Laboratorio di Sistemi Operativi

System Call

Renzo Davoli
Alberto Montresor

Sommario

Introduzione

- System call
  - per utilizzare i servizi del s.o., il programmatore ha a disposizione una serie di “entry point” per il kernel chiamate system calls
    - in UNIX, circa 70-200, a seconda della versione
  - nei sistemi UNIX, ogni system call corrisponde ad un funzione di libreria C
    - il programmatore chiama la funzione, utilizzando la sequenza di chiamata standard di C
    - la funzione invoca il servizio del sistema operativo nel modo più opportuno

Introduzione

- Funzioni di libreria:
  - funzioni di utilità che forniscono servizi general purpose al programmatore
  - queste funzioni non sono entry point del kernel, sebbene alcune di esse possano fare uso delle system call per realizzare i propri servizi
  - esempi:
    - La funzione **`printf`** può invocare la system call **`write`** per stampare
    - La funzione **`strcpy`** (string copy) e la funzione **`atoi`** (convert ASCII to integer) non coinvolgono il sistema operativo

Introduzione

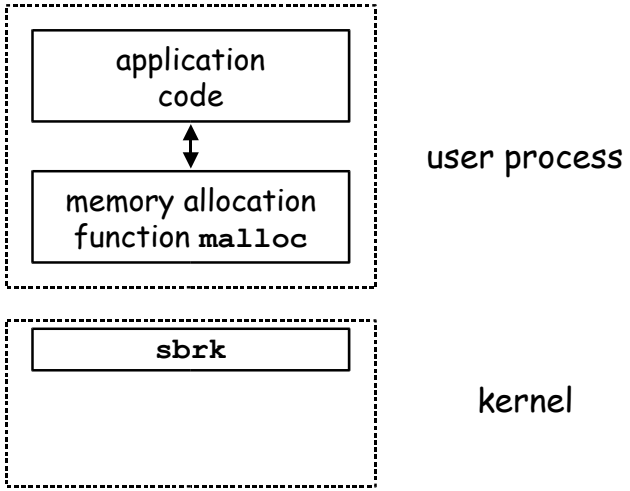
- **Distinzione:**
  - Dal punto di vista del programmatore, non vi sono grosse distinzioni fra le funzioni di libreria e le system call
    - Entrambe sono funzioni
  - Dal punto di vista di chi implementa un sistema operativo, la differenza è ovviamente più importante
- **Nota:**
  - E' possibile sostituire le funzioni di libreria con altre funzioni che realizzino lo stesso compito, magari in modo diverso
  - In generale, non è possibile sostituire le system call, che dipendono dal sistema operativo

Introduzione

- **Le funzioni `malloc, free`**
  - gestiscono l'allocazione della memoria e le procedure associate di garbage collection
  - esistono diversi modi per implementare la **`malloc`** e la **`free`** (best fit, first fit, worst fit, etc)
  - è possibile sostituire la **`malloc`** con il nostro meccanismo di allocazione
- **La system call `sbrk`:**
  - non è una funzione di allocazione della memoria general-purpose;
  - il suo unico compito è di aumentare o diminuire lo spazio di memoria associato ad un processo

Introduzione

- **La funzione `malloc` (e qualunque funzione alternativa per la gestione della memoria) è basata sulla `sbrk`:**



Introduzione - Standard

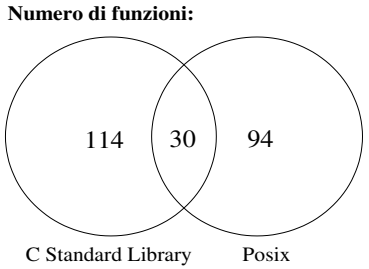
- **POSIX (Portable Operating System Interface)**
  - E' una famiglia di standard sviluppato dall'IEEE
    - IEEE 1003.1 (POSIX.1)
      Definizione delle system call (operating system interface)
    - IEEE 1003.2 (POSIX.2)
      Shell e utility
    - IEEE 1003.7 (POSIX.7)
      System administration
  - Noi siamo interessati allo standard IEEE 1003.1 (POSIX.1)
    - POSIX.1 1988 Original standard
    - POSIX.1 1990 Revised text
    - POSIX.1a 1993 Addendum
    - POSIX.1b 1993 Real-time extensions
    - POSIX.1c 1996 Thread extensions

Introduzione - Standard

- **POSIX.1**
  - `cpio.h` valori di archivio x cpio
  - `dirent.h` gestione directory
  - `fcntl.h` controllo dei file
  - `grp.h` gestione del file group
  - `pwd.h` gestione del file password
  - `tar.h` valori di archivio x tar
  - `termios.h` I/O terminale
  - `unistd.h` costanti simboliche
  - `utime.h` informazioni sul tempo / file
  - `sys/stat.h` informazioni sui file
  - `sys/times.h` informazioni sul tempo / processi
  - `sys/types.h` tipi di dato di sistema
  - `sys/wait.h` controllo processi

Introduzione - Standard

- **POSIX.1 include anche alcune primitive della C Standard Library**

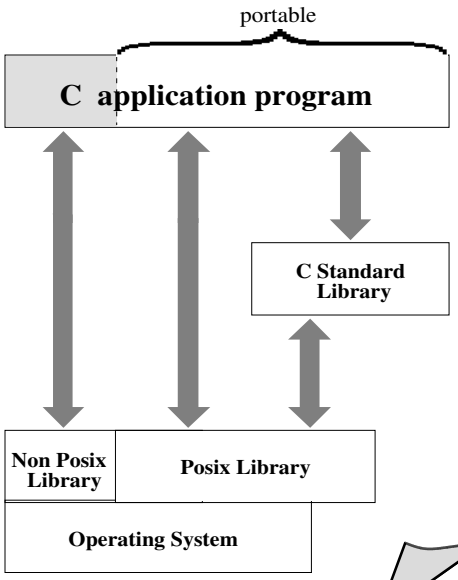


- **Esempio:**
  - `open()` è POSIX
  - `fopen()` è Standard C e POSIX
  - `sin()` è Standard C

Introduzione - Standard

- **Portabilità:**

  Un sorgente C Standard conforme a POSIX può essere eseguito, una volta ricompilato, su qualunque sistema POSIX dotato di un ambiente di programmazione C Standard



Introduzione - Standard

- **Per scrivere un programma conforme a POSIX, bisogna includere gli header files richiesti dalle varie primitive usate, scelti fra:**
  - header files della C Standard Library (contenenti opportune modifiche POSIX)
  - header files specifici POSIX
- **Nel seguito, gli header files saranno omessi dalle slide;**
  - per ottenere la lista completa degli header necessari, utilizzate `man`
- **Inoltre:**
  - Se vogliamo compilare un programma in modo tale che dipenda solo dallo standard POSIX, dobbiamo inserire prima degli `#include`:

```
#define _POSIX_SOURCE 1
```

Introduzione – Tipi Primitivi

- **Tipi di dato**
  - Storicamente, le variabili UNIX sono state associate a certi tipi di dato C
    - ad esempio: nel passato, i major/minor device number sono stati collocati un variabile a 16 bit (8 major, 8 minor)
    - sistemi più grandi possono richiedere una dimensione maggiori (e.g. SVR4 usa 32 bit, 14 major, 18 minor)
  - Lo header `sys/types.h` definisce i tipi dei dati di sistema primitivi (dipendenti dall'implementazione):
    - ad esempio, in linux `sys/types.h` definisce il tipo `dev_t` come una quantità a 16 bit
    - questi tipi sono definiti per garantire portabilità
    - sono definiti tramite `typedef`

Introduzione – Tipi Primitivi

- **Alcuni dei primitive system data types**
  - `clock_t` counter of clock ticks
  - `dev_t` device numbers
  - `fd_set` file descriptor set
  - `fpos_t` file position
  - `gid_t` group id
  - `ino_t` inode number
  - `mode_t` file types, file creation mode
  - `nlink_t` number of hard links
  - `off_t` offset
  - `pid_t` process id
  - `size_t` dimensioni (unsigned)
  - `ssize_t` count of bytes (signed) (read, write)
  - `time_t` counter of seconds since the Epoch
  - `uid_t` user id

Introduzione - Limiti

- **Limiti delle implementazioni UNIX**
  - esistono un gran numero di valori costanti e magic number che dipendono dall'implementazione di UNIX
  - nel passato, questi valori erano "hard-coded" nei programmi (i valori venivano inseriti direttamente)
  - per garantire una maggior portabilità, i vari standard hanno definito dei metodi più portabili per accedere alle costanti specifiche di un'implementazione

Introduzione - Limiti

- **Si considerano tre classi di valori costanti:**
  - opzioni compile-time (il sistema supporta job control?)
  - limiti compile-time (quant'è il valore massimo per short?)
  - limiti run-time (quanti caratteri in un nome di file?)
- **Note:**
  - le prime due classi sono determinate a tempo di compilazione, e quindi i loro valori possono essere definiti come costanti negli header di POSIX/ANSI C
  - la terza classe richiede che il valore sia ottenuto chiamando un'opportuna funzione

Introduzione - Limiti

- **Limiti ANSI-C**
  - tutti i limiti ANSI C sono limiti compile-time
  - sono definiti nello header `limits.h` (`/usr/include/limits.h`)
  - contiene informazioni sui tipi primitivi, come ad esempio
    - numero di bit in un `char`
    - valore massimi/minimi di `char`, `signed char`, `unsigned char`, `int`, `unsigned int`, `short`, `unsigned short`, `long`, `unsigned long`
  - vedi `limits.h`

Introduzione - Limiti

- **Limiti POSIX.1**
  - POSIX.1 definisce un gran numero di costanti che definiscono limiti implementativi del sistema operativo
  - questo è uno degli aspetti più complessi di POSIX.1
- **Limiti runtime**
  - alcuni limiti runtime possono essere fissi in un'implementazione
  - altri possono variare all'interno della stessa implementazione
  - esempio:
    la dimensione massima di nome di file dipende dal particolare file system considerato

Introduzione - Limiti

- **33 limiti e costanti, suddivisi in 7 categorie:**
  - *valori minimi invarianti* (13 costanti definite in `limits.h`)
    - questi valori non cambiano da un sistema ad un altro.
    - rappresentano i valori minimi che devono essere supportati da qualunque implementazione POSIX
      - `_POSIX_ARG_MAX` **number of arguments** **4096**
      - `_POSIX_CHILD_MAX` **number of child processes per uid** **6**
      - `_POSIX_LINK_MAX` **number of links to a file** **8**
      - ...
    - nota: limiti troppo bassi per essere significativi
  - *limiti invarianti*:
    - `SSIZE_MAX` max size of type `ssize`
  - *limiti che possono crescere a run-time*:
    - `NGROUPS_MAX` **`max n. of gids associated to a process`**

Introduzione - Limiti

- **33 limiti e costanti, suddivisi in 7 categorie:**
  - *Limiti invarianti run-time* (eventualmente indeterminati):
    - `ARG_MAX` max. number of arguments for exec
    - `CHILD_MAX` max. number of processes per user ID
    - `CLK_TCK` number of clocks ticks per second
    - `OPEN_MAX` max number of open files per process
    - `PASS_MAX` max. number of significant char. in password
    - `STREAM_MAX` max. number of standard I/O streams
    - `TZNAME_MAX` max. number of byte for names of a time zone
  - *Costanti simboliche a tempo di compilazione*:
    - `_POSIX_SAVED_IDS` if implementations supports saved ids
    - `_POSIX _VERSION` posix version
    - `_POSIX _JOB_CONTROL` if implementations supports job control

Introduzione - Limiti

- **33 limiti e costanti, suddivisi in 7 categorie:**
  - limiti variabili per i pathname (eventualmente indeterm.):
    - `LINK_MAX` max value of link's count
    - `MAX_CANON` terminal-related
    - `MAX_INPUT` terminal-related
    - `NAME_MAX` max number of chars for a filename (no-null)
    - `PATH_MAX` max number of chars for a pathname (no-null)
    - `PIPE_BUF` max. number of bytes atomic. written in a pipe
  - flag variabili:
    - `_POSIX_NO_TRUNC` if filenames longer than NAME_MAX are truncated
    - `_POSIX_CHOWN_RESTRICTED` if use of chown is restricted

Introduzione - Limiti

- **Funzioni di lettura limiti:**
  - `long sysconf(int name);`
  - `long pathconf(const char *path, int name);`
  - `long fpathconf(int filedes, int name)`
- **La prima legge i valori invarianti**
  - esempio di nome: _SC_*nome costante*
- **La seconda e la terza leggono i valori che possono variare a seconda del file a cui sono applicati**
  - esempio di nome: _PC_*nome_costante*
- **Vedi codice contenuto in `conf.c`**

Introduzione - Limiti

- **Limiti indeterminati**
  - se non è definito nello header `limits.h`…
  - se `sysconfig()` o `pathconfig()` ritornano –1…
  - …allora il valore non è definito
- **Esempio:**
  - molti programmi hanno necessità di allocare spazio per i pathname; la domanda è: quanto?
  - `pathalloc.c` cerca di determinare `PATH_MAX`
  - in caso il valore "di default" non sia sufficiente, in alcuni casi è possibile aumentare lo spazio previsto e ritentare
  - esempio: `getcwd()` (get current working directory)

Introduzione - Limiti

- **Esempio**
  - una sequenza di codice comune in un processo daemon (che esegue in background, non connesso ad un terminale) è quella di chiudere tutti i file aperti
  - come?

```
#include <sys/param.h>

for (i=0; i < NOFILE; i++) close(i);
```

  - oppure può utilizzare il codice contenuto in `openmax.c`

Introduzione – Gestione Errore

- **La maggior parte delle system call:**
  - restituisce il valore -1 in caso di errore ed assegna lo specifico codice di errore alla variabile globale

```
extern int errno;
```

