Sisteme de operare

Laborator 2

Semestrul I 2023-2024

Vlad Olaru

Laborator 2

- file I/O
- descriptori de fisiere
- operatii cu fisiere
- partajarea fisierelor
- fisiere si directoare

Descriptori de fisiere

- fisierele deschise de un proces sunt referite in kernel prin descriptori de fisiere
  - intreg nenegativ intors de apelurile open/creat

    int open(const char * pathname, int flags, mode_t mode);

    int creat(const char * pathname, mode_t mode);

- functiile de citire/scriere folosesc FDs, nu nume de fisiere !
- conventie Unix: 0,1 & 2 asociati cu stdin (STDIN_FILENO), stdout (STDOUT_FILENO) si stderr (STDERR_FILENO)
  - nu e o proprietate a kernelului, ci a shell-urilor Unix !
  - multe aplicatii Unix vor esua daca nu se respecta conventia

Open/creat/close

- open
  - flag-uri de R/W: O_RDONLY, O_WRONLY, O_RDWR
  - alte flag-uri: O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_NONBLOCK, O_SYNC
  - returneaza (garantat) cel mai mic descriptor de fisiere nealocat
    - folosit de aplicatii pentru a gestiona stdin/stdout/stderr
    - ex:

```
close(1);
fd = open("somefile", O_WRONLY); //  fd = 1 !
// a avut loc redirectarea stdout catre "somefile"
```

- creat
  - apel echivalent cu

    open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)
  - Obs: fisierul e creat doar pt scriere !
  - solutie mai buna:

    open (pathname, O_RDWR | O_CREAT | O_TRUNC, mode)

- close

  int close (int fd);
  - cand procesul executa *exit()*, toate fisierele deschise sunt automat inchise cu *close*

Pozitionarea file pointer-ului

- orice fisier deschis are asociat un “offset curent”, nr. nenegativ care exprima nr de octeti de la inceputul fisierului
  - operatiile de citire/scriere folosesc implicit acest offset si incrementeaza corespunzator offsetul current
  - initializat cu 0 daca fisierul nu a fost deschis cu O_APPEND
- pozitionarea explicita a file pointer-ului se face cu apelul sistem *lseek*:

  off_t lseek(int fd, off_t offset, int whence)
- valoarea parametrului offset este relativa la valoarea parametrului whence care poate fi
  - SEEK_SET: FP este pozitionat la *offset* bytes de la inceputul fisierului
  - SEEK_CUR: FP este incrementat/decrementat cu *offset* bytes fata de offsetul curent
  - SEEK_END: FP este pozitionat relativ la sfarsitul fisierului (*offset* poate fi pozitiv sau negativ)
- intoarce valoarea noului offset curent (file pointer, FP)
  - poate fi folosita pentru a testa daca fisierul referit este capabil de seek (pipe-uri, stdin, vor intoarce eroare)

Read/write

- ssize_t read(int fd, void *buf, size_t count);
  - intoarce nr de octeti cititi (0 daca s-a ajuns la EOF) in *buf*
  - daca FP + count > EOF, intoarce nr de octeti disponibili
  - cand fd refera un terminal, se citeste cate o linie (eg, pana la \n)
  - cand fd este un socket (citire din retea) e posibil sa intoarca mai putin octeti decat *count*
  - incepe sa citeasca de la offsetul curent si incrementeaza FP cu *count* octeti
- ssize_t write(int fd, void *buf, size_t count);
  - intoace nr de octeti scrisi
  - scrie *count* octeti incepand de la offsetul curent pe care il incrementeaza cu valoarea *count* (sau cu nr de octeti care au putut fi scrisi cu success, v. situatia de disc plin)
  - pt. fisiere deschise in mod O_APPEND, FP se pozitioneaza la sf. fisierului si de acolo incepe scrierea

