This work is licensed under a Creative Commons license

Attribution-NonCommercial-NoDerivatives 4.0 International
(CC BY-NC-ND 4.0)

You are free to:

Share copy and redistribute the material in any medium or format.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial You may not use the material for commercial purposes.

NoDerivatives If you remix, transform, or build upon the material, you may not distribute the modified material.

File e processi

Le system call POSIX

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
Twitter & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

Outline

System call

- Un processo non può interagire *direttamente* con l'HW
- Una chiamata di sistema = system call, AKA syscall è una "chiamata *controllata*" all'interno del kernel
- Per effettuare una syscall è necessario usare *assembly*, ma...
- La libreria C offre delle funzioni *wrapper*

Funzioni wrapper

Queste funzioni (chiamabili da) C,

- preparano argomenti/#-syscall secondo le convenzioni del sistema
  - Es, Linux x86: #-syscall e parametri caricati in specifici registri
- eseguono una *trap* (Linux x86 = `INT 0x80`/x64 = `SYSCALL`)
- il kernel
  - controlla la validità di #/argomenti
  - esegue la richiesta e scrive il risultato in un registro
  - ritorna alla modalità utente, tramite un'istruzione speciale (`IRET`)
- la funzione wrapper controlla il risultato
  - in caso di errore imposta `errno` e restituisce un codice di errore, tipicamente -1
  - altrimenti, restituisce il risultato al chiamante

Una syscall è *molto* più costosa di una semplice chiamata a funzione

Esito

- Controllate *sempre* il valore di ritorno di funzioni/syscall
- In caso di errore, `errno` indica la ragione del fallimento
  - potete considerarla una specie di "variabile globale" intera
    - ogni thread ha la sua
  - in caso di successo, non è detto che venga azzerata/modificata

Vedete anche `errno(3)`, `perror(3)` e `strerror(3)`

Tipi di dato

- Varie informazioni, per esempio i PID, vengono memorizzate in `int`/`long`/... a seconda del sistema
- Scrivete codice portabile usando gli alias definiti dagli header; per esempio, `getpid` vi restituisce un `pid_t`
- Questo crea qualche problema quando si vogliono stampare con `printf` e funzioni simili
  - Un approccio è usate un cast a un tipo sicuramente più grande
    - Per esempio, `long` o `long long`
    - Dal C99 in poi è possibile usare `intmax_t` (e `uintmax_t`), stampandoli con `%jd` e `%ju`, definiti in `stdint.h`

Outline

File descriptors

- Tutto l'I/O avviene tramite i file descriptors
  - interi non negativi che identificano un file aperto
  - "file" in senso abbastanza generale, non solo *file regolari*, anche connessioni di rete, pipe, dispositivi, ...
  - ogni processo ha i suoi
- Tre sono "speciali" (solo per convenzione)

  0 `STDIN_FILENO` $\leftrightarrow$ `stdin`/`cin`
  1 `STDOUT_FILENO` $\leftrightarrow$ `stdout`/`cout`
  2 `STDERR_FILENO` $\leftrightarrow$ `stderr`/`cerr`

  le costanti sono definite in `unistd.h`

Sul libro vedete:
`http://pages.cs.wisc.edu/~remzi/OSTEP/file-intro.pdf`
(ma alcune cose le discuteremo più avanti, quando parleremo dell'implementazione del file-system)