- **Nota:**
  - se la system call ha successo, `errno` non viene resettato
- **Lo header file `errno.h` contiene la definizione dei nomi simbolici dei codici di errore**

```
# define EPERM  1    /* Not owner */
# define ENOENT 2    /* No such file or dir */
# define ESRCH  3    /* No such process */
# define EINTR  4    /* Interr. system call */
# define EIO    5    /* I/O error */
```


Introduzione – Gestione Errore

- **La primitiva `void perror (const char *str )`**
  - converte il codice in `errno` in un messaggio in inglese, e lo stampa anteponendogli il messaggio di errore `str`
- **Esempio:**

```
...
fd=open("nonexist.txt", O_RDONLY);
if (fd==-1) perror ("main");
...
-->  main: No such file or directory
```


Controllo processi

- **Identificatore di processo**
  - ogni processo ha un identificatore univoco (intero non negativo)
  - utilizzato spesso per creare altri identificatori e garantire unicità
    - funzione `char* tmpnam(char* );`
- **Identificatori standard:**
  - pid 0: Non assegnato o assegnato a un kernel process
  - pid 1: Processo `init (/sbin/init)`
    - Invocato dal kernel al termine della procedura di bootstrap
    - Porta il sistema ad un certo stato (ad es. multiuser)

Controllo processi

- **Process id ed altri identificatori**
  - `pid_t getpid();` // Process id of calling process
  - `pid_t getppid();` // Process id of parent process
  - `uid_t getuid();` // Real user id
  - `uid_t geteuid();` // Effective user id
  - `gid_t getgid();` // Real group id
  - `gid_t getegid();` // Effective group id

Controllo processi - Creazione

- **System call:** `pid_t fork();`
  - crea un nuovo processo child, copiando completamente l'immagine di memoria del processo parent
    - data, heap, stack vengono copiati
    - il codice viene spesso condiviso
    - in alcuni casi, si esegue copy-on-write
  - sia il processo child che il processo parent continuano ad eseguire l'istruzione successiva alla **`fork`**
  - **`fork`** viene chiamata una volta, ma ritorna due volte
    - processo child: ritorna 0
      (E' possibile accedere al pid del parent tramite la system call **`getppid`**)
    - processo parente: ritorna il process id del processo child
    - errore: ritorna valore negativo

Controllo processi - Creazione

- **Esempio: fork1.c**
  - Scopo di questo esempio è mostrare alcune caratteristiche della creazione di processi
- **Nota:**
  - inizializza un paio di variabili
  - stampa utilizzando **`write`**
  - stampa utilizzando **`printf`**
  - chiama **`fork`** e crea un processo child
  - continua in modo diverso:
    - Il processo child modifica le variabili
    - Il processo parent dorme per due secondi
  - stampa il contenuto delle variabili, etc.

Controllo processi - Creazione

- **Esecuzione:**
  - in generale, l'ordine di esecuzione tra child e parent dipende dal meccanismo di scheduling
  - il trucco utilizzato in questo prima programma (**`sleep`** per due secondi) non garantisce nulla; meglio usare **`wait()`**
- **Output:**

```
$ a.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89        variabili child cambiate
pid = 429, glob = 6, var = 88        variabili parent immutate
```


Controllo processi - Creazione

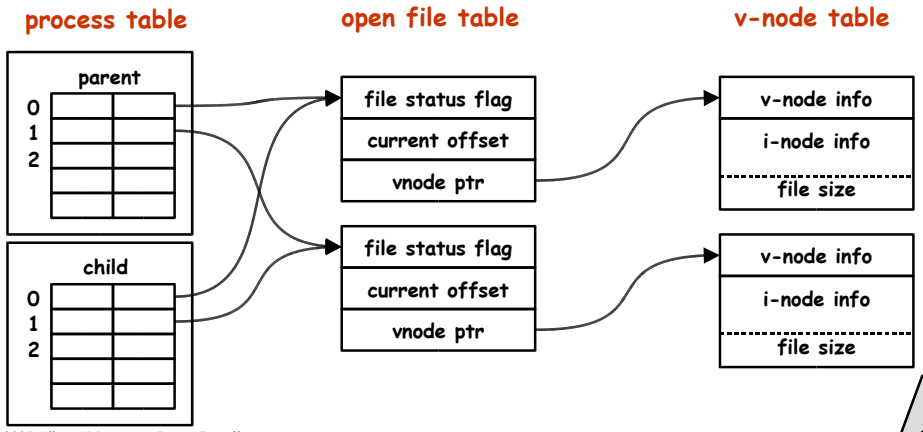
- **Relazione tra `fork` e funzioni di I/O**

```
$ a.out > temp.out ; cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89 variabili child cambiate
before fork
pid = 429, glob = 6, var = 88 variabili parent immutate
```

- **Spiegazione:**
  - **`write`** non è bufferizzata
  - **`printf`** è bufferizzata
    - line-buffered se connessa ad un terminal device
    - full-buffered altrimenti

Controllo processi - Creazione

- **Relazione tra `fork` e file aperti (redirezione)**
  - una caratteristica della chiamata **`fork`** è che tutti i descrittori che sono aperti nel processo parent sono duplicati nel processo child



Controllo processi - Creazione

- **Relazione tra `fork` e file aperti (redirezione)**
  - è importante notare che parent e child condividono lo stesso file offset
  - esempio:
    - si consideri un processo che esegue **`fork`** e poi attende che il processo child termini (system call **`wait`**)
    - supponiamo che lo **`stdout`** sia rediretto ad un file, e che entrambi i processi scrivano su **`stdout`**
    - se parent e child non condividessero lo stesso offset, avremmo un problema:
      - il child scrive su **`stdout`** e aggiorna il proprio current offset
      - il parent sovrascrive **`stdout`** e aggiorna il proprio current offset

Controllo processi - Creazione

- **Come gestire i descrittori dopo una `fork`**
  - il processo parent aspetta che il processo child termini
    - in questo caso, i file descriptor vengono lasciati immutati
    - eventuali modifiche ai file fatte dal processo child verranno riflesse nella file table entry e nella v-node entry del processo figlio
  - i processi parent e child sono indipendenti
    - in questo caso, ognuno chiuderà i descrittori non necessari e proseguirà opportunamente

Controllo processi - Creazione

- **Proprietà che il processo child eredita dal processo parent:**
  - real uid, real gid, effective uid, effective gid
  - gids supplementari
  - id del gruppo di processi
  - session ID
  - terminale di controllo
  - set-user-ID flag e set-group-ID flag
  - directory corrente
  - directory root
  - maschera di creazione file (umask)
  - maschera dei segnali
  - flag close-on-exec per tutti i descrittori aperti
  - environment

Controllo processi - Creazione

- **Proprietà che il processo child non eredita dal processo parent:**
  - valore di ritorno di **fork**
  - process ID
  - process ID del processo parent
  - file locks
  - l'insieme dei segnali in attesa viene svuotato

Controllo processi - Creazione

- **Ragioni per il fallimento di `fork`**
  - il numero massimo di processi nel sistema è stato raggiunto
  - il numero massimo di processi per user id è stato raggiunto
- **Utilizzo di `fork`**
  - quando un processo vuole duplicare se stesso in modo che parent e child eseguano parti diverse del codice
    - Network servers: il parent resta in attesa di richieste dalla rete; quando una richiesta arriva, viene assegnata ad un child, mentre il parent resta in attesa di richieste
  - quando un processo vuole eseguire un programma diverso (utilizzo della chiamata **exec**)

Controllo processi - Creazione

- **System call: `pid_t vfork();`**
  - Stessa sequenza di chiamata e stessi valori di ritorno di **fork**
  - Semantica differente:
    - **vfork** crea un nuovo processo allo scopo di eseguire un nuovo programma con **exec**()
    - **vfork** non copia l'immagine di memoria del parent; i due processi condividono lo spazio di indirizzamento fino a quando il child esegue **exec**() o **exit**()
    - Guadagno di efficienza
    - **vfork** garantisce che il child esegua per primo:
      - continua ad eseguire fino a quando non esegue **exec**() o **exit**()
      - il parent continua solo dopo una di queste chiamate

Controllo processi - Creazione

- **Esempio: vfork1.c**
  - Differenza tra **vfork** e **fork**
- **Nota:**
  - Incrementare le variabili nel child cambia i valori nell'immagine di memoria del parent
  - Il child chiama una versione speciale di **exit()**
  - Il parent non chiama **wait**, perché il child esegue prima
- **Output:**

```
$ a.out
before vfork
pid = 432, glob = 7, var = 89
```

variabili cambiate

Controllo processi - Terminazione

- **Esistono tre modi per terminare normalmente:**
  - eseguire un return da main; equivalente a chiamare `exit`()
  - chiamare la funzione `exit`:
    - invocazione di tutti gli exit handlers che sono stati registrati
    - chiusura di tutti gli I/O stream standard
    - specificata in ANSI C; non completa per POSIX
  - chiamare la system call `_exit`
    - si occupa dei dettagli POSIX-specific;
    - chiamata da `exit`

Controllo processi - Terminazione

- **Esempio: vfork1.c**
  - Differenza tra `vfork` e `fork` (ancora)
- **Nota:**
  - Cosa succederebbe in caso di chiamata della funzione `exit` invece di `_exit`?
  - `stdout` verrebbe chiuso (sia nel parent che nel child) e quindi la chiamata a `printf` fallirebbe
- **Output:**

```
$ a.out
before vfork
```


Controllo processi - Terminazione

- **Esistono due modi per terminare in modo anormale:**
  - Quando un processo riceve certi segnali
    - Generati dal processo stesso
    - Da altri processi
    - Dal kernel
  - Chiamando `abort()`
    - Questo è un caso speciale del primo, in quanto genera il segnale `SIGABRT`
- **Terminazione normale/anormale**
  - Qualunque sia il modo di terminare del processo, l'effetto per il kernel è lo stesso:
    - rimozione della memoria utilizzata dal processo
    - chiusura dei descrittori aperti, etc.

Controllo processi - Terminazione

- **Exit status:**
  - Il valore che viene passato ad `exit` e `_exit` e che notifica il parent su come è terminato il processo (errori, ok, etc.)
- **Termination status:**
  - Il valore che viene generato dal kernel nel caso di una terminazione normale/anormale
  - Nel caso si parli di terminazione anormale, si specifica la ragione per questa terminazione anormale
- **Come ottenere questi valori?**
  - Tramite le funzioni `wait` e `waitpid` descritte in seguito

Controllo processi - Terminazione

- **Cosa succede se il parent termina prima del child?**
  - si vuole evitare che un processo divenga "orfano"
  - il processo child viene "adottato" dal processo `init` (pid 1)
  - meccanismo: quando un processo termina, si esamina la tabella dei processi per vedere se aveva figli; in caso affermativo, il parent pid viene posto uguale a 1
- **Cosa succede se il child termina prima del parent?**
  - se il child scompare, il parent non avrebbe più modo di ottenere informazioni sul termination/exit status del child
  - per questo motivo, alcune informazioni sul child vengono mantenute in memoria e il processo diventa uno *zombie*

Controllo processi - Terminazione

- **Status zombie**
  - vengono mantenute le informazioni che potrebbero essere richieste dal processo parent tramite `wait` e `waitpid`
    - Process ID
    - Termination status
    - Accounting information (tempo impiegato dal processo)
  - il processo resterà uno zombie fino a quando il parent non eseguirà una delle system call `wait`
- **Child del processo `init`:**
  - non possono diventare zombie
  - tutte le volte che un child di `init` termina, `init` esegue una chiamata `wait` e raccoglie eventuali informazioni
  - in questo modo gli zombie vengono eliminati

Controllo processi - Terminazione

- **Notifica della terminazione di un figlio:**
  - Quando un processo termina (normalmente o no), il parent viene notificato tramite un segnale `SIGCHLD`
  - Notifica asincrona
  - Il parent:
    - Può ignorare il segnale (default)
    - Può fornire un signal handler (una funzione che viene chiamata quando il segnale viene lanciato)
- **System call:**

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

  - Utilizzate per ottenere informazioni sulla terminazione dei processi child

Controllo processi - Terminazione

- **Quando un processo chiama `wait` o `waitpid`:**
  - può bloccarsi
    - Se tutti i suoi child sono ancora in esecuzione
  - può ritornare immediatamente con il termination status di un child
    - Se un child ha terminato ed il suo termination status è in attesa di essere raccolto
  - può ritornare immediatamente con un errore
    - Se il processo non ha alcun child
- **Nota:**
  - se eseguiamo una system call `wait` poiché abbiamo ricevuto `SIGCHLD`, essa termina immediatamente
  - altrimenti può bloccarsi

Controllo processi - Terminazione

- **Significato degli argomenti:**
  - `status` è un puntatore ad un intero; se diverso da `NULL`, il termination status viene messo in questa locazione
  - Il valore di ritorno è il process id del child che ha terminato
- **Differenza tra `wait` e `waitpid`:**
  - `wait` può bloccare il chiamante fino a quando un qualsiasi child non termina;
  - `waitpid` ha delle opzioni per evitare di bloccarsi
  - `waitpid` può mettersi in attesa di uno specifico processo

Controllo processi - Terminazione

- **Il contenuto del termination status dipende dall'implementazione:**
  - bit per la terminazione normale, bit per l'exit status, etc.
- **Standard POSIX:**
  - Macro `WIFEXITED(status)`
    - Ritorna `true` se lo status corrisponde ad un child che ha terminato normalmente. In questo caso, possiamo usare:
  - Macro `WEXITSTATUS(status)`
    - Ritorna l'exit status specificato dal comando
  - Macro `WIFSIGNALED(status)`
    - Ritorna `true` se lo status corrisponde ad un child che ha terminato in modo anormale. In questo caso possiamo usare:

Controllo processi - Terminazione

- **Standard POSIX:**
  - Macro `WTERMSIG(status)`
    - Ritorna il numero di segnale che ha causato la terminazione
  - Macro `WCOREDUMP(status)`
    - Ritorna `true` se un file core è stato generato
    - Non POSIX
  - Macro `WIFSTOPPED(status)`
    - Argomento riguardante il job control
    - Ritorna `true` se il processo è stato stoppato, nel qual caso è possibile utilizzare:
  - Macro `WSTOPSIG(status)`
    - Ritorna il numero di segnale che ha causato lo stop

Controllo processi - Terminazione

- **Esempio: prexit.c, wait1.c**
  - La funzione `pr_exit` utilizza le macro appena descritte per stampare una stringa che descrive il termination status
  - Il programma `main` utilizza questa funzione per dimostrare i valori differenti per il termination status
- **Nota:**
  - Si noti la gestione tramite `ifdef` della variabile `WCOREDUMP`
- **Output**

```
$ a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core dumped)
abnormal termination, signal number = 8 (core dumped)
```

Controllo processi - Terminazione

- **System call:**

```
pid_t waitpid(pid_t pid, int *statalloc, int options);
```

  - Argomento `pid`:
    - `pid == -1` si comporta come `wait`
    - `pid > 0` attende la terminazione del child con process id corrispondente
    - `pid == 0` attende la terminazione di qualsiasi child con process group id uguale a quello del chiamante
    - `pid < -1` attende la terminazione di qualsisi child con process group ID uguale a `–pid`
- **Options:**
  - `WNOHANG` non si blocca se il child non ha terminato

Controllo processi - Terminazione

- **Esempio: fork2.c**
  - illustra come sia possibile evitare di stare in attesa di un processo figlio ed evitare che questo diventi uno zombie
- **Nota:**
  - chiama fork due volte:
    - il parent effettua il primo `fork` e crea il primo child
    - il primo child effettua un secondo `fork` e crea il secondo child
    - il primo child termina
    - il second child viene adottato da `init`
    - il parent aspetta la terminazione del primo child e poi lavora in concorrenza
  - evitare i processi zombie ha il vantaggio di diminuire il numero di entry nella process table

Controllo processi - Esecuzione

- **Quando un processo chiama una delle system call `exec`**
  - il processo viene rimpiazzato completamente da un nuovo programma (text, data, heap, stack vengono sostituiti)
  - il nuovo programma inizia a partire dalla sua funzione main
  - il process id non cambia
- **Esistono sei versioni di `exec`:**
  - con/senza gestione della variabile di ambiente `PATH`
    - Se viene gestita la variabile d'ambiente, un comando corrispondente ad un singolo filename verrà cercato nel `PATH`
  - con variabili di ambiente ereditate / con variabili di ambiente specificate
  - con array di argomenti / con argomenti nella chiamata (`NULL` terminated)

Controllo processi - Esecuzione

- **System call:**

```
int execl(char *pathname, char *arg0, …);

int execv(char *pathname, char *argv[]);

int execle(char *pathname, char *arg0, …, char* envp[]);

int execve(char *pathname, char *argv[] , char* envp[]);

int execlp(char *filename, char *arg0, …);

int execvp(char *filename, char *argv[]);
```

- **Nota:**
  - Normalmente una sola di queste è una system call, le altre sono chiamate di libreria

Controllo processi - Esecuzione

Funzione	pathname	filename	arg list	argv[ ]	environ	envp[ ]
`execl`	•		•		•	
`execlp`		•	•		•	
`execle`	•		•			•
`execv`	•			•	•	
`execvp`		•		•	•	
`execve`	•			•		•
lettere		p	l	v		e

Controllo processi - Esecuzione

- **Cosa viene ereditato da `exec`?**
  - process ID e parent process ID
  - real uid e real gid
  - supplementary gid
  - process group ID
  - session ID
  - terminale di controllo
  - current working directory
  - root directory
  - maschera creazione file (umask)
  - file locks
  - maschera dei segnali
  - segnali in attesa

Controllo processi - Esecuzione

- **Cosa non viene ereditato da `exec`?**
  - effective user id e effective group id
    - vengono settati in base ai valori dei bit di protezione
- **Cosa succede ai file aperti?**
  - Dipende dal flag close-on-exec che è associato ad ogni descrittore
    - Se close-on-exec è true, vengono chiusi
    - Altrimenti, vengono lasciati aperti (comportamento di default)

Controllo processi - Esecuzione

- **Esempio: exec1.c, echoall.c**
  - L'esempio effettua due **`fork`**, eseguendo il programma **`echoall`** che stampa gli argomenti e le variabili di ambiente
  - Nota: i due processi sono in concorrenza
- **Output:**

```
$ a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
argv[0]: echoall
$ argv[1]: only 1 arg
USER=montreso
HOME=/home/montreso
...
```

Controllo processi - Esecuzione

- **Funzione:** `int system(char* command);`
  - Esegue un comando, aspettando la sua terminazione
  - Standard e system:
    - Funzione definita in ANSI C, ma il suo modo di operare è fortemente dipendente dal sistema
    - Non è definita in POSIX.1, perché non è un'interfaccia al sistema operativo
    - Definita in POSIX.2
- **Esempio di implementazione: system.c**
  - Questa implementazione non gestisce correttamente i segnali
  - Nelle prossime lezioni vedremo una versione migliorata

Controllo processi - Proprietà

- **System call:**

```
int setuid(uid_t uid);
int setgid(gid_t gid);
```

  - Cambiano real uid/gid e effective uid/gid
  - Esistono delle regole per cambiare questi id:
    - Se il processo ha privilegi da superutente, la funzione **`setuid`** cambia real uid/ effective uid / saved-set-uid con uid
    - Se il processo non ha privilegi da superutente e uid è uguale a real uid o a saved-set-uid, la funzione **`setuid`** cambia effective uid
    - Se nessuna di queste condizioni è vera, **`errno`** è settato uguale a EPERM e viene ritornato un errore
  - Per quanto riguarda group id, il discorso è identico

Controllo processi - Proprietà

- **Nota**
  - Solo un processo eseguito da root può cambiare il real uid; normalmente, esso viene settato da login e non cambia più
  - L'effective uid viene settato dalle system call **`exec`**, solo se il bit set-user-id è settato per il particolare eseguibile
  - **`setuid`** può essere chiamato in ogni istante per cambiare l'effective uid tra real e saved
  - saved-set-uid viene copiato dal valore di effective-uid al momento di eseguire la **`exec`**

Controllo processi – Race condition

- **Ripasso:**
  - una race condition avviene quando processi multipli cercano di accedere a dati condivisi e il risultato finale dipende dall'ordine di esecuzione dei processi
- **`fork`:**
  - L'utilizzo di una funzione **`fork`** dà origine a problemi di race condition
  - In alcuni casi, il risultato di una esecuzione può dipendere:
    - da chi parte per primo fra parent e child
    - dal carico del sistema
    - dall'algoritmo di scheduling
  - Non è possibile fare previsioni

Controllo processi – Race condition

- **Esempio di race condition: fork2.c**
  - Se il sistema è carico, il secondo figlio può uscire da `sleep` prima che il primo figlio riesca a fare `exit`
- **Risoluzioni del problema?**
  1. fare `waitpid` sul process id del processo padre?
     NO! Si può fare `wait` solo sui processi figli
  2. utilizzare un meccanismo di polling:

```
while (getppid() != 1)
  sleep(1)
```

     NO! Funziona, ma spreca le risorse del sistema
  - utilizzare un meccanismo di sincronizzazione più potente (segnali, pipe, etc)

Controllo processi – Race condition

- **Esempio: tellwait1.c**
  - Un programma che stampa due stringhe, ad opera di due processi diversi
  - L'output dipende dall'ordine in cui i processi vengono eseguiti
- **Esempi di output:**

```
$ a.out
output from child
output from parent
$ a.out
oouuttppuutt ffrroomm cphairledn
t
$ a.out
outoutput from child
put from parent
```


Gestione File

- **Gestione file: generalità**
  - Un file per essere usato deve essere aperto (open)
  - L'operazione open:
    - localizza il file nel file system attraverso il suo pathname
    - copia in memoria il descrittore del file (i-node)
    - associa al file un intero non negativo (file descriptor), che verrà usato nelle operazioni di accesso al file, invece del pathname
  - I file standard non devono essere aperti, perchè sono aperti dalla shell. Sono associati ai file descriptor 0 (input), 1 (output) e 2 (error).
  - La close rende disponibile il file descriptor per ulteriori usi

Gestione File

- **Esempio:**

```
int fd;
...
fd=open(pathname, ...);
if (fd==-1) { /*gestione errore*/ }
...
read(fd, ...);
...
write(fd,...);
...
close(fd);
```

- **Nota:**
  - Un file può essere aperto più volte, e quindi avere più file descriptor associati contemporaneamente

Gestione File

- `int open(const char *path, int oflag, ...);`
  - apre (o crea) il file specificato da **`pathname`** (assoluto o relativo), secondo la modalità specificata in **`oflag`**
  - restituisce il file descriptor con il quale ci si riferirà al file successivamente (o -1 se errore)
- **Valori di oflag**
  - `O_RDONLY` read-only (0)
  - `O_WRONLY` write-only (1)
  - `O_RDWR` read and write (2)
  - Solo una di queste costanti può essere utilizzata in oflag
  - Altre costanti (che vanno aggiunte in or ad una di queste tre) permettono di definire alcuni comportamenti particolari

Gestione File

- **Valori di oflag:**
  - `O_APPEND` append
  - `O_CREAT` creazione del file
  - `O_EXECL` con `O_CREAT`, ritorna un errore se il file esiste
  - `O_TRUNC` se il file esiste, viene svuotato
  - `O_NONBLOCK` file speciali (discusso in seguito)
  - `O_SYNC` synchronous write
- **Se si specifica `O_CREAT`, è necessario specificare anche i permessi iniziali come terzo argomento**
- **La maschera `O_ACCMODE` (uguale a 3) permette di isolare la modalità di accesso**
  - **`oflag & O_ACCMODE`** può essere uguale a **`O_RDONLY`**, **`O_WRONLY`**, oppure **`O_RDWR`**

