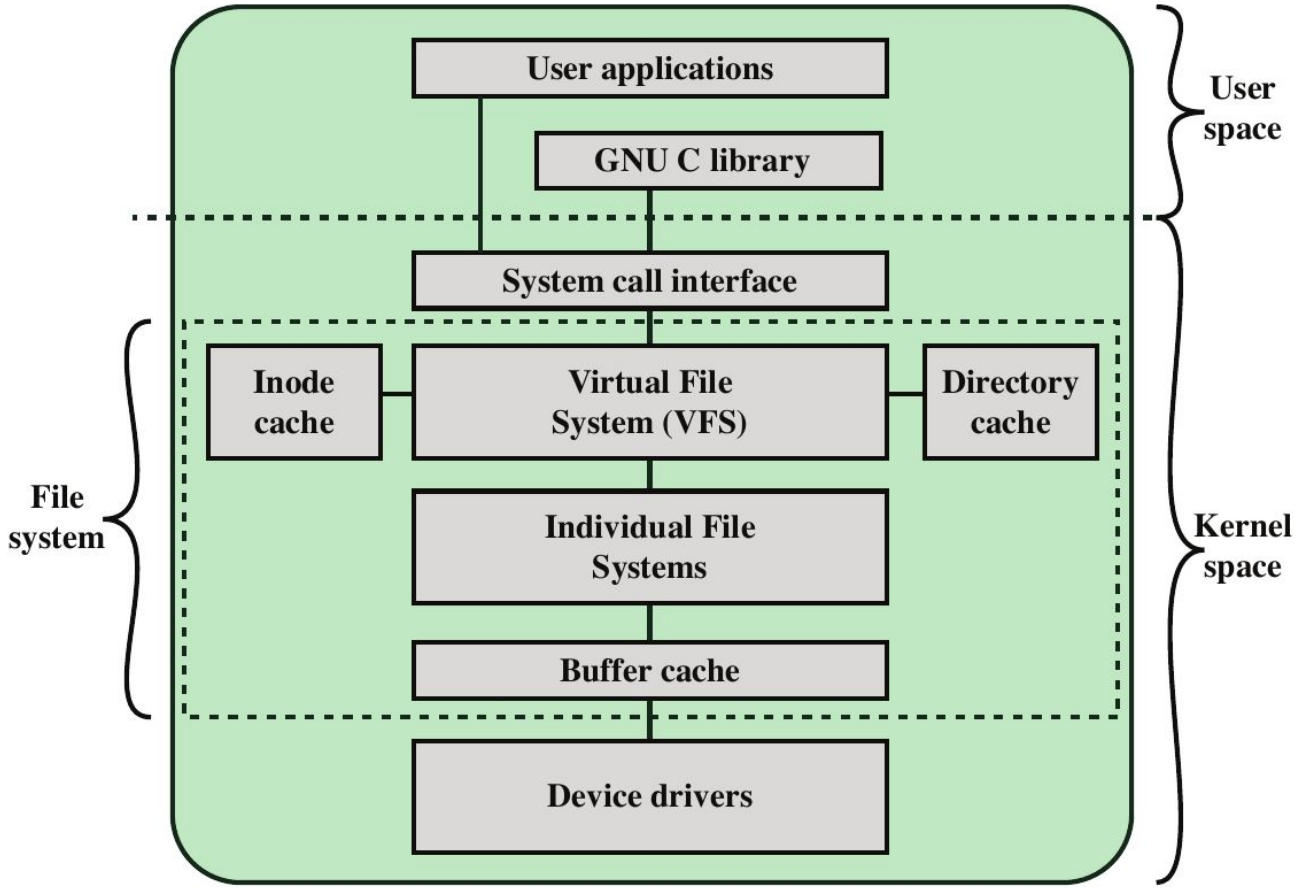
Systemy operacyjne

Wykład 11: Pliki i katalogi



Dziś będziemy mówić o korzystaniu z plików w przestrzeni użytkownika, o tym jakie usługi jądra wystawia na zewnątrz i nieoczywistych interakcjach między programami, a jądrem.

Typ pliku vs. format pliku

Typ pliku to rodzaj zasobu jądra, które zostało udostępnione użytkownikowi przez interfejs interakcji z plikiem.

Format pliku specyfikuje wewnętrzną strukturę pliku → film, obraz, dokument tekstowy. Nazewnictwo rozszerzeń plików (**`.mkv`**, **`.png`**, **`.docx`**) to tylko konwencja. Prawdziwy format pliku należy wykryć na podstawie **magicznych bajtów** (ang. *magic number*)!

Prezentacja: `file`

Wczytuje bazę danych `/usr/lib/file/magic.mgc` i używa algorytmów dopasowania wzorca do wyznaczenia formatu.

Dostęp do pliku

Po otwarciu pliku otrzymujemy **uchwyt** (ang. *handle*) do zasobu.

dostęp sekwencyjny operacja dostępu ma charakterystykę temporalną, po wykonaniu operacji nie możemy cofnąć się w czasie np. pakiety sieciowe, drukarka, potok

dostęp swobodny plik to ciąg danych w pamięci; posiada rozmiar, który możemy zmieniać; operacje odczytu i zapisu zmieniają pozycję **kursora**, możemy przesunąć kursor w dowolne miejsce

Q: Skoro dwa procesy mogą współdzielić plik, to czy współdzielą kursor?
A: W pewnych przypadkach tak. Zależy jak uchwyty mapują się na zasoby.

Typy plików

- **plik zwykły** → ciąg bajtów o **swobodnym dostępie**
- **katalog** → ciąg rekordów opisujących zawartość katalogu
- **potok** → ciąg bajtów o **dostępie sekwencyjnym** zawierający dane wysłane z innego procesu
- **urządzenie znakowe** → ciąg bajtów o dostępie sekwencyjnym odpowiadający urządzeniu (np. port szeregowy, drukarka, …)
- **urządzenie blokowe** → ciąg bajtów o swobodnym dostępie, najefektywniejszy dostęp blokowy
- **gniazdo domeny uniksowej** → jak potok, ale dwukierunkowe
- **dowiązanie symboliczne** → “wskaźnik” na inny plik

Atrybuty pliku

Nazwa pliku w danych **katalogu**! Atrybuty w **metadanych pliku**, a konkretniej w **i-węźle** (ang. *i-node*) systemu plików.

Podstawowe atrybuty plików: rozmiar (bajty), właściciel, grupa, uprawnienia dostępu, czasy utworzenia / modyfikacji / dostępu.