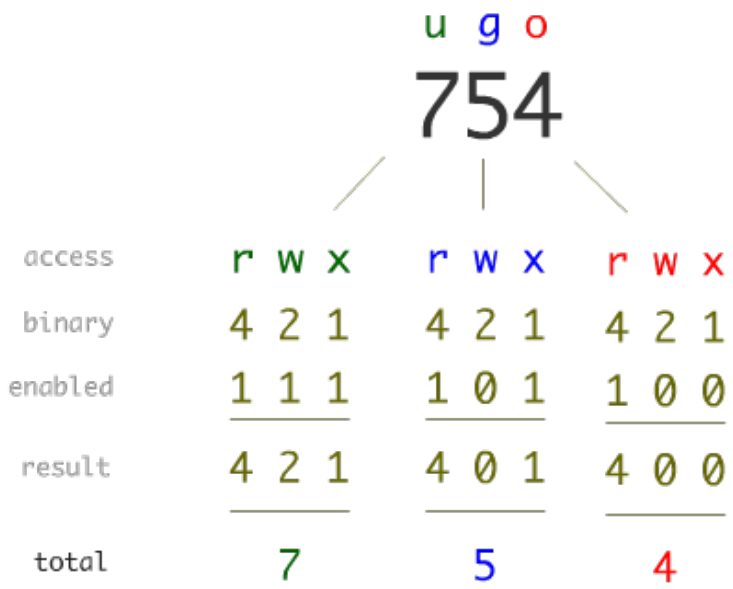
System call open

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

- `flags` *bitmask* che indica se si vuole aprire in lettura, scrittura, ...
  - i flag si possono recuperare e (qualcuno) modificare con `fcntl(2)`
  - la maggior parte dei flag riguarda il file
    - lettura/scrittura
    - `O_APPEND`
    - `O_NONBLOCK`
    - ...

    ma esistono (per ora uno: `FD_CLOEXEC`) flag associati al fd
- `mode` è utilizzato solo quando viene creato un file
  - ovvero, `flags` contiene `O_CREAT` o `O_TMPFILE`
  - può essere comodo specificarlo in ottale (vedere prossima slide)
  - stiamo ignorando `umask(2)`; in pratica, `mode &= ~umask`
  - `mode` vale per le *prossime* aperture: possiamo tranquillamente scrivere in un file *a sola lettura* se abbiamo specificato `O_WRONLY` o `O_RDWR`
- viene sempre restituito il *più piccolo* file descriptor disponibile
  - oppure -1, in caso di errore

Permessi sui file



Presa da: `https://danielmiessler.com/study/unixlinux_permissions/`

Comodo: `https://chmodcommand.com/`

Attenzione:

1. ci sono altri bit, ne parleremo in seguito
2. per cambiare permessi/proprietario `chmod`/`chown`

chmod 777

`https://twitter.com/nixcraft/status/1372473181923995651/`

Battute a parte, semplificando un po', root può leggere/scrivere qualsiasi file, ed eseguire qualsiasi file purché ci sia almeno un bit `x` settato

Per una discussione più approfondita si veda:
`https://unix.stackexchange.com/a/412247`

Esercizio

Scrivete un programma che crea un file (regolare) chiamato `pippo`

- Che sia leggibile solo dal proprietario (non scrivibile da nessuno, proprietario compreso)
- Cosa succede se il file esiste già?
  - La syscall va a buon fine?
  - Il contenuto del file esistente viene preservato?
  - Potete fare in modo che un file già esistente non venga ri-creato per errore?
- Modificate quel file con un editor di testo (potete farlo? come?)

Ripasso (e non solo) — 1/2

- Makefile
- valgrind — `http://valgrind.org`
  - Installazione: `apt install valgrind`
  - Quick-start: compilate con i simboli di debug (`-ggdb`), poi `valgrind [--leak-check=full]` *`eseguibile argomenti`*
- address sanitizer, ancora più semplice ed efficiente; compilate con:
  - `-fsanitize=address -g`

Ripasso (e non solo) — 2/2

- gdb (+ GEF)
  - GEF: `https://gef.readthedocs.io/en/master/`
  - mio "cheatsheet": `https://github.com/zxgio/gdb_gef-cheatsheet`
  - consiglio la lettura (almeno dei primi capitoli) di "GDB User manual": `https://www.sourceware.org/gdb/documentation/`
- Se siete "allergici" al terminale, sceglietevi un IDE (es. CLion)

Come ripasso sulla gestione della memoria dinamica, vedere:
`http://pages.cs.wisc.edu/~remzi/OSTEP/vm-api.pdf`

System call `read`, `write` e `close`

- `ssize_t read(int fd, void *buf, size_t count);`
- `ssize_t write(int fd, const void *buf, size_t count);`
- `int close(int fd);`
  - sì, può fallire (per esempio, fd già chiuso, errori specifici di alcuni FS)
  - in ogni caso, il file descriptor viene rilasciato → ritentare la chiusura può portare a oscure *race-condition* in programmi multithread