Gestione File

- `int creat(const char *path, mode_t mode);`
  - crea un nuovo file normale di specificato pathname, e lo apre in scrittura
  - mode specifica i permessi iniziali; l'owner è l'effective user-id del processo
  - se il file esiste già, lo svuota (owner e mode restano invariati)
  - restituisce il file descriptor, o -1 se errore
- **Equivalenze:**

```
creat(path, mode);
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```


Gestione File

- `int close(int filedes);`
  - chiude il file descriptor **`filedes`**
  - restituisce l'esito dell'operazione (0 o -1)
- **Nota:**
  - Quando un processo termina, tutti i suoi file vengono comunque chiusi automaticamente

Gestione File

- **Ogni file aperto:**
  - e' associato ad un "current file offset", la posizione attuale all'interno del file
  - un valore non negativo che misura il numero di byte dall'inizio del file
  - Operazioni **`read`**/**`write`** leggono dalla posizione attuale e incrementano il current file offset in avanti del numero di byte letti o scritti
  - Quando viene aperto un file
    - Il current file descriptor viene posizionato a 0…
    - a meno che l'opzione **`O_APPEND`** non sia specificata

Gestione File

- `off_t lseek(int filedes, off_t offset, int whence);`
  - sposta la posizione corrente nel file **`filedes`** di **`offset`** bytes a partire dalla posizione specificata in **`whence`**:
  - **`SEEK_SET`** dall'inizio del file
  - **`SEEK_CUR`** dalla posizione corrente
  - **`SEEK_END`** dalla fine del file
  - restituisce la posizione corrente dopo la **`lseek`**, o -1 se errore
- **Nota:**
  - La **`lseek`** non effettua alcuna operazione di I/O

Gestione File

- `ssize_t read(int filedes, void *buf, size_t nbyte);`
  - legge in **`*buf`** una sequenza di **`nbyte`** byte dalla posizione corrente del file **`filedes`**
  - aggiorna la posizione corrente
  - restituisce il numero di bytes effettivamente letti, o -1 se errore
  - Esistono un certo numero di casi in cui il numero di byte letti e' inferiore al numero di byte richiesti:
    - Fine di un file regolare
    - Per letture da stream provenienti dalla rete
    - Per letture da terminale
    - etc.

Gestione File

- `ssize_t write(int filedes, const void *buf, size_t nbyte);`
  - scriven in **`*buf`** una sequenza di **`nbyte`** byte dalla posizione corrente del file **`filedes`**
  - aggiorna la posizione corrente
  - restituisce il numero di bytes effettivamente scritti, o -1 se errore

Gestione File

- **Esempio 1 (vedi `seek.c`):**
  - In alcuni file, l'operazione di **`seek`** non è ammessa
  - Esempio d'uso:
    - `a.out < /etc/motd`

      `seek OK`
    - `ls -l | a.out`

      `cannot seek`
- **Esempio 2 (vedi `hole.c`):**
  - Il file offset può essere può essere più grande della dimensione corrente del file, nel qual caso la prossima scrittura estenderà il file (creando un "buco" in mezzo)
  - Tutti i byte non scritti vengono letti come 0

Gestione File

- **Esempio 3 (vedi `mycat.c`)**
  - Semplice programma che copia lo standard input sullo standard output
  - Nota:
    - Si utilizzano **`STDIN_FILENO`** e **`STDOUT_FILENO`** (definiti in **`unistd.h`**) al posto di O e 1
  - Si utilizza una dimensione variabile per il buffer di lettura, contenuta nella costante **`BUFFERSIZE`**
  - Come si può notare (lucido seguente), la dimensione ottimale per il sistema utilizzato era 8192 byte

Gestione File

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	# loops
1	23.8	397.9	423.4	1468802
2	12.3	202.0	215.2	734401
4	6.1	100.6	107.2	367201
8	3.0	50.7	54.0	183601
16	1.5	25.3	27.0	91801
32	0.7	12.8	13.7	45901
64	0.3	6.6	7.0	22950
128	0.2	3.3	3.6	11475
256	0.1	1.8	1.9	5738
512	0.0	1.0	1.1	2869
1024	0.0	0.6	0.6	1435
2048	0.0	0.4	0.4	718
4096	0.0	0.4	0.4	359
8192	0.0	0.3	0.3	180
16384	0.0	0.3	0.3	90
32768	0.0	0.3	0.3	45
65536	0.0	0.3	0.3	23
131072	0.0	0.3	0.3	12

Gestione File – File Sharing

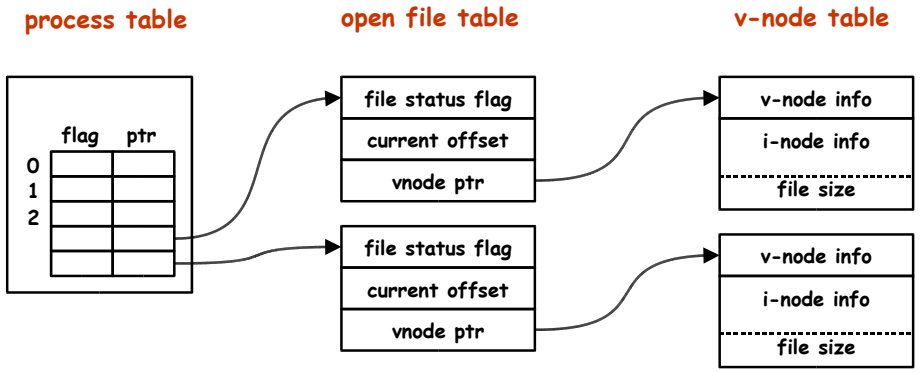
- **File Sharing**
  - Unix supporta la condivisione dei file aperti tra processi differenti
- **Tre strutture dati differenti sono utilizzate dal kernel**
  - Nel process control block di ogni processo è contenuta una tabella di descrittori di file aperti
    - organizzata come vettore, indicizzata dal numero di descrittore
    - Ad ogni entry della tabella sono associati:
      - un flag del descrittore di file
      - un puntatore ad una tabella di file aperti

Gestione File – File Sharing

- **Tre strutture dati differenti sono utilizzate dal kernel**
  - Ogni entry della tabella dei file aperti contiene:
    - il file status flag per il file (lettura, scrittura, append, etc.)
    - il current file offset
    - un puntatore alla tabella dei v-node per il file
  - Ogni entry della tabella dei v-node contiene:
    - Informazioni sul tipo di file
    - Puntatori alle funzioni che operano sul file
    - In molti casi, contengono l'i-node del file (o un puntatore ad esso)

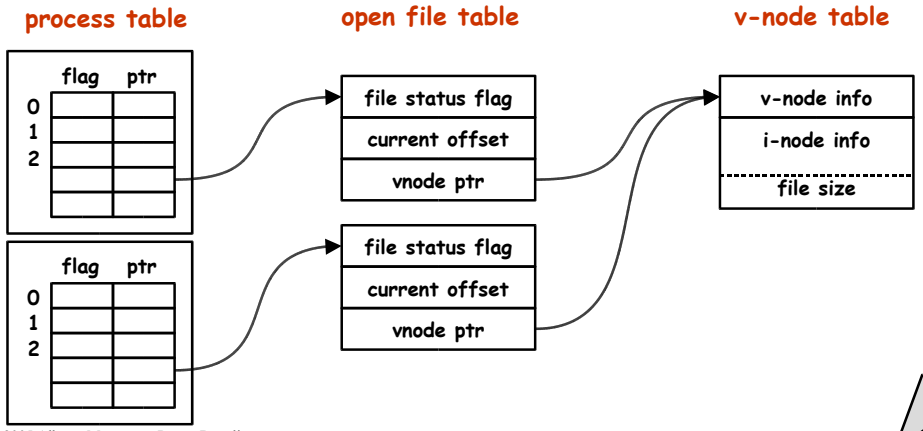
Gestione File – File Sharing

- **Esempio:**
  - Le tre tabelle con un processo che apre due file distinti



Gestione File – File Sharing

- **Esempio:**
  - Le tre tabelle con due processi che aprono lo stesso file



Gestione File – File Sharing

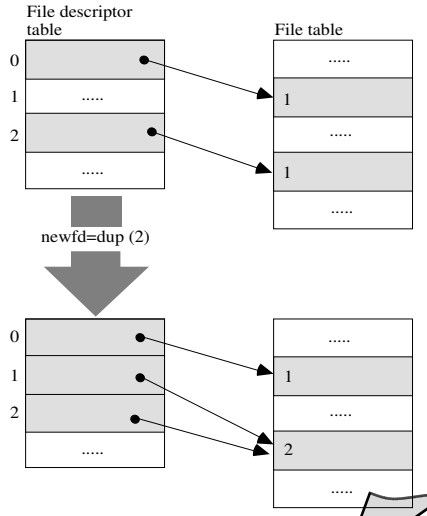
- **Vista questa organizzazione, possiamo essere più specifici riguardo alle operazioni viste in precedenza**
  - Alla conclusione di ogni write
    - il current offset nella file table entry viene incrementato
    - se il current offset è maggiore della dimensione del file nella v-node table entry, questa viene incrementata
  - se il file è aperto con il flag O_APPEND
    - un flag corrispondente è settato nella file table entry
    - ad ogni write, il current offset viene prima posto uguale alla dimensione del file nella v-node table entry
  - lseek modifica unicamente il current offset nella file table entry corrispondente

Gestione File – File Sharing

- **Ulteriori dettagli:**
  - E' possibile che più file descriptor entry siano associate a una singola file table entry
    - tramite la funzione dup, all'interno dello stesso processo
    - tramite fork, tra due processi diversi
  - E' interessante notare che esistono due tipi di flag:
    - alcuni sono associati alla file descriptor entry, e quindi sono particolari del processo
    - altri sono associati alla file table entry, e quindi possono essere condivisi fra più processi
    - Esiste la possibilità di modificare questi flag (funzione fcntl)
  - E' importante notare che questo sistema di strutture dati condivise può portare a problemi di concorrenza

Gestione File – File Sharing

- **Un file descriptor esistente viene duplicato da una delle seguenti funzioni:**
  - `int dup(int filedes);`
  - `int dup2(int filedes, int filedes2);`
- **Entrambe le funzioni:**
  - "duplicano" un file descriptor, ovvero creano un nuovo file descriptor che punta alla stessa file table entry del file descriptor specificato



Gestione File – File Sharing

- **Funzione dup**
  - Seleziona il più basso file descriptor libero della tabella dei file descriptor
  - Assegna la nuova file descriptor entry al file descriptor selezionato
  - Ritorna il file descriptor selezionato
- **Funzione dup2**
  - Con **`dup2`**, specifichiamo il valore del nuovo descrittore come argomento **`filedes2`**
  - Se **`filedes2`** è già aperto, viene chiuso e sostituito con il descrittore duplicato
  - Ritorna il file descriptor selezionato

Gestione File – Funzioni Atomiche

- **Si consideri un singolo processo che vuole inserire dei dati alla fine di un file (operazione append)**
  - Nelle prime versioni di UNIX, il flag **`O_APPEND`** non esisteva
  - Il codice poteva essere scritto nel modo seguente:

```
if (lseek(fd, OL, SEEK_END) < 0)
  err_sys("lseek error");
if (write(fd, buff, 100) != 100)
  err_sys("write error");
```

  - Questo funziona quando abbiamo un solo processo che accede al file
- **Cosa succede se abbiamo due (o più) processi?**

Gestione File - Fcntl

- `int fcntl(int filedes, int cmd, … /* int arg */)`
  - La funzione fcntl può cambiare le proprietà di un file aperto
    - Argomento 1: è il descrittore del file su cui operare
    - Argomento 2: è il comando da eseguire
    - Argomento 3: quando presente, parametro del comando;
      - in generale, un valore intero
      - nel caso di record locking, un puntatore
  - Comandi:
    - duplicazione di file descriptor (**`F_DUPFD`**)
    - get/set file descriptor flag (**`F_GETFD, F_SETFD`**)
    - get/set file status flag (**`F_GETFL, F_SETFL`**)
    - get/set async. I/O ownership (**`F_GETOWN, F_SETOWN`**)
    - get/set record locks (**`F_GETLK, F_SETLK, F_SETLKW`**)

Gestione File - Fcntl

- `int fcntl(int filedes, F_DUPFD, int min)`
  - Duplica il file descriptor specificato da **`filedes`**
  - Ritorna il nuovo file descriptor
  - Il file descriptor scelto è uguale al valore più basso corrispondente ad un file descriptor non aperto e che sia maggiore o uguale a **`min`**
- `int fcntl(int filedes, F_GETFD)`
  - Ritorna i file descriptor flag associati a **`filedes`**
  - Attualmente, è definito un solo file descriptor flag:
    - Se **`FD_CLOEXEC`** è true, il file descriptor viene chiuso eseguendo una **`exec`**

Gestione File - Fcntl

- `int fcntl(int filedes, F_SETFD, int newflag)`
  - Modifica i file descriptor flag associati a **`filedes`**, utilizzando il terzo argomento come nuovo insieme di flag
- `int fcntl(int filedes, F_GETFL)`
  - Ritorna i file status flag associati a **`filedes`**
  - I file status flag sono quelli utilizzati nella funzione **`open`**
  - Per determinare la modalità di accesso, è possibile utilizzare la maschera **`ACC_MODE`**
  - Per determinare gli altri flag, è possibile utilizzare le costanti definite (**`O_APPEND, O_NONBLOCK, O_SYNC`**)

Gestione File - Fcntl

- `int fcntl(int filedes, F_SETFL, int newflag)`
  - Modifica i file status flag associati a **`filedes`** con il valore specificato in **`newflag`**
  - I soli valori che possono essere modificati sono **`O_APPEND, O_NONBLOCK, O_SYNC`**; l'access mode deve rimanere inalterato
- **Gli altri comandi di `fcntl` verranno trattati in seguito**
  - Quando introduciamo il concetto di segnale
  - Quando introduciamo il concetto di locking

Gestione File - Fcntl

- **Esempio (vedi fileflags.c):**
  - Il codice proposto prende un argomento singolo da linea di comando (che specifica un descrittore di file) e stampa una descrizione delle flag del file per quel descrittore
  - Esempio:

```
$ a.out 0 < /dev/tty
read only
$ a.out 1 > temp.txt
$ cat temp.txt
write only
$ a.out 2 2>> temp.txt
write only, append
$ a.out 5 5<> temp.txt
read write
```


Gestione file - Fcntl

- **Esempio:**
  - La funzione `set_fl` mette a 1 i flag specificati
  - Correttamente, richiede prima i flag correnti, utilizza la maschera con or, e salva di nuovi i flag

```
void set_fl(int fd, int flags)
   /* flags are file status flags to turn on */
{
   int val;
   if ( (val = fcntl(fd, F_GETFL, 0)) < 0)
      err_sys("fcntl F_GETFL error");
   val |= flags;         /* turn on flags */
   if (fcntl(fd, F_SETFL, val) < 0)
      err_sys("fcntl F_SETFL error");
}
```


Gestione File - Fcntl

- **E' possibile utilizzare la funzione `set_fl` nel programma che copia lo standard input nello standard output**
- **Mettiamo ad un il flag `O_SYNC`**
  - `set_fl(STDOUT_FILENO, O_SYNC)`
  - Questo richiede che ogni write sia stata completata e scritta su disco prima di ritornare il controllo al chiamante
- **Questo dimostra l'utilità di `fcntl`**
  - I file sono stati aperti dalla shell; non c'era modo di settare il flag `O_SYNC`
- **Tempi:**
  - La richiesta di utilizzare `O_SYNC` allunga i tempi di esecuzione del programma

Gestione File - Ioctl

- **La funzione `ioctl` raccoglie tutti i comportamenti che non possono essere racchiusi nella interfaccia basata su file**
- **`int iocntl(int filedes, int request, …)`**
- **Categorie di comandi**
  - disk labels I/O
  - file I/O
  - magnetic tapes I/O
  - socket I/O
  - terminal I/O

Gestione File - Fsync

- **La system call fsync()**
  - effettua l'operazione di "flush" dei dati bufferizzati dal s.o. per il file descriptor fd, ovvero li scrive sul disco o sul dispositivo sottostante
- **Motivazione**
  - il gestore del file system può mantenere i dati in buffer in memoria per diversi secondi, prima di scriverli su disco
  - per ragioni efficienza
  - ritorna 0 in caso di successo, -1 altrimenti

Gestione File System - Select

- **System call select()**
  - permette ad un processo di aspettare contemporaneamente su file descriptor multipli, con un timeout opzionale
- `int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
- **Parametri di input**
  - *fds sono insiemi di file descriptor, realizzati tramite mappe di bit
  - n è la dimensione massima di questi insiemi
  - timeval è il timeout opzionale
- **Parametri di output**
  - ritorna il numero di file descriptor che sono pronti per operazioni di I/O immediate
  - modifica gli insiemi di wait descriptor, ponendo a 1 i bit relativi ai file descriptor pronti per l'I/O

Gestione File – Funzioni Atomiche

- **Ogni processo possiede una file descriptor entry ed una file table entry "personale"**
- **La v-node `table` entry è condivisa**
- **Si consideri il seguente interleaving tra i due processi:**
  - inizialmente, la dimensione del file sia 1500
  - processo 1 esegue **`lseek`** e pone il suo current offset a 1500
  - processo 2 esegue **`lseek`** e pone il suo current offset a 1500
  - processo 2 esegue **`write`**; scrive i byte 1500-1599 e pone la dimensione del file a 1600
  - processo 1 esegue **`write`**; sovrascrive i byte 1500-1599 e pone la dimensione del file a 1600
- **Soluzione: utilizzate open con `O_APPEND` (write atomica)**

Gestione File – Funzioni Atomiche

- **Si consideri un singolo processo che voglia creare un file, riportando un messaggio di errore se il file è già esistente**
  - Nelle prime versioni di UNIX, i flag **`O_CREAT`** and **`O_EXCL`** non esistevano
  - Il codice poteva essere scritto nel modo seguente:

```
if ( ( fd = open(pathname, O_WRONLY) ) < 0)
  if (errno == ENOENT) {
    if ( (fd = creat(pathname, mode) < 0)
      err_sys("creat error");
  } else
    err_sys("open error");
```

  - Questo funziona quando abbiamo un solo processo che accede al file
- **Cosa succede se abbiamo due (o più) processi?**

Gestione File - Funzioni Atomiche

- **Problema:**
  - Cosa succede se il file viene creato da un altro processo tra l'operazione **`open`** e l'operazione **`creat`**, e questo file scrive delle informazioni su questo file?
  - I dati vengono persi, perché verranno sovrascritti
- **Il problema è evitato dalla system call open con i flag `O_CREAT` e `O_EXCL`**

Gestione File - Funzioni Atomiche

- **Questi due sequenze di codice sono equivalenti:**
  - `dup(filedes)`
  - `fcntl(filedes, F_DUPFD, 0);`
- **Questi due sequenze di codice sono equivalenti?**
  - `dup2(filedes, filedes2)`
  - `close(filedes2);`
    `dup2(filedes, F_DUPFD, filedes2);`
- **Risposta: no!**
  - Possono esserci altri file descriptor liberi
  - Il codice può essere interrotto da un signal catcher, il quale potenzialmente può aprire nuovi file

Esercizio: Come funziona O_APPEND?

- **Domanda:**
  - Se aprite un file for read-write con il flag **`O_APPEND`**:
    - è possibile leggere da qualunque posizione del file utilizzando **`lseek`**?
    - è possibile sovrascrivere dati presenti sul file?
  - Scrivere un programma per verificare

File e Directory

- **open, read, write, close**
  - permettono di leggere e scrivere file regolari
- **Ulteriori operazioni:**
  - leggere gli attributi di un file (stat)
  - modificare gli attributi di un file (chmod, chown, chgrp, …)
  - concetto di hard link (link, unlink, remove)
  - lettura di directory

File e Directory - Stat

- System call `stat`:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat* buf);
int lstat(const char *pathname, struct stat *buf);
```

- Le tre funzioni ritornano un struttura stat contenente informazioni sul file
  - `stat` identifica il file tramite un pathname
  - `fstat` identifica un file aperto tramite il suo descrittore
  - `lstat`, se applicato ad un link simbolico, ritorna informazioni sul link simbolico, non sul file linkato

File e Directory - Stat

- Il secondo argomento delle funzioni `stat` è un puntatore ad una struttura di informazioni sul file specificato

```
struct stat {
   mode_t st_mode;    // File type & mode
   ino_t  st_ino;     // i-node number
   dev_t  st_dev;     // device number (file system)
   dev_t  st_rdev;    // device n. for special files
   nlink_t st_nlink;  // number of links
   uit_t  uid;        // user ID of owner
   gid_t  gid;        // group ID of owner
   off_t  st_size;    // size in bytes, for reg. files
   time_t st_atime;   // time of last access
   time_t st_mtime;   // time of last modif.
   time_t st_ctime;   // time of last status change
   long   st_blksize; // best I/O block size
   long   st_blocks;  // number of 512-byte blocks
}
```


File e Directory – Tipo e accesso

- Tipi di file – Campo `st_mode`
- Macro per determinare il tipo di file
  - File regolari S_ISREG()
  - Directory file S_ISDIR()
  - Character special file S_ISCHR()
    - Device a caratteri (es. `/dev/console`)
  - Block special file S_ISBLK()
    - File speciale a blocchi (es. `/dev/fd0`)
  - FIFO S_ISFIFO()
    - File speciale utilizzato per interprocess communication. Chiamato anche named pipe
  - Socket S_ISSOCK()
    - File utilizzato per comunicazione di rete (e non) fra processi.
  - Link simbolico S_ISLNK()

File e Directory – Tipo e accesso

- Esempio (vedi filetype.c)
  - analizza il campo `st_mode` dei file elencati su linea di comando
  - riporta il tipo
  - esempio:

```
a.out /etc/passwd /etc /dev/ttya /dev/sd0a \ /bin/sh
/etc/passwd: regular
/etc: directory
/dev/ttya: character special
/dev/sd0a: block special
/bin/sh: link special
```

File e directory – Tipo e accesso

- ***set-user-ID e set-group-ID*:**
  - in **`st_mode`**, esiste un flag (*set-user-ID*) che fa in modo che quando questo file viene eseguito,
    effective user id == **`st_uid`**
  - in **`st_mode`**, esiste un flag (*set-group-ID*) che fa in modo che quando questo file viene eseguito,
    effective group id == ***`st_gid`***
- **E' possibile utilizzare questi due bit per risolvere il problema di passwd:**
  - l'owner del comando **`passwd`** è **`root`**
  - quando **`passwd`** viene eseguito, il suo *effective user id* è uguale a **`root`**
  - il comando può modificare in scrittura il file **`/etc/passwd`**

File e directory – Tipo e accesso

- **Costanti per accedere ai diritti di lettura e scrittura contenuti in `st_mode`**
  - **`S_ISUID`** set-user-ID
  - **`S_ISGID`** set-group-ID
  - **`S_IRUSR`** accesso in lettura, owner
  - **`S_IWUSR`** accesso in scrittura, owner
  - **`S_IXUSR`** accesso in esecuzione, owner
  - **`S_IRGRP`** accesso in lettura, gruppo
  - **`S_IWGRP`** accesso in scrittura, gruppo
  - **`S_IXGRP`** accesso in esecuzione, gruppo
  - **`S_IROTH`** accesso in lettura, altri
  - **`S_IWOTH`** accesso in scrittura, altri
  - **`S_IXOTH`** accesso in esecuzione, altri

File e Directory – Tipo e accesso

- **System call:**
  `int access(const char* pathname, int mode)`
  - Quando si accede ad un file, vengono utilizzati effective uid e effective gid
  - A volte, può essere necessario verificare l'accessibilità in base a real uid e real gid
  - Per fare questo, si utilizza access
  - Mode è un maschera ottenuta tramite bitwise or delle seguenti costanti:
    - R_OK test per read permission
    - W_OK test per write permission
    - X_OK test per execute permission

File e Directory – Tipo e accesso

- **System call:**
  `mode_t umask(mode_t cmask);`
  - Cambia la maschera utilizzata per la creazione dei file; ritorna la maschera precedente; nessun caso di errore
  - Per formare la maschera, si utilizzano le costanti S_IRUSR, S_IWUSR viste in precedenza
  - Funzionamento:
    - La maschera viene utilizzata tutte le volte che un processo crea un nuovo file
    - Tutti i bit che sono accesi nella maschera, verranno spenti nell'access mode del file creato

File e Directory – Tipo e accesso

- **System call:**

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

  - Cambia i diritti di un file specificato dal pathname (chmod) o di un file aperto (fchmod)
  - Per cambiare i diritti di un file, l'effective uid del processo deve essere uguale all'owner del file oppure deve essere uguale a root

File e Directory – Proprietario

- **Creazione file**
  - L'access mode del file viene scelto seguendo la modalità illustrata in precedenza
  - Lo user id viene posto uguale allo effective uid del processo
  - POSIX permette due possibilità per il group id:
    - Può essere uguale allo effective gid del processo
    - Può essere uguale al group id della directory in cui il file viene creato
  - In alcuni sistemi, questo può dipendere dal bit set-group-id della directory in cui viene creato il file

File e Directory – Proprietario

- **System call:**

```
int chown(char* pathname, uid_t owner, gid_t group);
int fchown(int filedes, uid_t owner, gid_t group);
int lchown(char* pathname, uid_t owner, gid_t group);
```

  - Queste tre funzioni cambiano lo user id e il group id di un file
    - Nella prima, il file è specificato come pathname
    - Nella seconda, il file aperto è specificato da un file descriptor
    - Nella terza, si cambia il possessore del link simbolico, non del file stesso
  - Restrizioni:
    - In alcuni sistemi, solo il superutente può cambiare l'owner di un file (per evitare problemi di quota)
    - Costante POSIX_CHOWN_RESTRICTED definisce se tale restrizione è in vigore

File e Directory – Dimensioni

- **Dimensione di un file:**
  - il campo **`st_size`** di **`stat`** contiene la dimensione effettiva del file
  - il campo **`st_block`** di **`stat`** contiene il numero di blocchi utilizzati per scrivere il file
    - dimensione "standard" di 512 byte
    - alcune implementazioni usano valori diversi; non portabile
  - il campo **`st_blksize`** di **`stat`** contiene la dimensione preferita per i buffer di lettura/scrittura
    - può essere utilizzato nell'esempio di copia che avevamo visto in precedenza

File e Directory – Dimensioni

- **"Buchi" nei file**
  - la dimensione di un file e il numero di blocchi occupati possono non coincidere
  - questo avviene quando in un file c'è un "buco" creato da lseek
- **Esempio:**

```
$ ls –l core

-rw-r—r– 1 root 8483248 Dec 1 12:20 core

$ du –s core

272 core
```


File e Directory – Dimensioni

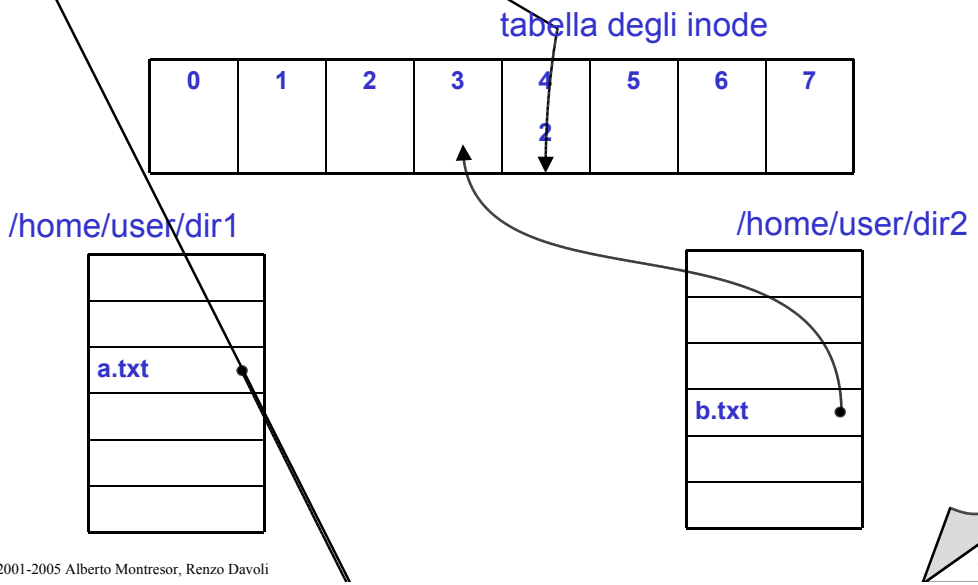
- **System call:**

```
int truncate(char* pathname, off_t len);
int ftruncate(int filedes, off_t len);
```

  - queste due funzioni cambiano la lunghezza di un file, portandola alla dimensione specificata
    - se la nuova dimensione è più corta, tronca il file alla dimensione specificata
    - se la nuova dimensione è più lunga, allunga il file alla dimensione specificata
  - non è standard POSIX

File e Directory – Hard link

- **Concetto di link:**



File e Directory – Hard link

- **Hard link:**
  - Nella figura precedente, sono mostrate due directory entry che puntano allo stesso i-node (due hard link)
  - Ogni nodo mantiene un contatore con il numero di directory entry che puntano ad esso (numero di link)
  - Operazioni:
    - Linking: aggiungere un link ad un file
    - Unlinking: rimuovere un link da un file
  - Solo quando il numero di link scende a zero è possibile rimuovere un link da un file (delete)
  - Nella struttura **`stat`**: il campo **`st_nlink`** contiene il numero di link di un file

File e Directory – Hard link

- **Suddivisione delle informazioni:**
  - gli inode mantengono tutte le informazioni sul file: dimensioni, permessi, puntatori ai blocchi dati, etc.
  - le directory entry mantengono un file name e un inode number
- **Limitazioni degli hard-link:**
  - la directory entry contiene un numero di inode sullo stesso file system
  - non è possibile fare hard-linking con file su file system diversi
  - in questo caso, si utilizzano *symbolic link*

File e Directory – Hard link

- **Hard link per directory:**
  - ogni directory *dir* ha un numero di hard link maggiore uguale a 2:
    - Uno perché *dir* possiede un link nella sua directory genitore
    - Uno perché *dir* possiede un'entry "." che punta a se stessa
  - ogni sottodirectory di *dir* aggiunge un hard link:
    - Uno perché la sottodirectory possiede un'entry ".." che punta alla directory genitore