Prezentacja: `stat`

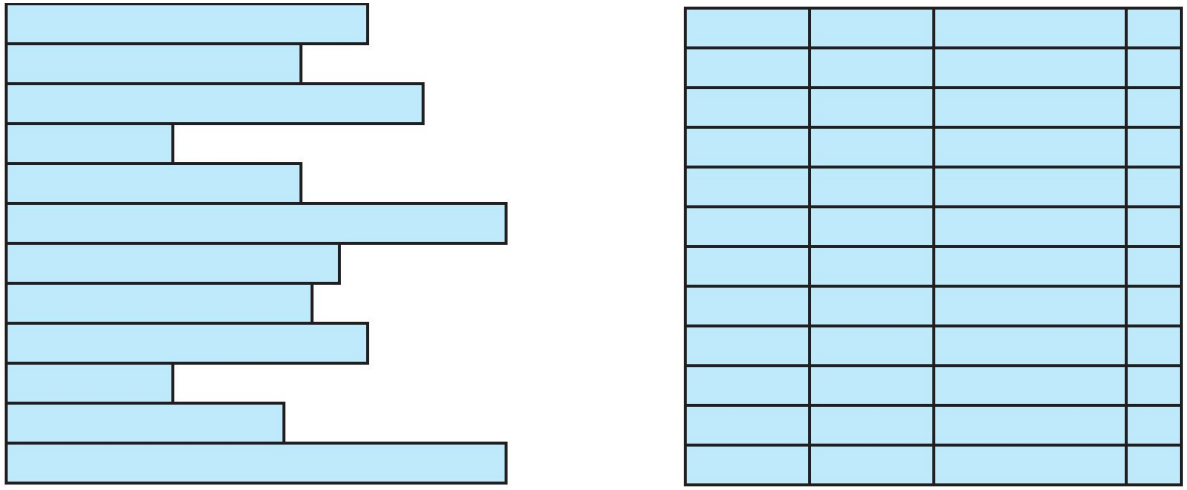
Rozszerzone atrybuty plików (xattr) pary klucz-wartość trzymane w dodatkowych danych pliku.

```
$ getfattr --dump --absolute 10.1.1.32.3491.ps
user.xdg.origin.url="http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.
1.1.32.3491&rep=rep1&type=ps"
user.xdg.referrer.url="https://www.google.pl/"
```

Organizacja plików (1)

Jądro traktuje pliki jako zbiór rekordów, albo ciąg bajtów (Multics).

sterta dopisujemy do pliku **rekordy** (różny zestaw pól i długość)
p. sekwencyjny ustalone rekordy, uporządkowany **kluczem**

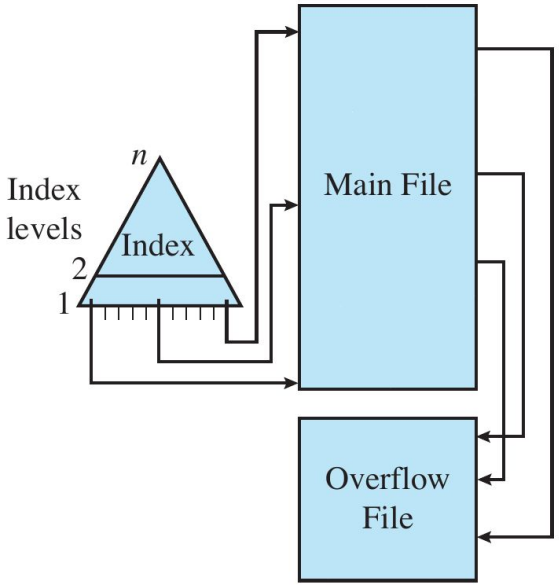


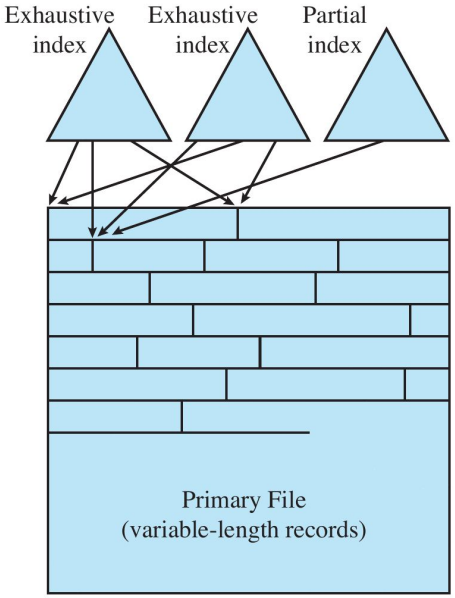
Jaki koszt operacji wstawiania, wyszukiwania i usuwania? Czy rekordy odpowiadają blokom dyskowym? Czy trzeba **kompaktować**?

Organizacja plików (2)

p. indeksowany sekwencyjny indeks przyspiesza dostęp losowy, dodawanie z zachowaniem porządku wolne, nowe rekordy zapisujemy w dzienniku i periodycznie łączymy z głównym plikiem

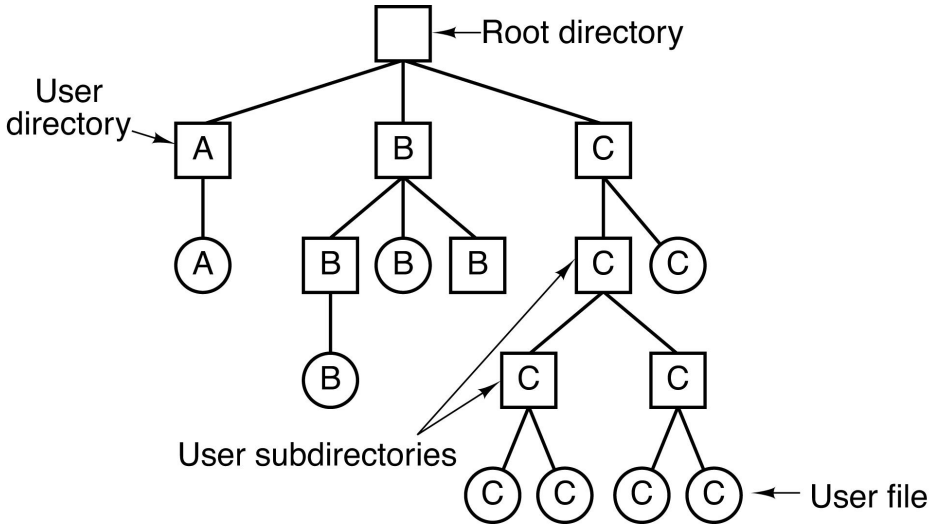
p. indeksowany dostęp do rekordów tylko przez indeksy





Hierarchiczna struktura katalogów

W uniksach globalna przestrzeń nazw z wszystkimi zasobami plikowymi. W węzłach katalogi, a w liściach pozostałe zasoby. Katalog może zostać przesłonięty drzewem wybranego systemu plików → **punkt montażowy**.