Byte letti/scritti

`read` e `write` possono restituire un valore più piccolo di `count` (ma > 0); questo *non* è un errore. Per esempio, da terminale (di default) si legge solo una linea. 0 indica EOF

Esercizio

Scrivete un "`cat` dei poveri", ovvero un programma che

- se invocato senza parametri legge da standard-input
- altrimenti, dal/dai file specificati da riga di comando

scrivendo tutto quello che riesce a leggere su standard-output

Debugging

Per i casi più complessi serve `gdb`, ma spesso bastano `strace` o `ltrace` (per usare quest'ultimo, potreste dover compilare con `-z lazy`)

Cosa succede se specificate:

- file che non esistono?
- file che esistono ma di cui non avete il permesso di lettura?

File offset

- A ogni file aperto è associato un file offset/"pointer" che indica in che posizione leggere/scrivere (all'interno del file)
  - usato per i file regolari
  - non ha senso andare avanti/indietro su un socket, dispositivo, etc
- Dopo `open` l'offset è 0
- Viene spostato da `read` e `write`
- Oppure da una syscall apposta:

  ```
  off_t lseek(int fd, off_t offset, int whence);
  ```

  dove `whence` può essere: `SEEK_SET`, `SEEK_CUR` o `SEEK_END`
- Può essere spostato oltre la fine del file
  - questo NON cambia la dimensione del file
  - un file può avere "buchi" che corrispondono (logicamente) a byte a 0
    - → `huge_file` / `xxd ...`

Metadati di un file

Per recuperare i metadati di un file:

- `int stat(const char *pathname, struct stat *statbuf);`
- `int lstat(const char *pathname, struct stat *statbuf);`
  - come `stat`, ma non segue i link simbolici
- `int fstat(int fd, struct stat *statbuf);`

```
struct stat {
        dev_t     st_dev;         // major (12 bits) + minor (20 bits)
        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
}
```

→ `stat /tmp/huuuuuuge huge_file{,.c}`

Limiti

Ogni tipo di filesystem ha dei limiti, per cui, per esempio, la lunghezza massima di un nome file può variare a seconda della directory considerata. Per leggere i limiti potete usare:

- `getconf(1)`; per esempio: `getconf NAME_MAX /bin`
- `fpathconf(3)/pathconf(3)`; per esempio:
  `printf("%ld\n", pathconf("/bin", _PC_NAME_MAX));`

Costanti POSIX: attenzione

Esistono delle costanti `_POSIX_`*`qualcosa`* che, nonostante abbiano "`MAX`" nel nome, corrispondono alla misura *minima* che deve essere garantita. Per esempio, `_POSIX_NAME_MAX` è 14, mentre sul mio Linux la lunghezza massima di un nome di file è 255.

Outline

Processo di compilazione e linking

gcc/clang sono programmi che "pilotano" l'intero processo di compilazione lanciando: preprocessore, compilatore, assemblatore e linker

- Compilatori e assemblatori producono file oggetto/rilocabili:

$$\texttt{*.c + *.h} \xrightarrow{\texttt{(cpp)+cc1}} \texttt{*.s} \xrightarrow{\texttt{as}} \texttt{*.o}$$

- `ld`, il link editor (AKA linker statico), mette assieme file-oggetto/librerie, eseguendo rilocazione e risoluzione dei simboli:

$$\texttt{*.o (+ *.a + *.so)} \xrightarrow{\texttt{ld}} \texttt{a.out}$$

- Infine, il linking può essere...

Linking statico e dinamico

- statico
  - `ld` fa tutto il lavoro, creando programmi autocontenuti
    - i "pezzi" di librerie necessari vengono copiati dentro al programma
    - gli eseguibili funzionano su "tutte" le macchine (con stesso HW/s.o.)
- (statico +) dinamico
  - `ld` prepara l'eseguibile, "annotando" le dipendenze esterne
  - fra cui, il suo *interprete, ovvero, il linker dinamico*: `ld.so`
  - quest'ultimo mette assieme i pezzi mancanti a tempo di esecuzione
  - *default* sui sistemi moderni
    - fa risparmiare spazio, sia su disco, sia in memoria fisica
    - facilita l'aggiornamento (ma anche la "distruzione globale")
    - può essere problematico portare l'eseguibile da una macchina a un'altra

Cosa contiene un eseguibile?