File e Directory – Hard link

- **System call:**

```
int link(char* oldpath, char* newpath);
```

  - crea un nuovo link ad un file esistente
  - operazione atomica: aggiunge la directory entry e aumenta il numero di link per l'inode identificato da `oldpath`
  - errori:
    - `oldpath` non esiste
    - `newpath` esiste già
    - solo root può creare un hard link ad una directory (per evitare di creare loop, che possono causare problemi)
    - `oldpath` e `newpath` appartengono a file system diversi
  - utilizzato dal comando `ln`

File e Directory – Hard link

- **System call:**

```
int unlink(char* path);
int remove(char* path);
```

  - rimuove un hard link per il file specificato da path
  - operazione atomica: rimuove la directory entry e decrementa di uno il numero di link del file
  - errori:
    - path non esiste
    - un utente diverso da root cerca di fare unlink su una directory
  - utilizzato dal comando `rm`
  - un file può essere effettivamente cancellato dall'hard disk:
    - quando il suo conteggio raggiunge 0

File e Directory – Hard link

- **Unlink: cosa succede quando un file è aperto?**
  - Il file viene rimosso dalla directory
  - Il file non viene rimosso dal file system fino alla chiusura
  - Quando il file viene chiuso, il sistema controlla se il numero di hard link è sceso a zero; in tal caso rimuove il file
- **Esempio:**

```
$ du -s tempfile            9040 tempfile
$ df /home                  Used: 181979
$ a.out &                   file unlinked
$ ls -l tempfile            tempfile not found
$ df /home                  Used: 181979
done
$ df /home                  Used: 172939
```


File e Directory – Hard link

- **Come utilizzare questa proprietà:**
  - per assicurarsi che i file temporanei non vengano lasciati in giro in caso di crash del processo
  - sequenza di operazioni:
    - viene creato un file temporaneo
    - viene aperto dal processo
    - si effettua una operazione di unlink
  - Il file non viene cancellato fino al termine/crash del programma, che causa la chiusura di tutti i file temporanei

File e Directory – Rename

- **System call:**

```
int rename(char* oldpath, char* newpath);
```

  - Cambia il nome di un file da oldpath a newpath
  - Casi:
    - se `oldpath` specifica un file regolare, `newpath` non può essere una directory esistente
    - se `oldpath` specifica un file regolare e `newpath` è un file regolare esistente, questo viene rimosso e sostituito
    - sono necessarie write permission su entrambe le directory
    - se `oldpath` specifica una directory, `newpath` non può essere un file regolare esistente
    - se `oldpath` specifica una directory, `newpath` non può essere una directory non vuota

File e Directory – Link simbolici

- **Un link simbolico:**
  - E' un file speciale che contiene il pathname assoluto di un altro file
  - E' un puntatore indiretto ad un file (a differenza degli hard link che sono puntatori diretti agli inode)
  - Introdotto per superare le limitazioni degli hard link:
    - hard link possibili solo fra file nello stesso file system
    - hard link a directory possibili solo al superuser
  - nota: questo comporta la possibilità di creare loop
    - i programmi che analizzano il file system devono essere in grado di gestire questi loop (esempio in seguito)

File e Directory – Link simbolici

- **Quando si utilizza una funzione, bisogna avere chiaro se:**
  - segue i link simbolici (la funzione si applica al file puntato dal link simbolico)
  - non segue i link simbolici (la funzione si applica sul link)
- **Funzioni che non seguono i link simbolici:**
  - lchown, lstat, readlink, remove, rename, unlink
- **Esempio (tramite shell):**

```
$ ln -s /no/such/file myfile
$ ls myfile
myfile
$ cat myfile
cat: myfile: No such file or directory
$ ls -l myfile
lrwx------ montreso  myfile -> /no/such/file
```


File e Directory – Link simbolici

- **System call:**

```
int symlink(char* oldpath, char* newpath);
```

  - crea una nuova directory entry `newpath` con un link simbolico che punta al file specificato da `oldpath`
  - nota:
    - `oldpath` e `newpath` non devono risiedere necessariamente nello stesso file system
- **System call:**

```
int readlink(char* path, char* buf, int size);
```

  - poiché la open segue il link simbolico, abbiamo bisogno di una system call per ottenere il contenuto del link simbolico
  - questa system call copia in `buf` il valore del link simbolico

File e Directory – Tempi

- **Tre valori temporali sono mantenuti nella struttura stat**
  - `st_atime` (opzione `-u` in `ls`)
    - Ultimo tempo di accesso
    - read, creat/open (when creating a new file), utime
  - `st_mtime` (default in `ls`)
    - Ultimo tempo di modifica del contenuto
    - write, creat/open, truncate, utime
  - `st_ctime` (opzione `-c` in `ls`)
    - Ultimo cambiamento nello stato (nell'inode)
    - chmod, chown, creat, mkdir, open, remove, rename, truncate, link, unlink, utime, write
  - Attenzione ai cambiamenti su contenuto, inode e accesso per quanto riguarda le directory

File e Directory – Tempi

- **System call**

```
int utime(char* pathname,
          struct utimbuf *times);
struct utimbuf {
   time_t actime;
   time_t modtime;
}
```

  - Questa system call modifica il tempo di accesso e di modifica di un file; il tempo di changed-status viene modificato automaticamente al nuovo valore

File e Directory – Directory

- **System call:**
  `int mkdir(char* path, mode_t);`
  - Crea una nuova directory vuota dal path specificato
  - Modalità di accesso:
    - I permessi specificati da mode_t vengono modificati dalla maschera specificata da umask
    - E' necessario specificare i diritti di esecuzione (search)
- **System call:**
  `int rmdir(char* path);`
  - Rimuove la directory vuota specificata da path
  - Se il link count della directory scende a zero, la directory viene effettivamente rimossa; altrimenti si rimuove la directory

File e Directory – Directory

- **Ogni processo:**
  - ha una directory corrente a partire dalla quale vengono fatte le ricerche per i pathname relativi
- **System call:**
  `int chdir(char* path);`
  `int fchdir(int filedes);`
  - Cambia la directory corrente associata al processo, utilizzando un pathname oppure un file descriptor
- **Funzione:**
  `char* getcwd(char* buf, size_t size);`
  - Legge la directory corrente e riporta il pathname assoluto nel buffer specificato da **`buf`** e di dimensione **`size`**

File e Directory – Directory

- **Leggere / scrivere directory**
  - Le directory possono essere lette da chiunque abbia accesso in lettura
  - Solo il kernel può scrivere in una directory
    - Per evitare problemi di consistenza
    - I diritti di scrittura specificano la possibilità di utilizzare uno dei seguenti comandi:
      - **`open, creat, unlink, remove, symlink, mkdir, rmdir`**
  - Il formato del file speciale directory dipende dall'implementazione di Unix
  - Per questo motivo, esistono delle funzioni speciali per leggere il contenuto di una directory

File e Directory – Directory

- **Funzioni:**
  - **`DIR *opendir(const char  *pathname);`**
    - Apre una directory; ritorna un puntatore se tutto è ok, **`NULL`** in caso di errore
  - **`struct dirent *readdir(DIR *dp);`**
    - Ritorna un puntatore ad una struttura dati contenente informazioni sulla prossima entry, **`NULL`** se fine della directory o errore
  - **`void rewinddir(DIR *dp);`**
    - Ritorna all'inizio della directory
  - **`void closedir(DIR *dp);`**
    - Chiude la directory

File e directory – Directory

- La struttura `dirent` è definita in `dirent.h`:

```
struct dirent {
  ino_t d_ino;              // i-node number
  char  d_name[NAME_MAX+1];// null-terminated name
}
```

- Note:
  - la struttura `DIR` è una struttura interna utilizzata da queste quattro funzioni come "directory descriptor"
  - le entry di una directory vengono lette in un ordine non determinato a priori

Segnali - Introduzione

- I segnali sono interrupt software a livello di processo
  - Permetto la gestione di eventi asincroni che interrompono il normale funzionamento di un processo
- Segnali – breve storia
  - Versione "non affidabile" introdotti dalle prime versioni di Unix
    - I segnali potevano essere persi
  - Unix 4.3BSD e SVR3 introducono segnali affidabili
    - Si evita la possibilità che un segnale vada perso
  - POSIX.1 standardizza la gestione dei segnali

Segnali - Introduzione

- Caratteristiche dei segnali
  - Ogni segnale ha un identificatore
    - Identificatori di segnali iniziano con i tre caratteri SIG
    - Es. **`SIGABRT`** è il segnale di abort
  - Numero segnali: 15-40, a seconda della versione di UNIX
    - POSIX: 18
    - Linux: 38
  - I nomi simbolici corrispondono ad un intero positivo
    - Definizioni di costanti in **`signal.h`**
    - Il numero 0 è utilizzato per un caso particolare
- I segnali sono eventi asincroni
  - La gestione avviene tramite signal handler
    - "Se e quando avviene un segnale, esegui questa funzione"

Condizioni che possono generare segnali

- Pressione di tasti speciali sul terminale
  - Es: Premere il tasto **`Ctrl-C`** genera il segnale **`SIGINT`**
- Eccezioni hardware
  - Divisione per 0 (**`SIGFPE`**)
  - Riferimento non valido a memoria (**`SIGSEGV`**)
  - L'interrupt viene generato dall'hardware, e catturato dal kernel; questi invia il segnale al processo in esecuzione
- System call `kill`
  - Permette di spedire un segnale ad un altro processo
  - Limitazione: uid del processo che esegue **`kill`** deve essere lo stesso del processo a cui si spedisce il segnale, oppure deve essere 0 (root)

Condizioni che possono generare un segnale

- **Comando `kill`**
  - Interfaccia shell alla system call **`kill`**
- **Condizioni software**
  - Eventi asincroni generati dal software del sistema operativo, non dall'hardware della macchina
  - Esempi:
    - terminazione di un child (**`SIGCHLD`**)
    - generazione di un alarm (**`SIGALRM`**)

Segnali - Azioni associate

- **Ignorare il segnale**
  - Alcuni segnali che non possono essere ignorati: **`SIGKILL`** e **`SIGSTOP`**
    - Motivo: permettere al superutente di terminare processi
    - Segnali hardware: comportamento non definito in POSIX se ignorati
- **Esecuzione dell'azione di default**
  - Per molti segnali "critici", l'azione di default consiste nel terminare il processo
  - Può essere generato un file di core
  - Casi eccezionali senza core:
    - bit set-user-id e set-group-id settati e uid/gid diversi da owner/group;
    - mancanza di permessi in scrittura per la directory;
    - core file troppo grande

Segnali - Azioni associate

- **Catturare ("catch") il segnale:**
  - Il kernel informa il processo chiamando una funzione specificata dal processo stesso (signal handler)
  - Il signal handler gestisce il problema nel modo più opportuno
- **Esempio:**
  - nel caso del segnale **`SIGCHLD`** (terminazione di un child)
    → possibile azione: eseguire **`waitpid`**
  - nel caso del segnale **`SIGTERM`** (terminazione standard)
    → possibili azioni: rimuovere file temporanei, salvare file

Alcuni dei segnali più importanti

- **`SIGABRT` (Terminazione, core)**
  - Generato da system call **`abort()`**; terminazione anormale
- **`SIGALRM` (Terminazione)**
  - Generato da un timer settato con la system call **`alarm`** o la funzione **`setitimer`**
- **`SIGBUS` (Non POSIX; terminazione, core)**
  - Indica un hardware fault (definito dal s.o.)
- **`SIGCHLD` (Default: ignore)**
  - Quando un processo termina, **`SIGCHLD`** viene spedito al processo parent
  - Il processo parent deve definire un signal handler che chiami **`wait`** o **`waitpid`**
- **`SIGFPE` (Terminazione, core)**
  - Eccezione aritmetica, come divisioni per 0
- **`SIGHUP` (Terminazione)**
  - Inviato ad un processo se il terminale viene disconnesso

Alcuni dei segnali più importanti

- **SIGILL (Terminazione, core)**
  - Generato quando un processo ha eseguito un'azione illegale
- **SIGINT (Terminazione)**
  - Generato quando un processo riceve un carattere di interruzione (`Ctrl-C`) dal terminale
- **SIGIO (Non POSIX; default: terminazione, ignore)**
  - Evento I/O asincrono
- **SIGKILL (Terminazione)**
  - Maniera sicura per uccidere un processo
- **SIGPIPE (Terminazione)**
  - Scrittura su pipe/socket in cui il lettore ha terminato/chiuso
- **SIGSEGV (Terminazione, core)**
  - Generato quando un processo esegue un riferimento di memoria non valido

Alcuni dei segnali più importanti

- **SIGSYS (Terminazione, core)**
  - Invocazione non valida di system call
  - Esempio: parametro non corretto
- **SIGTERM (Terminazione)**
  - Segnale di terminazione normalmente generato dal comando kill
- **SIGURG (Non POSIX; ignora)**
  - Segnala il processo che una condizione urgente è avvenuta (dati out-of-bound ricevuti da una connessione di rete)
- **SIGUSR1, SIGUSR2 (Terminazione)**
  - Segnali non definiti utilizzabili a livello utente
- **SIGSTP (Default: stop process)**
  - Generato quando un processo riceve un carattere di suspend (`Ctrl-Z`) dal terminale

Segnali – Funzione signal

- **Funzione**

```
void (*signal(int signo, void (*func)(int)))(int);
```

- **Descrizione:**
  - `signo`: l'identificatore del segnale che si vuole catturare
  - `func`: l'azione che vogliamo che sia eseguita
    - `SIG_IGN`: ignora il segnale (non applicabile a `SIGKILL` e `SIGSTOP`)
    - `SIG_DFL`: azione di default
    - l'indirizzo del signal handler: quando si vuole catturare il segnale (non applicabile a `SIGKILL` e `SIGSTOP`)
  - valore di ritorno:
    - il valore del precedente signal handler se ok
    - SIG_ERR in caso di errore