Prezentacja: `mount`, `cat /proc/filesystems`

`WinNT` nie ma punktów montażowych, mamy las drzew (dyski `C:`, `D:`, `...`) – jak w `MS-DOS`, ale w `VMS` tak nie było! (Dave Cutler, `RSX-11` → `VMS` → `WinNT`)
Dyski wewnętrznie odwzorowane na drzewo wszystkich zasobów jądra.

Ścieżki i rozwiązywanie nazw

Rozwiązywanie nazw to proces odwzorowania **ścieżki** na zasób!
Ścieżka składa się z **komponentów** i **znaków separatora** “**/**”.

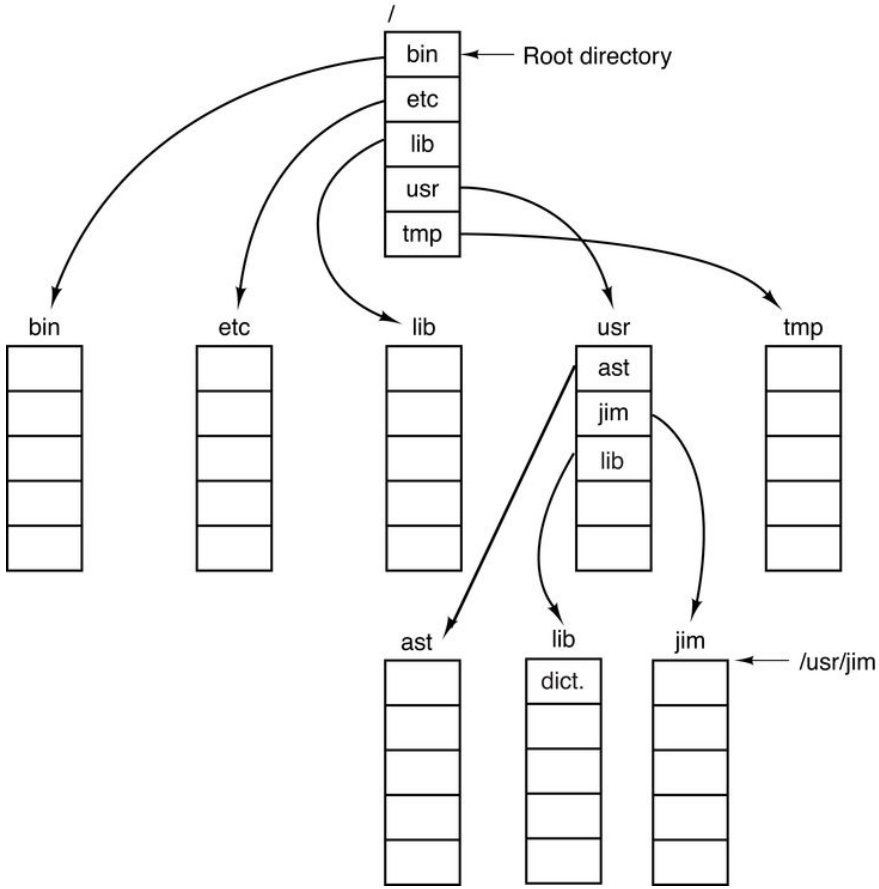
Ścieżka relatywna nie zaczyna się w katalogu głównym.
Odwzorowanie zależne od katalogu roboczego procesu (chdir, getcwd).

Ścieżka absolutna zaczyna się w katalogu głównym, jeśli nie zawiera “`.`”, “`..`” i dowiązań symbolicznych to jest dodatkowo **znormalizowana**.

Rozwiązywanie nazw (`namei`) należy do zadań **wirtualnego systemu plików** (ang. *Virtual File System*). Ścieżka odwzorowujem na **v-węzeł**, czyli reprezentację **i-węzła** w pamięci niezależną od systemu plików.
Proces jest kosztowny, więc wprowadzono pamięć podręczną `namecache`.
VFS sprawdza także **uprawnienia dostępu** na podstawie ucred.

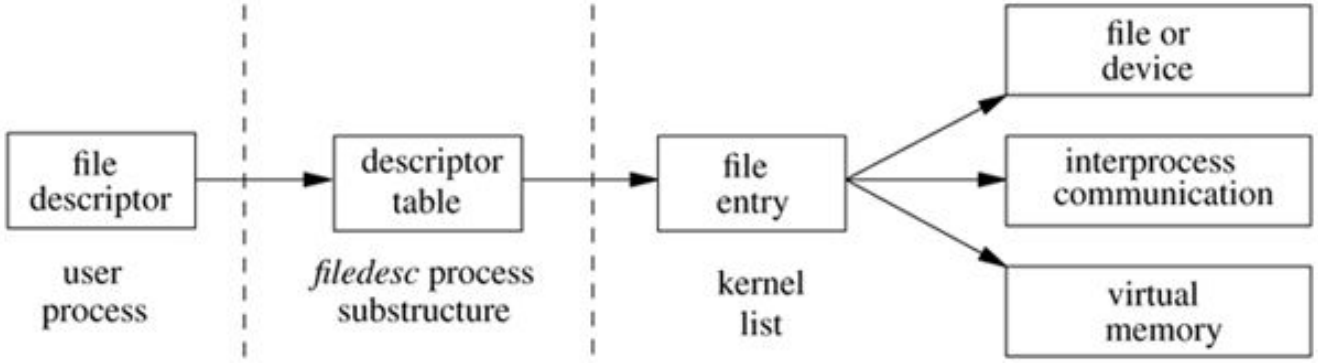
Unix: struktura katalogów (FHS)

bin podstawowe programy
boot jądro i bootloader
dev pliki urządzeń
etc pliki ustawień
home kat. użytkowników
lib biblioteki i sterowniki
mnt punkty montażowe
proc info o procesach
root katalog admina
usr drugorzędna hierarchia
var pliki często zmieniane