Gli eseguibili, così come i file rilocabili (`.o`) e le librerie possono contenere:

- Codice macchina → sezione `.text`
- Dati e dati a sola lettura → `.data` e `.rodata`
- Metadati
  - architettura (Intel/ARM/..., 32/64 bits, little/big endian, ...)
  - *entry-point*; no, non è il `main`
  - quanto spazio riservare per variabili globali non inizializzate → `.bss`
  - ...

In un eseguibile le sezioni vengono raggruppate in segmenti, i "pezzi" che vengono *mappati* in memoria dal s.o./linker-dinamico per l'esecuzione

Attenzione: questi *segmenti* NON necessariamente corrispondono ai segmenti della CPU/MMU

Spazi di indirizzamento (*Address spaces*)

Ogni processo "vede" il suo spazio di indirizzamento (*address space*), un'astrazione della memoria fisica
`http://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf`

Ovvero,

- i processi usano indirizzi logici/virtuali
- che vengono tradotti in indirizzi fisici dalla MMU
  `http://pages.cs.wisc.edu/~remzi/OSTEP/vm-mechanism.pdf`
  tramite:
  - segmentazione
    `http://pages.cs.wisc.edu/~remzi/OSTEP/vm-segmentation.pdf` e/o
  - paginazione
    `http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf`
    `http://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf`
    `http://pages.cs.wisc.edu/~remzi/OSTEP/vm-smalltables.pdf`

Queste sono cose che dovreste aver visto ad "Architettura dei Calcolatori".
Il succo è che: `(void *)0xbadc0ff3` non è uguale per tutti ☺

Cosa ci serve per far “girare” i programmi?

Ci basta *mappare* codice e dati, dal un ELF, per far “girare” programmi?

No, servono anche:

- stack — variabili locali/temporanei, parametri, indirizzi di ritorno
  - per ragioni “storiche”, cresce verso indirizzi descrescenti
- heap — memoria dinamica: `malloc`, `free`, etc
  - non serve davvero un altro segmento, può crescere il segmento dati
- il kernel, non accessibile in modalità utente
- . . .

Il codice di programmi e, grazie al linking-dinamico, librerie può essere condiviso (e mappato a indirizzi *diversi*) in processi diversi

Outline

(P)PID

- `pid_t getpid(void)`
- `pid_t getppid(void)`

I processi formano un albero, con radice *init* (PID=1)

- su Ubuntu in realtà è *systemd*, ma lo chiameremo lo stesso init ☺

PID

I PID vengono riutilizzati, quindi "identificano" i processi solo in uno specifico momento

Il kernel espone le informazioni tramite lo pseudo-filesystem /proc

- una directory per ogni processo
- vari file, la maggior parte a sola lettura
  - per esempio, `maps` mostra lo spazio di indirizzamento
- vedere `proc(5)`

Directory di lavoro

Ogni processo ha una

- directory root (radice), che viene usata per tutti i percorsi assoluti (=che iniziano con /)
  - non è necessariamente la root del file-system, ne parleremo più avanti
- directory di lavoro, che viene usata per tutti i percorsi relativi; vedere
  - `getcwd(2)` o, estensione GNU, `get_current_dir_name(3)` restituiscono quella corrente
  - `chdir(2)` e `fchdir(2)`, la modificano

API per la gestione dei processi

Le syscall principali per gestire i processi sono solo quattro:

- `fork`, crea un nuovo processo
- `_exit` (standard C: `exit`), termina il processo chiamante
  - quando non ci saranno ambiguità, diremo semplicemente *exit*
- `wait`, aspetta la terminazione di un processo figlio
  - non è vero al 100%, ne parliamo dopo
- `execve`, esegue un nuovo programma nel processo chiamante, *sostituendo l'intero spazio di indirizzamento*; spesso,
  - invocata dopo `fork`; pensate, ad esempio, alla shell
  - chiamata semplicemente *exec* o *exec**, per indicare l'intera famiglia di funzioni per eseguire programmi