Segnali – Funzione signal

- **Funzione**

```
void (*signal(int signo, void (*func)(int)))(int);
```

- **Definizione alternativa:**

```
typedef void sighandler_t(int);

sighandler_t *signal(int, sighandler_t*);
```

- **Definizione delle costanti (tipica) in `signal.h`:**
  - Queste costanti possono essere utilizzate come "puntatori a funzioni che prendono un intero e non ritornano nulla"
  - I valori devono essere tali che non possano essere assegnati a signal handler

```
#define SIG_ERR (void (*)())-1;

#define SIG_DFL (void (*)())0;

#define SIG_IGN (void (*)())1;
```

Segnali - Generazione

- **System call:** `int kill(pid_t pid, int signo);`
  - La funzione **`kill`** spedisce un segnale ad un processo oppure a un gruppo di processi
  - Argomento **`pid`**:
    - **`pid > 0`** spedito al processo identificato da pid
    - **`pid == 0`** spedito a tutti i processi appartenenti allo stesso gruppo del processo che invoca kill
    - **`pid < -1`** spedito al gruppo di processi identificati da –pid
    - **`pid == -1`** non definito
  - Argomento **`signo`**:
    - Numero di segnale spedito

Segnali - Generazione

- **System call:** `int kill(pid_t pid, int signo);`
  - Permessi:
    - Il superutente può spedire segnali a chiunque
    - Altrimenti, il real uid o l'effective uid della sorgente deve essere uguale al real uid o l'effective uid della destinazione
  - POSIX.1 definisce il segnale 0 come il *null signal*
  - Se il segnale spedito è null, **`kill`** esegue i normali meccanismi di controllo errore senza spedire segnali
    - Esempio: verifica dell'esistenza di un processo; spedizione del null signal al processo (nota: i process id vengono riciclati)
- **System call:** `int raise(int signo);`
  - Spedisce il segnale al processo chiamante

Segnali - Generazione

- **System call:** `unsigned int alarm(unsigned int sec);`
  - Questa funzione permette di creare un allarme che verrà generato dopo il numero specificato di secondi
  - Allo scadere del tempo, il segnale **`SIGALRM`** viene generato
  - Attenzione: *il sistema non è real-time*
    - Garantisce che la pausa sarà almeno di **`sec`** secondi
    - Il meccanismo di scheduling può ritardare l'esecuzione di un processo
  - Esiste un unico allarme per processo
    - Se un allarme è già settato, il numero di secondi rimasti prima dello scadere viene ritornato da alarm
    - Se **`sec`** è uguale a zero, l'allarme preesistente viene generato

Segnali - Generazione

- **System call:** `unsigned int alarm(unsigned int sec);`
  - L'azione di default per **`SIGALRM`** è di terminare il processo
  - Ma normalmente viene definito un signal handler per il segnale
- **System call:**
  - **`int getitimer(int which, struct itimerval *value);`**
  - **`int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);`**
  - Permettono un controllo più completo
- **System call:** `int pause();`
  - Questa funzione sospende il processo fino a quando un segnale non viene catturato
  - Ritorna –1 e setta **`errno`** a **`EINTR`**

Segnali - Esempio

- **Esempio: sigusr.c**
  - Cattura i segnali definiti dall'utente e stampa un messaggio di errore
- **Output:**

```
$ a.out &
[1]      235
$ kill –USR1 235   # spedisci segnale SIGUSR1 a 235
received SIGUSR1   # catturato
$ kill –USR2 235   # spedisci segnale SIGUSR1 a 235
received SIGUSR2   # catturato
$ kill 235         # spedisci segnale SIGTERM
[1] + Terminated   a.out &
```


Segnali – System call

- **System call:**
  `unsigned int sleep(unsigned int seconds);`
  - questa system call causa la sospensione del processo fin a quando:
    - l'ammontare di tempo specificato trascorre
      - return value: 0
    - un segnale viene catturato e il signal handler effettua un return
      - return value: tempo rimasto prima del completamento della sleep
  - nota:
    - la sleep può concludersi dopo il tempo richiesto
    - la sleep può essere implementata utilizzando alarm(), ma spesso questo non accade per evitare conflitti

Segnali – System call

- **System call:** `void abort();`
  - questa system call spedisce il segnale **`SIGABRT`** al processo
  - comportamento in caso di:
    - **`SIG_DFL`**: terminazione del processo
    - **`SIG_IGN`**: non ammesso
    - signal handler: il segnale viene catturato
  - nel caso che il segnale venga catturato, il signal handler:
    - può eseguire return
    - può invocare **`exit`** o **`_exit`**
  - in entrambi i casi, il processo viene terminato
  - motivazioni per il catching: cleanup

Segnali - Startup

- **Quando un programma esegue una system call `fork`:**
  - I signal catcher settati nel parent vengono copiati nel figlio
- **Quando un programma viene eseguito tramite `exec`**
  - Se il signal catcher per un certo segnale è default o ignore, viene lasciato inalterato nel child
  - Se il signal catcher è settato ad una particolare funzione, viene cambiato a default nel child
    - Motivazione: la funzione settato può non esistere più nel figlio
- **Casi particolari**
  - Quando un processo viene eseguito in background
    - Segnali **`SIGINT`** e **`SIGQUIT`** vengono settati a ignore
    - In che momento / da chi questa operazione viene effettuata?

Segnali – Funzioni reentrant

- **Quando un segnale viene catturato**
  - la normale sequenza di istruzioni viene interrotta
  - vengono eseguite le istruzioni del signal handler
  - quando il signal handler ritorna (invece di chiamare `exit`) la normale sequenza di istruzioni viene ripresa
- **Problemi:**
  - Cosa succede se un segnale viene catturato durante l'esecuzione di una `malloc` (che gestisce lo heap), e il signal handler invoca una chiamata a `malloc`?
  - In generale, può succedere di tutto…
  - Normalmente, ciò che accade è un segmentation fault…

Segnali – Funzioni reentrant

- **POSIX.1 garantisce che un certo numero di funzioni siano reentrant**
  - `_exit`, `access`, `alarm`, `chdir`,`chmod`, `chown`, `close`, `creat`, `dup`, `dup2`, `execle`, `execve`, `exit`, `fcntl`, `fork`, `fstat`, `get*id`, `kill`, `link`, `lseek`, `mkdir`, `mkfifo`, `open`, `pathconf`, `pause`, `pipe`, `read`, `rename`, `rmdir`, `set*id`, `sig`*, `sleep`, `stat`, `sysconf`, `time`, `times`, `umask`, `uname`, `unlink`, `utime`, `wait`, `waitpid`, `write`
- **Se una funzione manca…**
  - perché utilizza strutture dati statiche
  - perché chiama `malloc` e `free`
  - perché fa parte della libreria standard di I/O

Segnali – Funzioni reentrant

- **In ogni caso:**
  - Le funzioni reentrant listate in precedenza possono modificare la variabile `errno`
  - Un signal handler che chiama una di quelle funzioni dovrebbe salvare il valore di `errno` prima della funzione e ripristinarlo dopo
- **Esempio: reenter.c**
  - Utilizzo (errato) di `printf` nel signal handler
  - Utilizzo (errato) di `getpwnam` (analizza il `passwd` file)
  - Può generare segmentation fault

Segnali – Standard POSIX

- **Nelle prime versioni di UNIX**
  - I segnali non erano affidabili
    - Potevano andare persi (un segnale viene lanciato senza che un processo ne sia al corrente
    - Problema derivante in parte dal fatto che una volta catturato, il signal catcher deve essere ristabilito

```
signal(SIGINT, sig_int);
void sig_int() {
  signal(SIGINT, sig_int);
      /* process the signal */
}
```

  - Race condition tra `signal` la gestione del segnale e la signal all'interno di `sig_int()`

Segnali – Standard POSIX

- **Nelle prime versioni di UNIX**
  - I processi non avevano controllo sui segnali
    - Possono ignorarli o catturarli, ma non è possibile bloccarli temporaneamente per gestirli successivamente
    - Una soluzione sbagliata…
    -

```
int sig_int_flag;
int main() {
  signal(SIGINT, sig_int);
  while (sig_int_flag == 0)
        pause();
}
void sig_int() {
  signal(SIGINT, sig_int);
  sig_int_flag == 1
}
```


Segnali – Standard POSIX

- **Cosa succede se un processo riceve un segnale durante una system call?**
  - normalmente, l'eventuale azione associata viene eseguita solo dopo la terminazione della system call
  - in alcune system call "lente", le prime versioni di Unix potevano interrompere la system call
    - la quale ritornava –1 come errore e errno viene settata a EINTR
- **Motivazioni:**
  - in assenza di interruzioni da segnali:
    - una lettura da terminale resta bloccata per lunghi periodi di tempo
    - un segnale di interruzione non verrebbe mai consegnato
  - poiché il processo ha catturato un segnale, c'è una buona probabilità che sia successo qualcosa di significativo

Segnali – Standard POSIX

- **System call "lente":**
  - operazioni **`read`** su file che possono bloccare il chiamante per un tempo indeterminato (terminali, pipe, connessioni di rete)
  - operazione **`write`** su file che possono bloccare il chiamante per un tempo indeterminato prima di accettare dati
  - **`pause`**, **`wait`**, **`waitpid`**
  - certe operazioni **`ioctl`**
  - alcune system call per la comunicazione tra processi
- **Problemi:**
  - bisognerebbe gestire esplicitamente l'errore dato dalle interruzioni

Segnali – Standard POSIX

- **Esempio gestione:**

```
while ( (n= read(fd, buff, BUFFSIZE)) < 0) {
  if (errno != EINTR)
    break;
}
```

- **Restart automatico di alcune system call:**
  - Alcune system call possono ripartire in modo automatico:
    - per evitare questa gestione
    - perché in alcuni casi non è dato sapere se il file su cui si opera può bloccarsi indefinitamente
  - System call con restart: **`ioctl`**, **`read`**, **`write`**
    - solo quando operano su file che possono bloccarsi indefinitamente
  - System call senza restart: **`wait`**, **`waitpid`**
    - sempre

Segnali – Standard POSIX

- **POSIX e S.O. moderni:**
  - Capacità di bloccare le system call: standard
  - I signal handler rimangono installati: standard
  - Restart automatico delle system call: non specificato
    - In realtà, in molti S.O. moderni è possibile specificare se si desidera il restart automatico oppure no
- **POSIX specifica un meccanismo per segnali affidabili:**
  - E' possibile gestire ogni singolo dettaglio del meccanismo dei segnali
    - quali bloccare
    - quali gestire
    - come evitare di perderli, etc.

Segnali affidabili

- **Alcune definizioni:**
  - Diciamo che un segnale è *generato* per un processo quando accade l'evento associato al segnale
    - Esempio: riferimento memoria non valido ⇒ `SIGSEGV`
    - Quando il segnale viene generato, viene settato un flag nel process control block del processo
  - Diciamo che un segnale è *consegnato* ad un processo quando l'azione associata al segnale viene intrapresa
  - Diciamo che un segnale è *pendente* nell'intervallo di tempo che intercorre tra la generazione del segnale e la consegna

Segnali affidabili

- **Bloccare i segnali**
  - Un processo ha l'opzione di bloccare la consegna di un segnale per cui l'azione di default non è *ignore*
  - Se un segnale bloccato viene generato per un processo, il segnale rimane pending fino a quando:
    - il processo sblocca il segnale
    - il processo cambia l'azione associata al segnale ad ignore
  - E' possibile ottenere la lista dei segnali pending tramite la funzione `sigpending`
- **Cosa succede se un segnale bloccato viene generato più volte prima che il processo sblocchi il segnale?**
  - POSIX non specifica se i segnali debbano essere accodati oppure se vengano consegnati una volta sola

Segnali affidabili

- **Cosa succede se segnali diversi sono pronti per essere consegnati ad un processo?**
  - POSIX non specifica l'ordine in cui devono essere consegnati
  - POSIX suggerisce che segnali importanti (come `SIGSEGV`) siano consegnati prima di altri
- **Maschera dei segnali:**
  - Ogni processo ha una maschera di segnali che specifica quali segnali sono attualmente bloccati
  - E' possibile pensare a questa maschera come ad un valore numerico con un bit per ognuno dei possibili segnali
  - E' possibile esaminare la propria maschera utilizzando la system call `sigprocmask`

Segnali affidabili – Implementazione di sleep

- **Esempio: sleep1.c**
- **Problemi:**
  - Se il chiamante ha un allarme settato, **`sleep`** cancella questo allarme
    - E' necessario utilizzare il valore di ritorno di alarm
      - l'allarme precedente può scadere prima
      - l'allarme precedente può scadere dopo
  - Modifica l'azione associata a SIGALRM
    - Bisogna salvare questa azione da qualche parte, per poi ripristinarla
  - Race condition:
    - Fra la chiamata di **`alarm`** e la chiamata a **`pause`**
    - Se questo accade, il chiamante è sospeso per sempre

Segnali affidabili – Utilizzo di alarm

- **Esempio: read1.c**
  - Un utilizzo comune di **`alarm`**, è quello di descrivere un limite superiore alle operazioni che possono bloccarsi
- **Problemi:**
  - Ancora una volta, c'è una race condition tra la prima chiamata di **`alarm`** e la funzione **`read`**
    - La funzione può bloccarsi indefinitamente
    - Spesso il timeout è molto grande, e quindi non ci sono problemi; in ogni caso, è una race condition
  - Se le system call vengono fatte ripartire automaticamente, la **`read`** non viene interrotta
    - In questo caso il timeout non esegue nulla