Structuri de date pt fisiere deschise

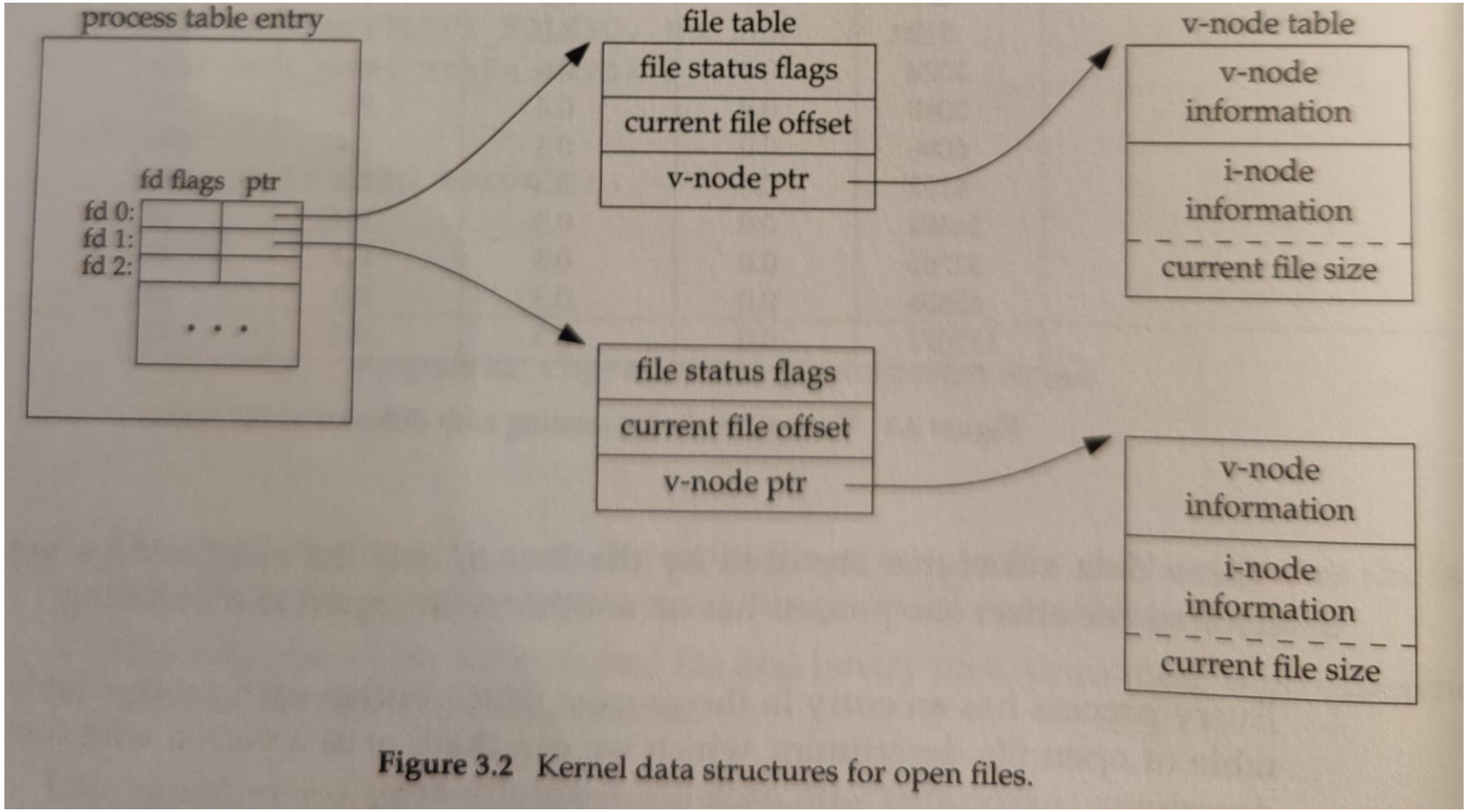

Figure 3.2 Kernel data structures for open files.