Fork

`pid_t fork(void)`

- "clona" un processo
  - *init* è speciale
  - sì, in Linux esistono `clone`/`[__]clone2`/`clone3`, ma non ne parliamo
- creando un nuovo processo, detto "figlio" del processo chiamante
- copia *quasi* identica del processo chiamante
  - cambiano chiaramente PID, PPID; però
  - i FD (flag compresi) vengono duplicati
  - l'intero spazio di indirizzamento viene copiato
    - nei sistemi moderni, copia logica grazie al *copy-on-write*
    - in passato, "trucchi sporchi" tipo `vfork` per evitare la copia
- in caso di successo, ritorna due volte (!), restituendo
  - il PID del figlio al padre
  - 0 al figlio

→ `fork_example1`

Fork in gdb

Normalmente gdb continua a seguire solo uno dei due processi

- scegliete quale con:
  `set follow-fork-mode [child|parent]`
- per seguirli entrambi (sconsigliato, è facile fare casino ☺):
  `set detach-on-fork off`
  - per vedere i processi collegati a gdb: `info inferiors`
  - per selezionare quello corrente: `inferior` *id*

Esempio: double_fork

```c
#include <sys/types.h>
#include <unistd.h>

int main()
{
        // per far stare nella slide, non controllo i valori di
        // ritorno; NON fatelo nel codice "vero"
        char msg[] = "pippo\n";
        fork();
        fork();
        write(STDOUT_FILENO, msg, sizeof msg);
}
```

. . . quanti “pippo”?

→ double_fork

Exit

```
void exit(int status)  // libreria C
void _exit(int status) // syscall (in Linux, exit_group(2))
```

La prima:

- chiama le funzioni registrate con `atexit(3)` e `on_exit(3)`
- svuota i buffer di I/O
- elimina file temporanei creati con `tmpfile(3)`

In entrambi i casi:

- vengono chiuse/rilasciate le risorse del processo
  - file descriptor, memory-mapping, System-V IPC objects, ...
- il processo termina con *exit-status* uguale a `(status & 0xff)`
  - dalla shell lo recuperate con `$?`
- eventuali figli, ora orfani, vengono adottati da *init* (PID=1)
  - in realtà, un po' più complicato (`prctl(2)`), ma ignoriamo la cosa

Restituire *n* dal `main` corrisponde a `exit(`*n*`)`; lo standard C definisce le costanti `EXIT_SUCCESS` (=0) ed `EXIT_FAILURE` (=1)

→ `exit_vs_exit`

Wait

`pid_t wait(int *wstatus)`
attende un "cambio di stato" in un figlio

- terminazione
- stop/ripartenza, tramite segnali

per aspettare un figlio particolare e altre opzioni → `waitpid(2)`

Se `wait` va a buon fine, `wstatus!=0` e

- `WIFEXITED(*wstatus)`, allora potete recuperare lo *exit-status* con `WEXITSTATUS(*wstatus)`
- `WIFSIGNALED(*wstatus)`, allora potete recuperare il segnale con `WTERMSIG(*wstatus)`
- ...

→ `fork_example2`

Zombie

Un processo terminato, non aspettato dal padre, è uno zombie

- il sistema rilascia alcune risorse, ma deve continuare a memorizzare alcune informazioni, per esempio, PID ed *exit-status*
- quando un processo termina, viene inviato al padre il segnale `SIGCHLD`
  - di default è ignorato
  - può essere catturato per "aspettare" i figli
    - siccome i segnali (non realtime) non si accodano, potrebbero esserci più figli terminati per un singolo segnale

→ `gng`

Exec (1/3)

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
// libc:
int execl(const char *pathname, const char *arg, ...  /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...  /* (char  *) NULL */);
int execle(const char *pathname, const char *arg, ...  /*, (char *) NULL,
                                                    char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Notate che `v`, `l`, `p` ed `e` NON sono messe a caso ☺

Viene avviato un programma all'interno del processo chiamante, sostituendone lo spazio di indirizzamento (prox slide), ma stessi:

- PID, PPID
- file descriptor (a meno di `FD_CLOEXEC`)