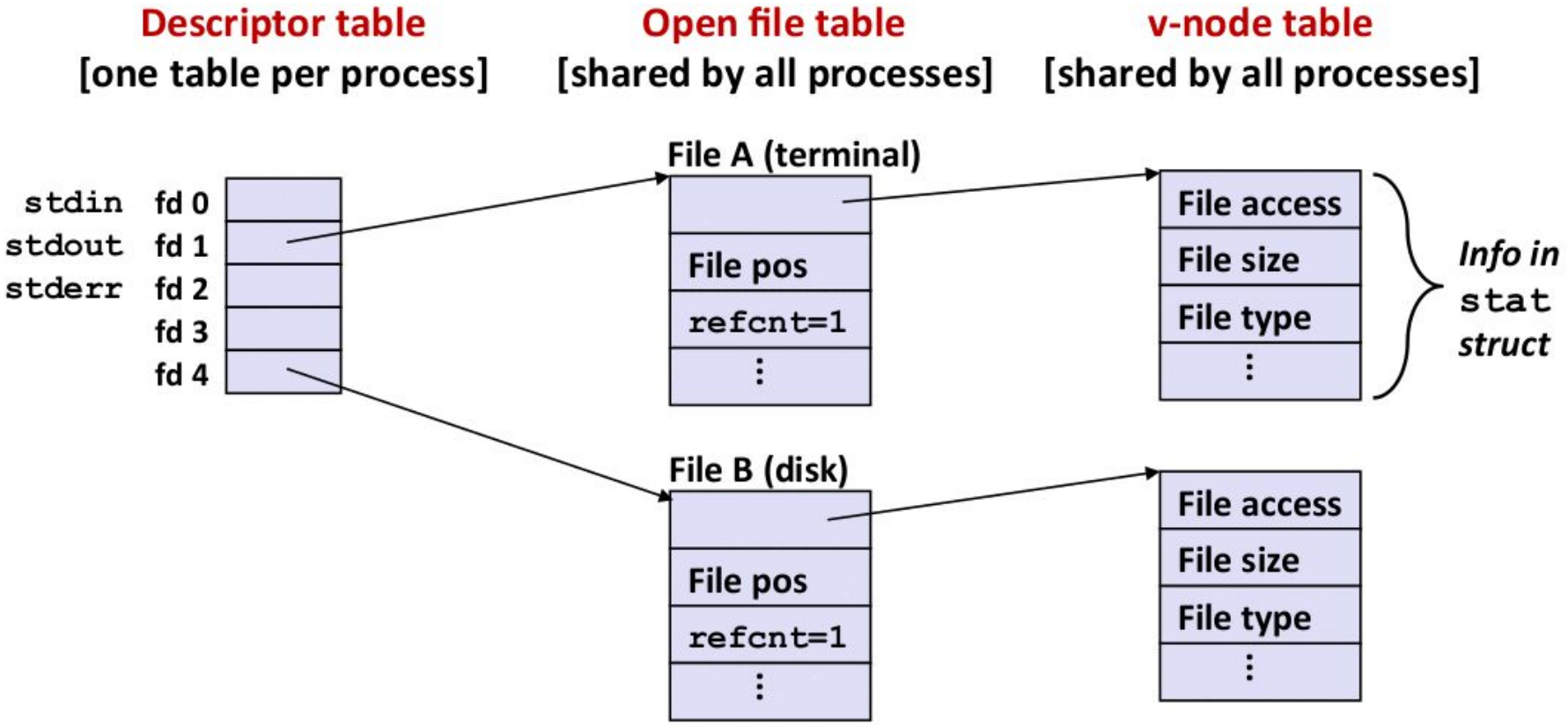
Unix: wywołania systemowe

Każdy otwarty zasób plikowy ma **uchwyt** (ang. handle) zwany **deskryptorem pliku** (ang. file descriptor) czyli liczbę całkowitą ≥ 0.



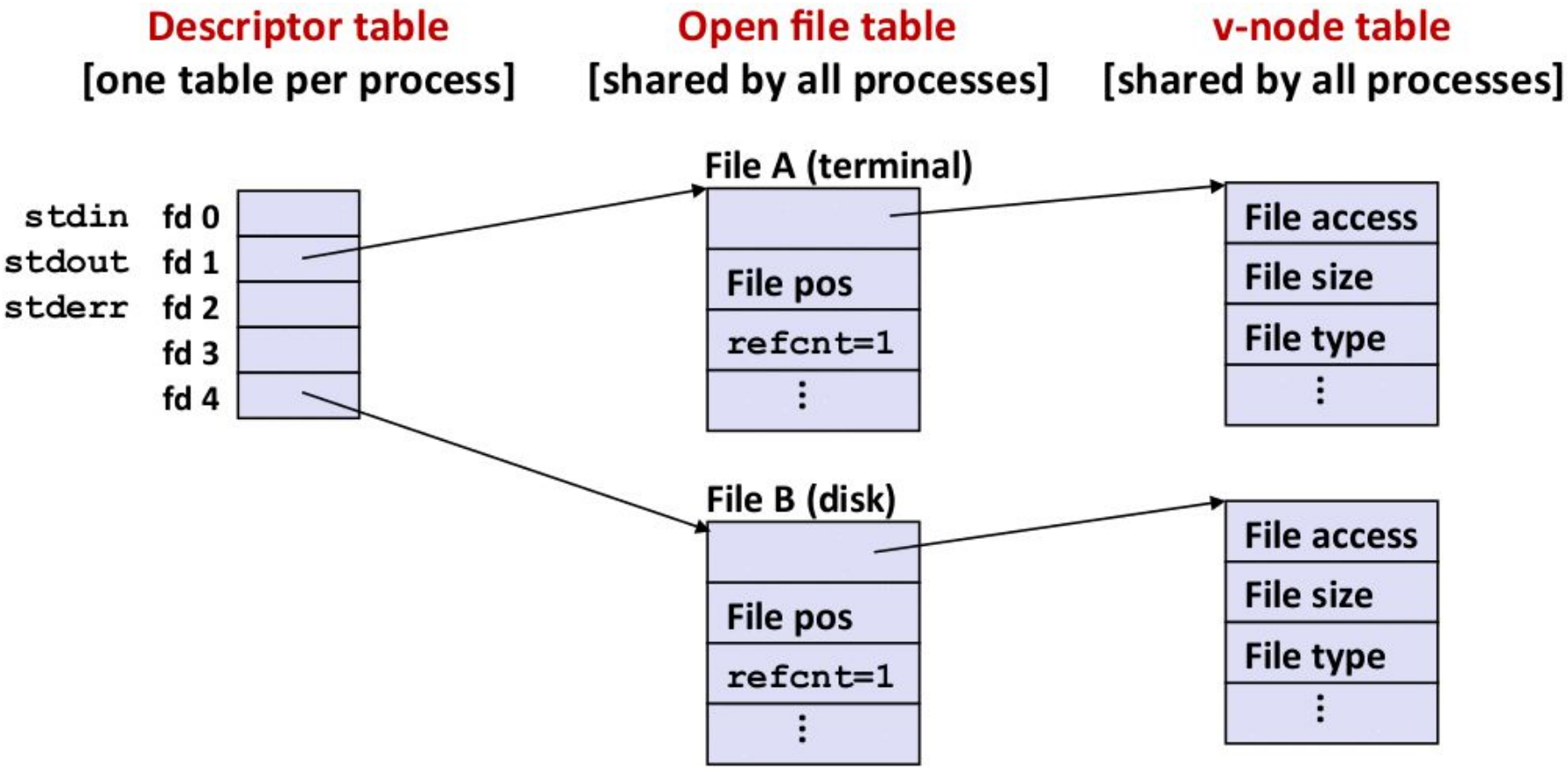
Implementacja wywołania systemowego odnajduje wpis w **tablicy deskryptorów plików** (filedesc) i skojarzony z nim **wpis pliku** (file). Każdy proces ma prywatną tablicę deskryptorów, ale wpisy mogą być współdzielone. Typu pliku (`DTYPE_VNODE`, `DTYPE_SOCKET`, `DTYPE_SHM`) determinuje zestaw implementacji **operacji na pliku** (fileops).