Partajarea fisierelor

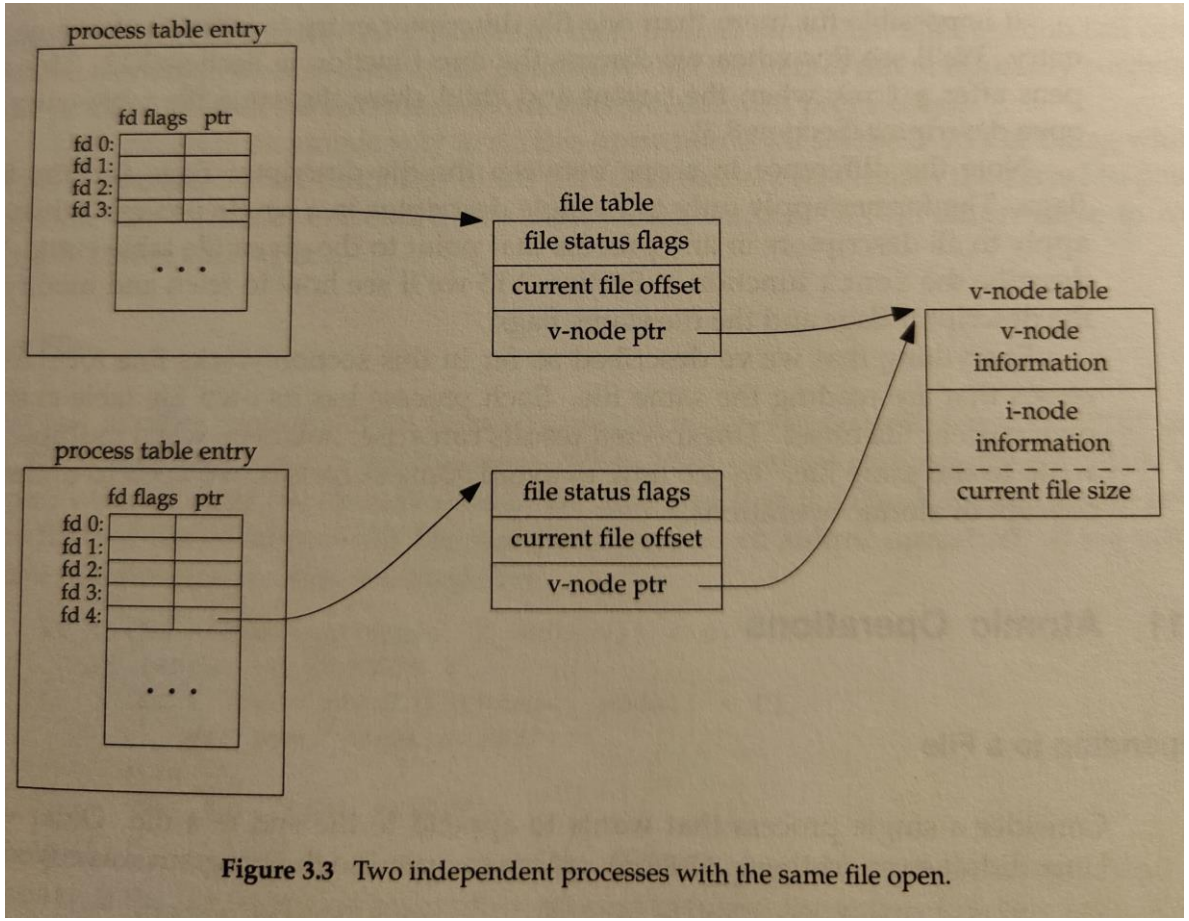

Figure 3.3 Two independent processes with the same file open.

Operatii atomice

- ex: operatia append executata de doua procese A si B
- fiecare proces executa:

```
lseek(fd, 0, SEEK_END);
write(fd, buf, 100);
```

- scenariu:
  - A executa *lseek* si pierde procesorul
  - B executa *lseek* + *write* si extinde fisierul
  - cand A revine pe procesor si executa *write* va suprascrie ceea ce a scris B
- solutie: A si B deschid fisierul cu O_APPEND
- DAR, in general, o operatie care necesita apeluri multiple de functii NU ESTE ATOMICA ! Necesita mijloace specifice de sincronizare

Fcntl/dup/dup2

- fcntl
  - schimba proprietatile unui fisier deschis

    int fcntl(int fd, int cmd, … /* args */);
  - cmd:
    - F_DUPFD: duplica descriptorul de fisier din args, fcntl intoarce noul fd (cea mai mica valoare nealocata >= argumentul din args)
    - F_GETFD/F_SETFD: get/set flag-uri fd (FD_CLOEXEC)
    - F_GETFL/F_SETFL: get/set status flags pt fd (O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_NONBLOCK, O_SYNC, O_ASYNC)
    - F_GETLK/F_SETLK: get/set record locks
- dup/dup2

  newfd = dup(fd) ⇔ fcntl(fd, F_DUPFD, 0)

  dup2(fd, newfd) ⇔ close(newfd);

  fcntl(fd, F_DUPFD, newfd);

  Obs: *dup2* e op. atomica ! *Close* + *fcntl* NU !