Exec (2/3)

Quando eseguiamo un programma il sistema (kernel+linker) prepara:

- codice, mappato `r-x` (`.text`)
- dati a sola lettura, mappati `r--` (`.rodata`)
- dati, mappati `rw-` (`.data`, `.bss` e *heap*)
- stack, mappato `rw-`
  - in fondo allo stack (quindi, a indirizzi più grandi) il kernel copia:
    - le variabili d'ambiente
    - gli argomenti della riga di comando
    - (altra "roba")
- (il kernel, "invisibile")

Poi,

- in caso di linking dinamico fa la stessa cosa, ricorsivamente, per codice e dati delle librerie necessarie (+*linking* vero e proprio)
- quando viene creato un thread, viene anche allocato un nuovo stack

Nei sistemi moderni, codice/dati vengono portati in RAM *on-demand*

Exec (3/3)

Se la syscall/funzione ritorna al chiamante, qualcosa è andato storto!

Se l'eseguibile ha i bit *set-user-ID/set-group-ID* abilitato, succedono "cose" (ne parleremo in un altro momento)

→ `exec{1,2}`

Libro e API per i processi

Il libro [ADAD18] tratta le syscall per la gestione di processi nel capitolo 5: `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf`

Attenzione: il libro fa un po' di confusione

A un certo punto, sembra che "creare un processo" equivalga a "voglio lanciare un programma, quindi faccio fork+exec".

- In POSIX l'unica syscall per creare un processo è `fork`
- Le varie `exec*`, a parte `execve`, NON sono syscall
- `execve` rimpiazza lo spazio di indirizzamento di un processo con quello per eseguire un programma. `exec*` NON creano un nuovo processo e NON fanno parte della *creazione* di un processo in senso stretto.

Leggete bene le pagine di manuale corrispondenti

Outline

Dup

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

- crea un FD *equivalente* a `oldfd`
  - a livello kernel, entrambi i FD "punteranno" allo stesso file aperto
  - potranno essere usati in modo intercambiabile: stessi offset, flag, inode
- restituisce il più piccolo FD disponibile (come `open`)
- non vengono copiati i flag del FD
  - `FD_CLOEXEC` non sarà attivo per quello nuovo; vedere `fcntl(2)`

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

- come `dup`, ma usa `newfd`
  - chiudendolo se necessario (solo le `oldfd` è valido)
- se `oldfd==newfd` non fa nulla