Reprezentacja plików w jądrze



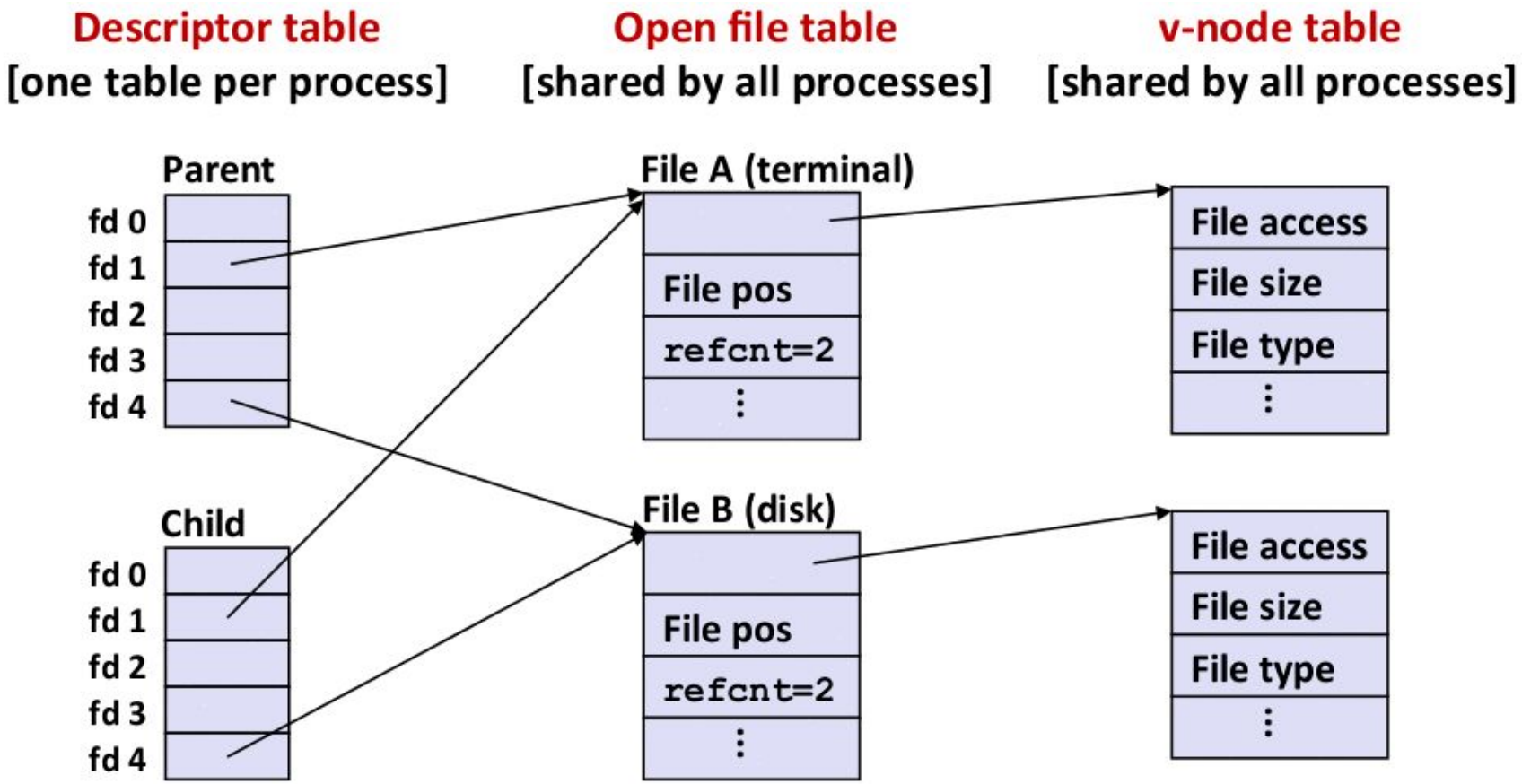
Każdy wpis pliku posiada **kursor** pliku, licznik referencji oraz odwołanie do obiektu wspierającego (potok vs. plik zwykły).

Współdzielenie plików (1)



Przed wywołaniem `fork()`.

Współdzielenie plików (2)



Po wywołaniu `fork()`.

Unix: operacje na plikach

```
int open(const char *path, int flags);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
off_t lseek(int fd, off_t offset, int whence);
int truncate(const char *path, off_t length);
int posix_fallocate(int fd, off_t offset, off_t len);
int posix_fadvise(int fd, off_t offset, off_t len, int advice);
```