Dup

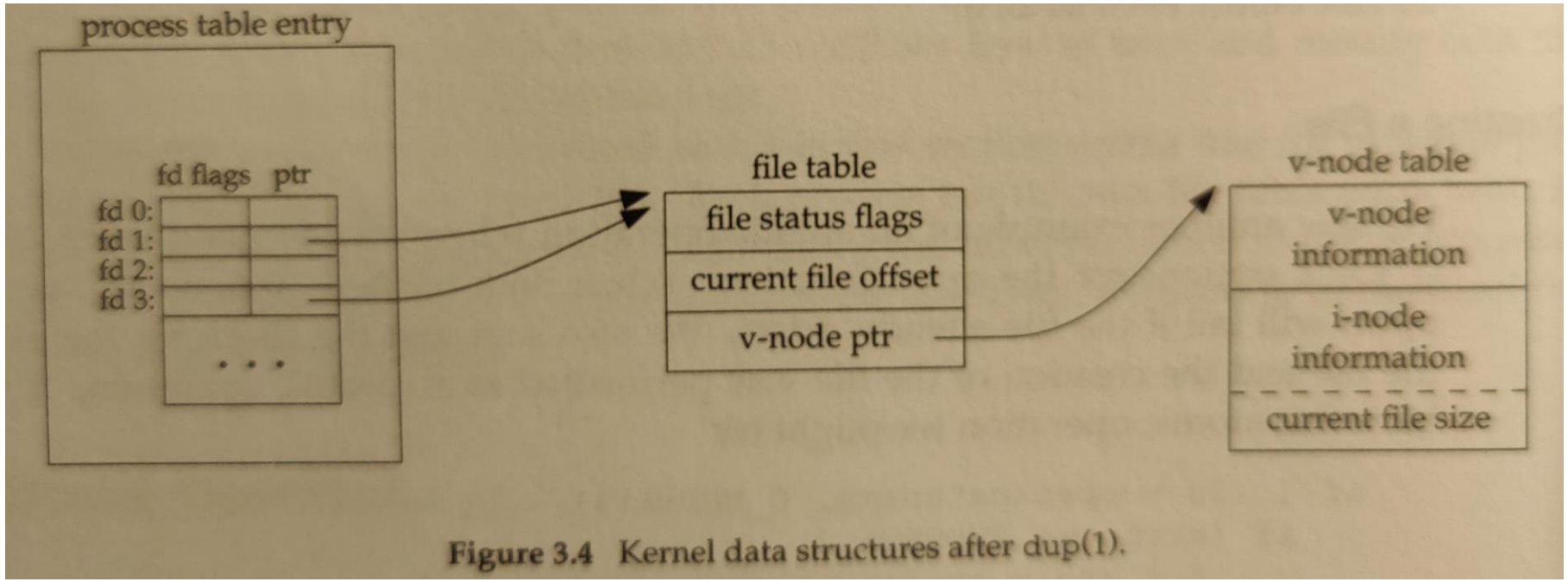

Figure 3.4 Kernel data structures after dup(1).

Stat/fstat/lstat

- furnizeaza informatii despre un fisier

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

int stat(int fd, struct stat *buf);

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

- ex:

```
struct stat buf;

lstat("/etc/profile", &buf);

printf("dimensiunea fisierului /etc/profile este %d\n", buf.st_size);
```

Obs: pt link-uri simbolice, dimensiunea fisierului este lungimea numelui fisierului referit de link

Tipuri de fisiere

- fisiere obinuite (regular files): contin date (text sau binare)
- directoare: contin numele altor fisiere si informatii despre ele
  - pot fi citite de catre procesele care au permisiunile potrivite
  - DOAR kernelul poate scrie in ele !
- fisiere speciale caracter: pt device-uri caracter (ex: tty, seriala)
- fisiere speciale block: discuri
- Obs: orice device din system e fie fisier block fie caracter
- FIFO: named pipe, mecanism IPC
- socket: abstractie pentru IPC peste retea
- link-uri simbolice: fisier care refera un alt fisier

ln –s <fisier-sursa> <link-simbolic>

ln –s /etc/profile ~/.system-wide-profile

Set UID, set GID

- fiecare proces are asociat
  - UID, GID real: identitatea reala a utilizatorului provenita din /etc/passwd
  - UID/GID efectiv
  - set-UID, set-GID salvate (copii ale UID/GID efectiv)
- in mod normal, UID/GID real = UID/GID efectiv
- cand se executa un program exista posibilitatea de a seta un flag in st_mode care spune ca:

"pe durata executiei acestui fisier UID/GID efectiv al procesului devine UID/GID-ul proprietarului fisierului"

ex: comanda de schimbare a parolei utilizator

$ passwd

este un program set-UID la root pt a avea drepturi de scriere in /etc/passwd (sau mai exact in /etc/shadow)

Permisiuni de acces la fisiere

- codificate in campul *st_mode* alt structurii *stat*
- grupate in trei categorii
  - permisiuni utilizator: S_IRUSR, S_IWUSR, S_IXUSR
  - permisiuni grup: S_IRGRP, S_IWGRP, S_IXGRP
  - permisiuni pt. alti utilizatori: S_IROTH S_IWOTH, S_IXOTH
- modificabile din shell cu ajutorul comenzii *chmod*

  ex: chmod u+rw <fisier>, chmod g-x <fisier>, chmod o-rwx <fisier>

  sau chmod 755 <fisier>, chmod 644 <fisier> , etc
- succesul accesului la un fisier e conditionat de combinatia dintre valoarea UID-ului efectiv, respectiv al GID-ului efectiv, al celui care executa comanda si bitii de permisiune

Acess/umask

- pt a testa accesibilitatea utilizatorului bazata pe UID/GID reale

  int access(const char *pathname, int mode)

  mode: “OR” logic intre valorile R_OK, W_OK, X_OK, F_OK

- orice fisier nou creat are setata o masca implicita a permisiunilor, data de valoarea umask (comanda shell)

  mode_t umask(mode_t cmask);

  cmask e un OR intre bitii de permisiune

- bitii setati in *umask* sunt off in permisiunea noului fisier creat

Chmod/fchmod

- permit schimbarea permisiunilor de acces la fisiere

  int chmod(const char *pathname, mode_t mode)

  int fchmod(int fd, m0de_t mode)

Stergerea fisierelor

- am vazut ca un fisier poate avea mai multe intrari de director care refera acelasi i-node
- un link catre un fisier care exista deja se poate crea cu

  int link(const char *pathname, const char *newpath)
- pt. a sterge o intrare de fisier dintr-un director se foloseste *unlink*

  int unlink(const char *pathname)
- stergerea necesita permisiunea de a scrie in director SI de a cauta in director (bitul x de execute setat in directorul din care stergem fisierul)
- pt a sterge directoare se foloseste *rmdir*
- *remove* pt fisiere e *unlink*, iar pt directoare e *rmdir*

  int remove(const char *pathname)

Link-uri simbolice

- *link* creaza “link-uri hard”
- limitarile link-urilor hard
  - link-ul si fisierul linkat trebuie sa se afle pe acelasi sistem de fisiere
  - doar *root-ul* poate crea linkuri hard catre directoare
- link-urile simbolice sunt fisiere care contin numele fisierului referit (un string)
- in general, comenzile shell dereferentiaza linkul simbolic
  - exceptii: *lstat*, *remove*, *rename*, *unlink*, samd

int symlink(const char *actualpath, const char *sympathy)

- creeaza un link symbolic

int readlink(const char *pathname, char * buf, int bufsize)

- deschide link-ul si citeste numele fisierului referit de link (actiune combinata *open* + *read* + *close*)

Lucrul cu directoare

- create cu *mkdir*, sterse cu *rmdir*

  int mkdir(const char *pathname, mode_t mode)

  int rmdir(const char *pathname)

- Obs:
  - *mode* in apelul *mkdir* trebuie sa includa permisiunea de execute !
  - *rmdir* nu poate sterge decat un director GOL !
- citirea directoarelor: secventa *openddir* + *readdir* + *closedir*

  DIR *openddir(const char *pathname)

  struct dirent *readdir(DIR *dp)

  closedir(DIR *dp);

Lucrul cu directoare (cont.)

- schimbarea directorului current

  int chdir(const char *pathname)

  int fchdir(int fd)

- aflarea directorului de lucru current (current working directory)

  char *getcwd(char *buf, size_t size)