Linux offre anche una `dup3`

Relazione fra file descriptor e file aperti: POSIX

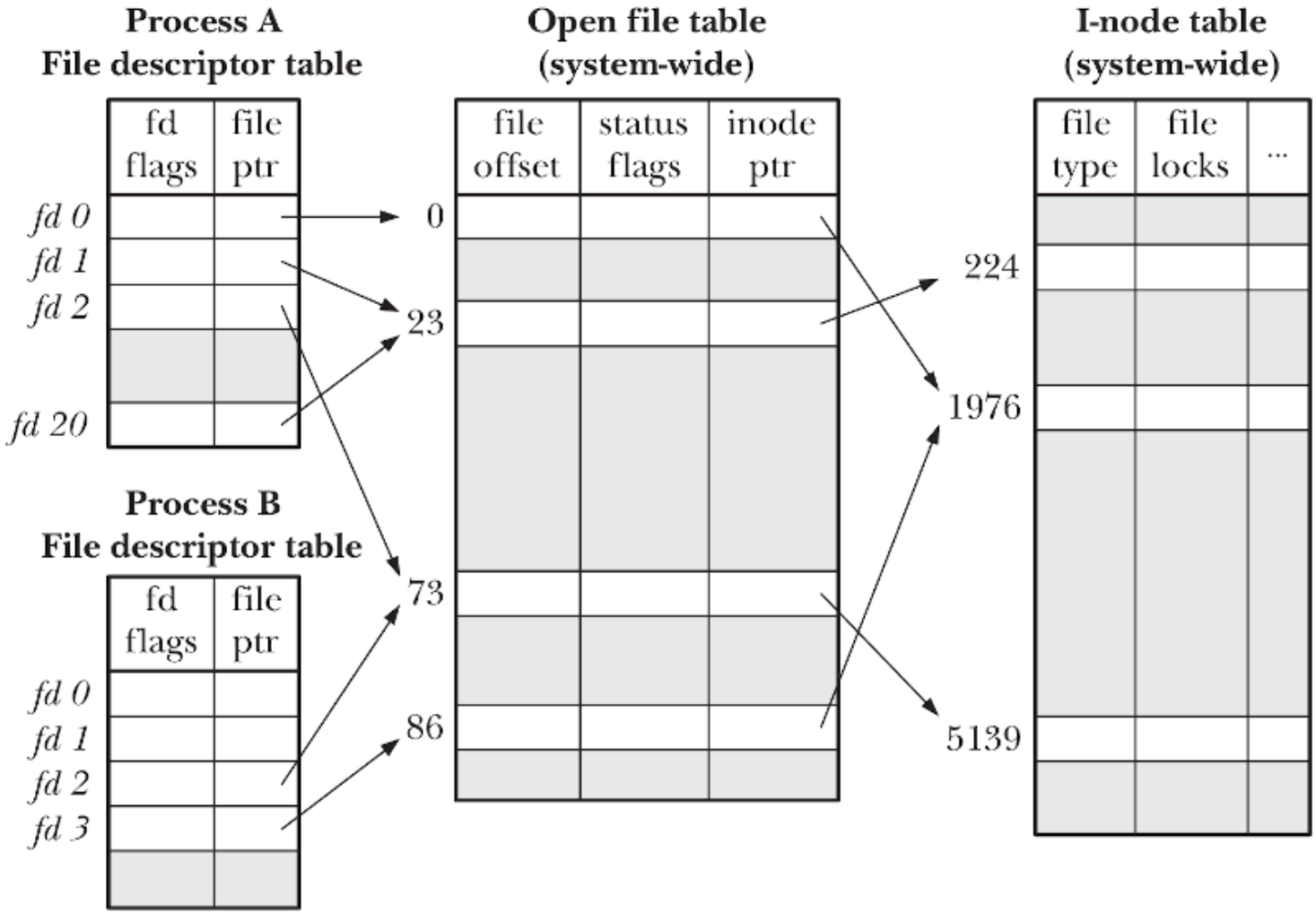


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Presa dal libro [Ker10]

Fd vs file-aperti vs inode

Notate che:

- Più FD possono "puntare" lo stesso file aperto → `dup(2)`, `dup2(2)`
- FD di processi diversi possono puntare lo stesso file aperto → `fork(2)`
  - anche FD diversi di processi diversi possono puntare lo stesso file aperto
- Più `open` possono aprire lo stesso file, quindi stesso *inode*

Quindi, è ben diverso aprire due volte lo stesso file e duplicare un FD

- anche in assenza di *race conditions*
- pensate a `read`, `write` e `lseek`

Pipe (anonime)

```
int pipe(int pipefd[2])
```

- crea un canale unidirezionale (anonimo)
- il canale è un *byte-stream*: non c'è un concetto di messaggio e/o corrispondenza fra numero di scritture e letture
- tipicamente usato per far comunicare processi
- `pipefd[0]` è il "lato" lettura, `pipefd[1]` quello scrittura
  - quello che viene scritto nella pipe finisce in un buffer del kernel
  - quando il buffer è pieno, le `write` vengono messe in attesa, o falliscono se flag `O_NONBLOCK` è abilitato
  - se tutti i FD di lettura sono stati chiusi, `write` provoca `SIGPIPE`
  - quando il buffer è vuoto, le `read` vengono messe in attesa, o falliscono se flag `O_NONBLOCK` è abilitato
  - se tutti i FD di scrittura sono stati chiusi, `read` restituisce 0, come EOF

Vedere `pipe(2)` e `pipe(7)`

Bibliografia

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.
Operating Systems: Three Easy Pieces.
Arpaci-Dusseau Books, August 2018.
http://pages.cs.wisc.edu/~remzi/OSTEP/.

[Ker10] Michael Kerrisk.
The Linux programming interface: a Linux and UNIX system programming handbook.
No Starch Press, 2010.