Flagi otwarcia pliku

O_RDONLY, **O_WRONLY**, **O_RDWR** tryb dostępu do zawartości

O_CLOEXEC deskryptor automatycznie zamykany przy wywołaniu **exec**

O_APPEND zapis atomowo przesuwa kursor na koniec i dopisuje

O_SYNC, **O_DSYNC** integralność danych i metadanych przy zapisie

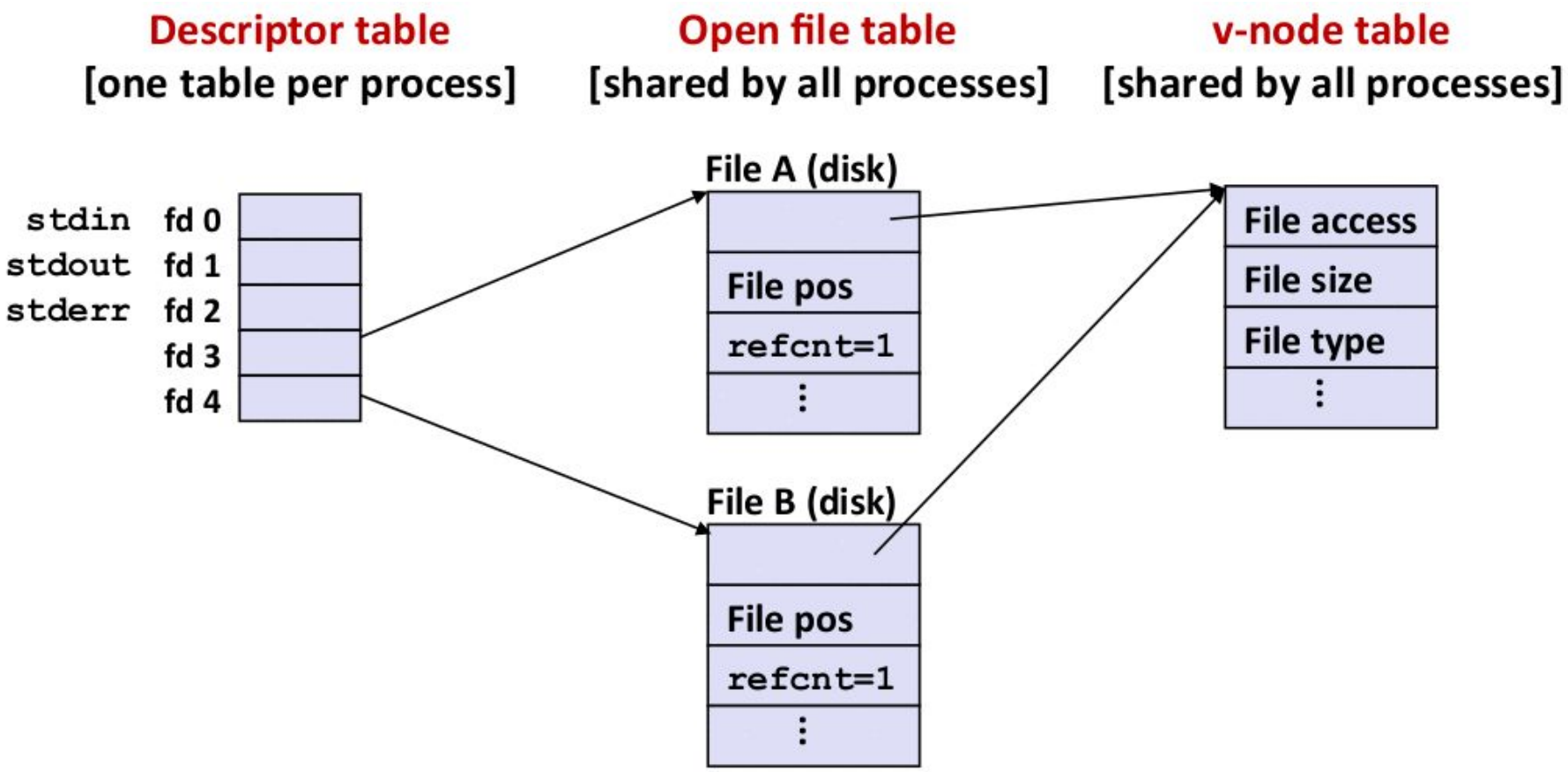
O_DIRECT pomija kopiowanie danych do jądra, operacje zawsze blokami

O_CREAT|O_EXCL utwórz plik pod warunkiem, że nie istnieje

O_DIRECTORY otwórz plik pod warunkiem, że jest katalogiem

O_NONBLOCK otwórz w trybie nieblokującym (~~pliki regularne~~)

Wielokrotne otwarcie pliku



Rozłączne kursory, ale obiekt wspierający ten sam.

Czytanie z terminala lub gniazda

```
ssize_t readn(int fd, void *buf, size_t n) {
  size_t nleft = n;
  while (nleft > 0) {
    ssize_t nread = read(fd, buf, nleft);
    if (nread < 0) {
      if (errno != EINTR) /* interrupted by sig handler return? */
        return -1;        /* no => return -1, errno set by read() */
      nread = 0;          /* yes => call read() again */
    }
    if (nread == 0)       /* EOF encountered ? */
      break;
    nleft -= nread;
    buf += nread;
  }
  return n - nleft;       /* return >= 0 */
}
```

