieniami operacja ta się powiodła?
   - Chcąc utworzyć plik otrzymaliśmy jego `struct dentry`, zabezpieczyliśmy dostęp do katalogu nadrzędnego, mieliśmy wszystkie wymagane uprawnienia, a sys_open() zakończyło się błędem.
   - Jeden użytkownik próbując otworzyć nowy plik otrzymał błąd "za dużo otwartych plików w systemie", a innemu użytkownikowi taka sama operacja się powiodła mimo, iż sytuacja w systemie nie uległa zmianie.
3. Zadania 3
   Opisać pola: `max_fds, next_fd, fd` w strukturze `files_struct` dla pewnego procesu po wykonaniu następującego ciągu operacji:
   - otworzenie pliku: plik1,
   - otworzenie pliku: plik2,
   - ...,
   - otworzenie pliku: plik36,
   - zamknięcie pliku: plik36,
   - zamknięcie pliku: plik30,
   - zamknięcie pliku: plik20,

- zamknięcie pliku: plik10,
- zamknięcie pliku: plik1.

Zakładamy, że przed rozpoczęciem tych operacji zajęte były deskryptory 0, 1, 2 oraz 5 i 6, zaś `max_fds` = 32.
Podać ciąg operacji, po wykonaniu których otrzymane zostaną dane wejściowe do tego zadania.
Co ulegnie zmianie jeżeli zamiast plików plik1, plik2, ... , plik36 użyjemy tylko jednego z nich?

Rozdział 4

Rozwiązania zadań

1. Rozwiązanie zadania 1
   Licznik odwołań `count` w strukturze `files_struct` zlicza ile procesów wykorzystuje tę strukturę, zatem wartość ta będzie większa od jedynki w przypadku, gdy struktura ta jest współdzielona przez kilka procesów (patrz np. polecenie `fork`).
   Wywołanie polecenia `dup` może spowodować zwiększenie licznika odwołań `f_count` w strukturze `file`.
   Do zwiększenia wartości licznika odwołań `d_count` w strukturze `dentry` wystarczy by plik, który jest opisywany przez tę strukturę, został otworzony kilkakrotnie. Nie ma przy tym znaczenia, czy operacja otwierania wykonana została przez różne, czy przez ten sam proces.

2. Rozwiązanie zadania 2

   - Sytuacja tak może mieć miejsce jeżeli ten drugi proces zamapował na pliku blokadę `VM_DENYWRITE`.
   - Jeżeli w trakcie starania się o zabezpieczenie dostępu do katalogu nadrzędnego ktoś usunął nasz plik otrzymamy błąd ENOENT (chyba nie jest to właściwy kod błędu, autorów jądra też niepokoi ta sytuacja).
   - Drugi użytkownik mógł być rootem i jako taki ma zarezerwowane dodatkowe zasoby na podjęcie działań prewencyjnych, w tym dodatkową `struct file`.

3. Rozwiązanie zadania 3
   Jeżeli na początku proces miał otworzone standardowe wejście (deskryptor 0), wyjście (deskryptor 1) oraz strumień błędu (deskryptor 2), `max_fds` $= 32$, `next_fd` $= 3$, to wystarczy, że proces otworzył czterokrotnie jakiś plik. Wtedy zajęte są deskryptory od 0 do 6, a `next_fd` $= 7$. Po zwolnieniu deskryptora numer 4, a następnie deskryptora numer 3 dostaniemy dane wejściowe do zadania. Oczywiście `next_fd` $= 3$, gdyż wszystkie deskryptory o mniejszych numerach muszą być zajęte.
   Po wykonaniu wszystkich operacji otwarcia 36 plików zajęte są deskryptory od 0 do 40, przy czym rozmiar tablicy fd (`max_fds`) został powiększony do 256, zaś `next_fd` $= 41$. Po wykonaniu następnych operacji otrzymujemy kolejno:

   zwolniony deskryptor numer 34, `next_fd` $= 34$, zwolniony deskryptor numer 24, `next_fd` $= 24$,

zwolniony deskryptor numer 14, `next_fd` $= 14$,

zwolniony deskryptor numer 3, `next_fd` $= 3$.

W zadaniu nie ma znaczenia czy przy wykonywanych operacjach używaliśmy tego samego, czy różnych plików.

Literatura

[1] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner: *Linux Kerner. Jądro systemu*, MIKOM Wydanie II (2000)

[2] A. Silberschatz, P. Galvin *Podstawy systemów operacynych*, WNT Wydanie III (2000)

[3] M. Bach *Budowa sytemu operacujnego Unix*, WNT ('95)

[4] Praca zbiorowa pod redakcją J. Mincer-Daszkiewicz *System operacyjny Linux (Wersja 2.2.12) - Podręcznik* (1999-2000)

[5] Praca zbiorowa pod redakcją J. Mincer-Daszkiewicz *System operacyjny Linux (Wersja 2.0.32) - Prezentacje* ('98-'99)

[6] *Linux Programers Manual*