Dowiązania symboliczne

```
int symlink(const char *target, const char *linkpath);
```

```
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

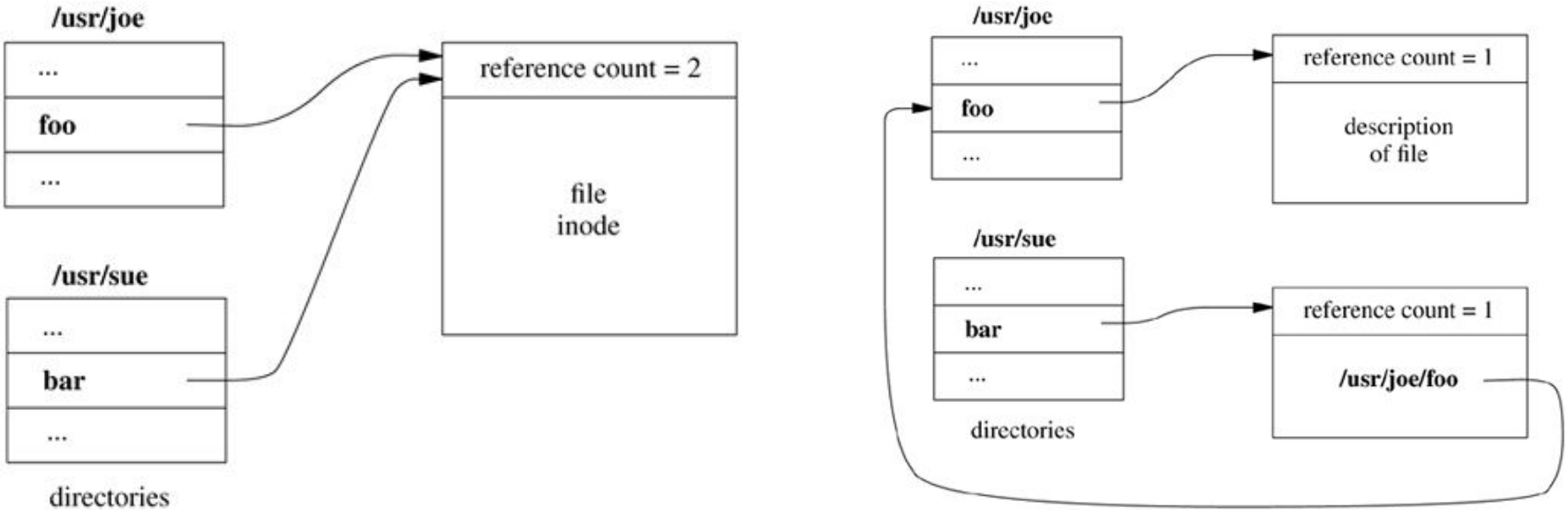
Dowiązania symboliczne (ang. *symbolic links*) specjalny typ pliku, który w zawartości przechowuje ścieżkę do innego pliku. System nie sprawdza poprawności tej ścieżki → może powstać pętla.

Działa jak słaba referencja → plik docelowy może przestać istnieć, system dopuszcza **wiszące dowiązania** (ang. *dangling symlinks*).

Dereferencja dowiązania jest przezroczysta. Nie wykonujemy operacji na pliku dowiązania tylko na tym na co wskazuje. Zawsze?

Problem na poziomie API! Jak pobrać właściwość dowiązania zamiast pliku docelowego? Funkcje z prefiksem `l`, np. `lstat`.

Dowiązanie symboliczne vs. twarde



Dowiązania twarde to wskaźniki na i-węzły (licznik referencji!) plików → różne nazwy tego samego pliku w obrębie jednego systemu plików.

Dowiązania symboliczne kodują ścieżkę do której należy przekierować algorytm rozwiązywania nazw.

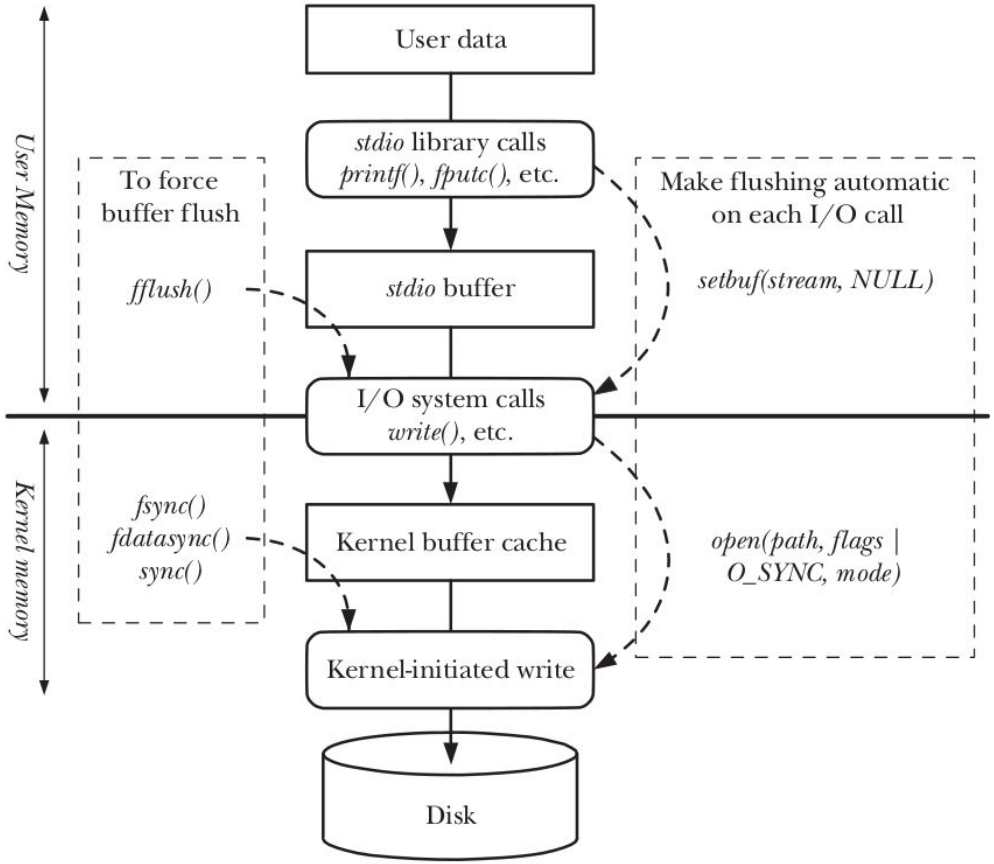
Buforowanie plików

Przestrzeń użytkownika → `setvbuf`, `fflush`.

Przestrzeń jądra:

- kopiowanie danych do jądra (uio)
- bufory systemu plików (page / buffer cache)
- bufory dysku

To kiedy mamy gwarancję, że dane zostały zapisane na dysk?



Buforowanie danych i metadanych

Program wraca z **`sys_write`**. Czy dane są już na dysku? Nie!

`sync` synchronizuje wszystkie bufory jądra z pamięcią drugorzędną

`fsync` (**`O_SYNC`**) synchronizuje dane i metadane wybranego pliku

`fdatasync` (**`O_DSYNC`**) synchronizuje tylko dane pliku

Jaka jest różnica? Dopisujemy na koniec pliku – dane zostały wypisane na dysk, a rozmiar pliku nie, bo jest w metadanych!

Q: Czy zawsze chcemy synchronizować jednocześnie dane i metadane?
A: Czas ostatniego dostępu (ang. *access time*) pewnie nie, szczególnie dla dysków półprzewodnikowych.

Unix: operacje na metadanych plików

```
int stat(const char *path, struct stat *buf);
```

```
int utimes(const char *path, const struct timeval times[2]);
```

```
int chown(const char *path, uid_t owner, gid_t group);
```

```
int chmod(const char *path, mode_t mode);
```

Powyższe wywołania systemowe odczytują i modyfikują dane **i-węzła**!

```
struct stat {
  dev_t     st_dev;         /* ID of device containing file */
  ino_t     st_ino;         /* Inode number */
  mode_t    st_mode;        /* File type and mode */
  nlink_t   st_nlink;       /* Number of hard links */
  uid_t     st_uid;         /* User ID of owner */
  gid_t     st_gid;         /* Group ID of owner */
  dev_t     st_rdev;        /* Device ID (if special file) */
  off_t     st_size;        /* Total size, in bytes */
  blksize_t st_blksize;     /* Block size for filesystem I/O */
  blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
  struct timespec st_atim;  /* Time of last access */
  struct timespec st_mtim;  /* Time of last modification */
  struct timespec st_ctim;  /* Time of last status change */
};
```

Czytanie metadanych pliku

```c
int main(int argc, char *argv[]) {
  struct stat buf;
  if (stat(argv[1], &buf) < 0)
    die("stat on %s", argv[1]);

  /* Determine file type */
  char *type = "other";
  if (S_ISREG(sb.st_mode))
    type = "regular";
  else if (S_ISDIR(sb.st_mode))
    type = "directory";
  /* Check read access */
  char *readok = (sb.st_mode & S_IRUSR) ? "yes" : "no";

  printf("type: %s, read: %s\n", type, readok);
  return EXIT_SUCCESS;
}
```

Atomowość operacji wejścia-wyjścia

Interfejs POSIX.1 jest **spójny sekwencyjnie** (ang. *sequential consistency*). Dla wielu procesów dopuszczalny dowolny przeplot operacji na pliku, ale zachowujemy porządek między zleceniami każdego z procesów z osobna.

Z punktu widzenia uniksa odczyty i zapisy są zawsze **atomowe**.

Tak silne gwarancje powodują, że w klastrach pojawiają się problemy z wydajnością:

- What's So Bad About POSIX I/O?
- POSIX IO Must Die!

Unix: operacje na katalogach

`int creat(const char *path, mode_t mode);`
`int unlink(const char *path);`
`int mkdir(const char *pathname, mode_t mode);`
`int rmdir(const char *pathname);`
`int rename(const char *oldpath, const char *newpath);`
`int link(const char *oldpath, const char *newpath);`
`int getdents(int fd, struct dirent *dirp, unsigned count);`

Katalog to plik indeksowany, rekordy czytamy **`getdents`**!

`rename`, **`link`** → **`EXDEV`**

```
struct dirent {
  uint64_t d_ino;    /* Inode number */
  uint64_t d_off;    /* Offset to next dirent */
  uint16_t d_reclen; /* Length of this dirent */
  char     d_name[]; /* Filename */
};
```

Niskopoziomowe listowanie katalogu

```
int main(int argc, char *argv[]) {
  int fd = x_open(argv[1], O_RDONLY | O_DIRECTORY);
  int nread;

  while ((nread = x_getdents(fd, buf, BUF_SIZE)) != 0) {
    for (int bpos = 0; bpos < nread;) {
      struct linux_dirent *d = (void *)(buf + bpos);
      char d_type = *(buf + bpos + d->d_reclen - 1);
      printf("%8ld %10s %9d %lx %s\n", d->d_ino, filetype(d_type),
             d->d_reclen, (long)d->d_off, d->d_name);
      bpos += d->d_reclen;
    }
  }

  x_close(fd);
  return EXIT_SUCCESS;
}
```

Unix: operacje na deskryptorach

```
int dup(int oldfd);
```

```
int dup2(int oldfd, int newfd);
```

```
int flock(int fd, int operation);
```

```
int fcntl(int fd, int cmd, ...);
```

Blokady **doradcze** (ang. *advisory*) i **przymusowe** (ang. *mandatory*).
W uniksach te pierwsze wystepują częściej → Linux mandatory locking.

Co i jak możemy blokować? Cały plik lub rekordy, do odczytu lub zapisu!
`fcntl` umożliwia zakładanie blokad i pieczęci, dzierżawienie plików, itp.

Q: Czy blokada jest skojarzona: z plikiem, z otwartym plikiem, z procesem?
A: Blokady rekordów `fcntl` według `POSIX` z procesem. Jeśli proces umarł lub zamknął deskryptor odnoszący się do pliku → blokady usuwane.

Unix: uprawnienia plików

Proces należy do **użytkownika**, który jest w swojej **grupie podstawowej** (ang. *primary*) i należy do grup dodatkowych (ang. *supplementary*).

Prezentacja: `id`

Każdy plik ma przypisanego **właściciela** i grupę. Pliki mają trzy zestawy uprawnień `rwx` (**`R`**`ead-`**`W`**`rite-e`**`X`**`ecute`) dla właściciela, grupy i innych.

Q: Znaczenie `rwx` dla plików jest oczywiste! A dla katalogów?
A: `R` : czytanie zawartości; `X` : dostęp do plików. jeśli nazwa jest znana;
`W` : modyfikowanie (tworzenie, usuwanie, zmiana nazwy plików)

`set-uid` / **`set-gid`** dla plików wykonywalnych – w momencie ładowania pliku nadawane są uprawnienia właściciela / grupy pliku (`su`, `login`)

`sticky` dla katalogów – usuwać może tylko właściciel pliku lub katalogu

Wektorowe wejście-wyjście

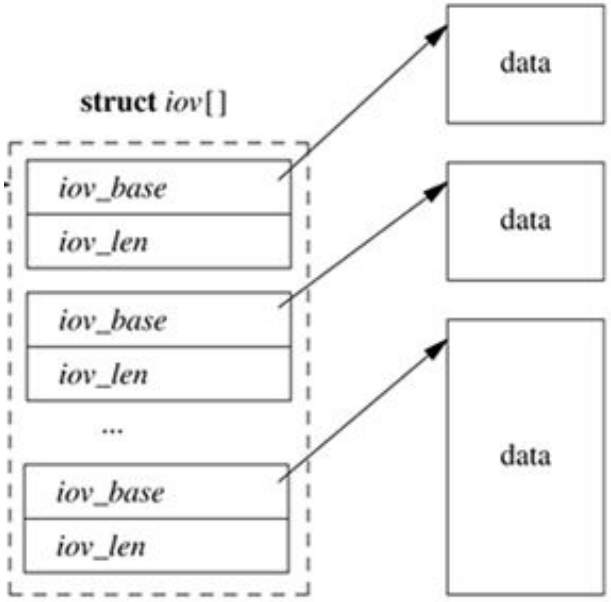
```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Motywacja: Aktualizujemy rekordy bazy danych, każdy po kilkadziesiąt bajtów. Rekordy są rozrzucone po całej tabeli.

Zapis rekordu wymaga dwóch operacji: `seek`, `write`. Koszt wywołania systemowego jest nie do zignorowania…

Jak zminimalizować koszt aktualizacji?

Scatter-gather I/O!



Inne zagadnienia

- **dziury** (ang. *file hole*), czyli **seek** za koniec pliku → rozmiar pliku vs. liczba zużywanych bloków, przydział miejsca `posix_fallocate`
- **asynchroniczne funkcje wejścia-wyjścia** POSIX.1 → `aio`
- **multipleksowanie wejścia-wyjścia** → implementacja `select` / `poll`, `kqueue` (BSD) lub `epoll` (Linux)
- **monitorowanie zdarzeń systemu plików** → kqueue lub `inotify`
- odczyt i modyfikacja **właściwości urządzeń** → `ioctl`
- listy uprawnień POSIX **Access Control List** → `acl`
- listy uprawnień **NFSv4** → `nfs4_acl`
- implementacja blokad na plikach

Pytania?