Segnali affidabili

- **Gli esempi precedenti:**
  - Mostrano come la gestione dei segnali non può essere fatta in modo ingenuo
  - Nel seguito, mostreremo delle soluzioni a questi problemi

Segnali affidabili – Segnali multipli

- **Signal set**
  - Abbiamo bisogno di un tipo di dati per rappresentare segnali multipli
    - necessario in funzioni tipo **`sigprocmask`**
    - implementazione dipendente dal s.o., e in particolare dal numero di segnali definiti
  - Il tipo **`sigset_t`** è definito in **`signal.h`**
- **Operazioni su `sigset_t`:**
  - **`int sigemptyset(sigset_t *set);`**
    - Inizializza la struttura dati puntata da **`set`** ad un insieme vuoto
  - **`int sigfillset(sigset_t *set);`**
    - Inizializza la struttura dati puntata da **`set`** ad un insieme che contiene tutti i segnali

Segnali affidabili – Segnali multipli

- **Operazioni su `sigset_t`:**
  - `int sigaddset(sigset_t *set, int signo);`
    - Aggiunge il segnale `signo` all'insieme puntata da `set`
  - `int sigdelset(sigset_t *set, int signo);`
    - Rimuove il segnale `signo` dalla struttura dati puntata da `set`
  - `int sigismember(sigset_t *set, int signo);`
    - Ritorna true se il segnale `signo` appartiene alla struttura dati puntata da `set`
- **Implementazione:**
  - In molte implementazioni, 31 segnali + null signal vengono collocati in un intero a 32 bit
  - Le varie operazioni si riducono ad accendere e spegnere bit in questo intero

Segnali affidabili – Gestione segnali

- **System call**
  `int sigprocmask(int how, sigset_t *set, sigset_t *oset);`
- **Descrizione:**
  - Argomento `oset`:
    - Se è diverso da `NULL`, al termine della chiamata questa struttura dati conterrà la maschera precedente
  - Argomento `set`:
    - Se è diverso da `NULL`, allora l'argomento `how` descrive come la maschera viene modificata
  - Argomento `how`
    - `SIG_BLOCK`: blocca i segnali compresi in `set` (bitwise or)
    - `SIG_UNBLOCK`: sblocca i segnali compresi in `set`
    - `SIG_SETMASK`: `set` è la nuova maschera

Segnali affidabili – Gestione segnali

- **Esempio:**
  - Stampa parte del contenuto della maschera dei segnali
  - Può essere richiamato da un signal handler (salvataggio `errno`)

```
void pr_mask() {
   sigset_t  sigset;
   int errno_save;
   errno_save = errno;
   if (sigprocmask(0, NULL, &sigset) < 0)
       perror("sigprocmask error");
   if (sigismember(&sigset, SIGINT))  printf("SIGINT ");
   if (sigismember(&sigset, SIGQUIT)) printf("SIGQUIT ");
       /* remaining signals can go here */
   printf("\n");
   errno = errno_save;
}
```


Segnali affidabili – Gestione segnali

- **System call:** `int sigpending(sigset_t *set);`
- **Descrizione:**
  - Ritorna l'insieme di segnali che sono attualmente pending per il processo corrente
- **Esempio:**

```
void pr_mask() {
   sigset_t  sigset;
   int errno_save = errno;
   if (sigpending(&sigset) < 0)
       perror("sigpending error");
   if (sigismember(&sigset, SIGINT))  printf("SIGINT ");
   if (sigismember(&sigset, SIGQUIT)) printf("SIGQUIT ");
       /* remaining signals can go here */
   printf("\n");
   errno = errno_save;
}
```

Segnali affidabili – Gestione segnali

- **Esempio: critical.c;**
  - viene bloccato **SIGINT**, salvando la maschera corrente
  - si esegue **sleep** per 5 secondi
  - al termine dello **sleep**, si verifica se **SIGINT** è pendente
  - sblocca **SIGINT** tornando alla maschera precedente
  - nella gestione di **SIGINT**:
    - ripristina l'azione di default

- **Output (1):**

```
$ a.out
^C
SIGINT pending
caught SIGINT
SIGINT unblocked
^C
$
```

- **Output (2):**

```
$ a.out
^C ^C ^C ^C ^C
SIGINT pending
caught SIGINT
SIGINT unblocked
^C
$
```


Segnali affidabili – Gestione segnali

- **System call**

```
int sigaction(int signo, struct sigaction
        *newact, struct sigaction *oldact);
```

  - Questa system call permette di esaminare e/o modificare l'azione associata ad un segnale
  - Versione più completa di **signal**; nei sistemi moderni, **signal** è implementata utilizzando **sigaction**
  - Argomento:
    - **signo** segnale considerato
    - **newact** se diverso da **NULL**, struttura dati contenente informazioni sulla nuova azione
    - **oldact** se diverso da **NULL**, struttura dati contenente informazioni sulla vecchia azione

Segnali affidabili – Gestione segnali

- **Struttura sigaction:**

```
struct sigaction {
   void (*sa_handler)();     /* signal handler */
   sigset_t sa_mask;         /* addit.block mask */
   int sa_flags;             /* options */
}
```

- **Descrizione:**
  - **sa_handler** è il puntatore all'azione per il segnale (un signal handler, **SIG_IGN** o **SIG_DFL**)
  - **sa_mask** è un insieme addizionale di segnali da bloccare quando un segnale viene catturato da un signal handler
  - **sa_flags** descrive flag addizionali per vincolare il comportamento del sistema

Segnali affidabili – Gestione segnali

- **Utilizzo di sa_mask**
  - All'inizio dell'esecuzione di un signal handler:
    - il valore corrente della **procmask** viene salvato
    - alla **procmask** vengono aggiunti
      - i segnali specificati in **sa_mask**
      - il segnale specificato da **signo**
  - Al termine dell'esecuzione di un signal handler:
    - la **procmask** viene ripristinata al valore salvato
- **Alcuni valori per sa_flag (non standard POSIX):**
  - **SA_RESTART** forza automatic restart per system call interrotte da questo segnale
  - **SA_INTERRUPT** elimina automatic restart per system call interrotte da questo segnale

Segnali affidabili – Gestione segnali

- **Esempio: signal.c**
  - Come implementare **`signal`** a partire da **`sigaction`**
  - Si noti
    - l'uso particolare di **`SA_INTERRUPT`** per **`SIGALRM`**
    - l'uso particolare di **`SA_RESTART`** per tutti gli altri
    - l'inizializzazione esplicita di **`sa_mask`** tramite **`sigemptyset`**

Segnali affidabili – Gestione segnali

- **E' possibile utilizzare `sigaction` e `sigprocmask`**
  - per proteggere sezioni di codice che non devono essere interrotte da un segnale
- **Attenzione a quando si sblocca il segnale:**
  - Se si desidera effettuare un'operazione **`pause`**, si va incontro a problemi di race condition
  - Esempio:

```
sigprocmask(SIG_SETMASK, &oldmask, NULL);
pause();
```

- **Per ovviare al problema:**
  - è necessaria un'operazione atomica che cambi la maschera ed effettui l'operazione **`pause`**

Segnali affidabili – Gestione segnali

- **System call:**

```
int sigsuspend(sigset_t *sigmask);
```

  - La procmask viene posta uguale al valore puntato da **`sigmask`**
  - Il processo è sospeso fino a quando:
    - un segnale viene catturato
    - un segnale causa la terminazione di un processo
  - Ritorna sempre –1 con **`errno`** uguale a **`EINTR`**

Segnali affidabili – Gestione segnali

- **Esempio: suspend2.c**
  - Attesa che un signal handler cambi il valore di una variabile globale
    - Cattura sia **`SIGINT`** che **`SIGQUIT`**
    - Termina solo con **`SIGQUIT`**
  - Si noti:
    - L'utilizzo del modificatore **`volatile`** per indicare che la variabile non deve essere mantenuta in un registro
    - L'utilizzo del tipo **`sig_atomic_t`** per la variabile **`quitflag`**
    - Non necessario in POSIX
    - In ANSI C, garantisce che l'aggiornamento della variabile verrà eseguito in modo atomico

Sincronizzazione tra processi

- **Esempio: tellwait1.c**
  - Un programma che stampa due stringhe, ad opera di due processi diversi
  - L'output dipende dall'ordine in cui i processi vengono eseguiti
- **Vogliamo sincronizzare i due processi:**
  - Prima stampa il padre, poi il figlio
- **Esempio: tellwait2.c**
  - Funzione TELL_WAIT:
    - inizializza strutture dati per la sincronizzazione
  - Funzione WAIT_PARENT:
    - attendi sincronizzazione dal parent
  - Funzione TELL_CHILD
    - invia segnale di sincronizzazione al client

Sincronizzazione tra processi

- **Esempio: tellwait.c**
  - Implementazione delle funzioni di sincronizzazione utilizzate nel programma precedente
  - Basate sullo scambio di segnali **`SIGUSR1`** e **`SIGUSR2`**
  - Funzione **`TELL_WAIT`** inizializza il processo, bloccando la ricezione di **`SIGUSR1`** e **`SIGUSR2`**
  - Le funzioni **`TELL_`**
    - utilizzano la system call **`kill`**
  - Le funzioni **`WAIT_`**
    - utilizzano le system call **`sigsuspend`**

Esercizio: Sincronizzazione tramite segnali

- **Descrizione**
  - realizzare un meccanismo basato su segnali per risolvere il problema produttore / consumatore
    - processo padre: produttore
      - produce numeri interi consecutivi
      - stampa il proprio pid e il numero prodotto
    - processo figlio: consumatore,
      - consuma i numeri prodotti dal padre
      - stampa il proprio pid e il numero consumato
  - il processo padre e figlio utilizzano un buffer di dimensione 1, realizzato con un file condiviso

Esercizio: Sincronizzazione tramite segnali

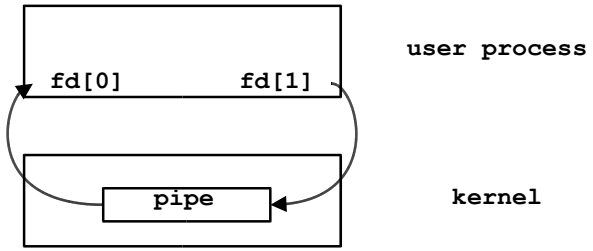
- **Descrizione**
  - processo padre:
    - apre il file condiviso in scrittura
    - utilizza lseek per scrivere sempre all'inizio
    - scrive i 4 byte che rappresentano un int
  - processo figlio:
    - apre il file condiviso in lettura
    - utilizza lseek per leggere sempre all'inizio
    - legge i 4 byte che rappresentano un int

Pipe

- **Cos'è un pipe?**
  - E' un canale di comunicazione che unisce due processi
- **Caratteristiche:**
  - La più vecchia e la più usata forma di interprocess communication utilizzata in Unix
  - Limitazioni
    - Sono half-duplex (comunicazione in un solo senso)
    - Utilizzabili solo tra processi con un "antenato" in comune
  - Come superare queste limitazioni?
    - Gli *stream pipe* sono full-duplex
    - *FIFO* (*named pipe*) possono essere utilizzati tra più processi
    - *named stream pipe* = stream pipe + FIFO

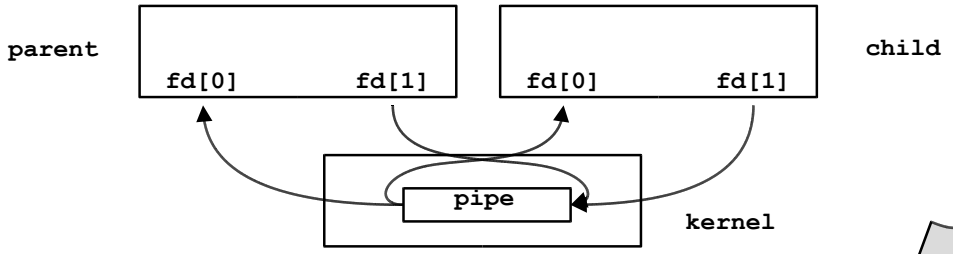
Pipe

- **System call:** `int pipe(int filedes[2]);`
  - Ritorna due descrittori di file attraverso l'argomento `filedes`
    - `filedes[0]` è aperto in lettura
    - `filedes[1]` è aperto in scrittura
  - L'output di `filedes[1]` è l'input di `filedes[0]`



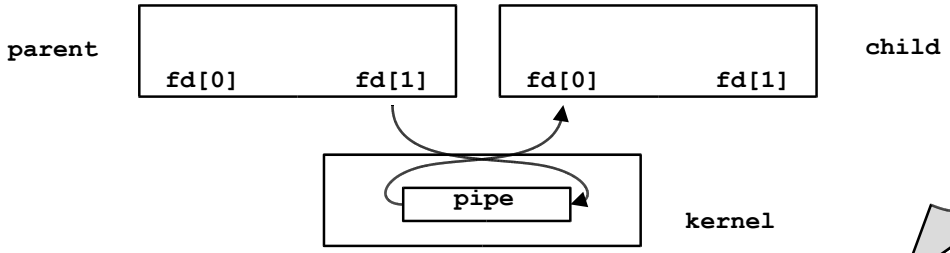
Pipe

- **Come utilizzare i pipe?**
  - I pipe in un singolo processo sono completamente inutili
  - Normalmente:
    - il processo che chiama pipe chiama `fork`
    - i descrittori vengono duplicati e creano un canale di comunicazione tra parent e child o viceversa



Pipe

- **Come utilizzare i pipe?**
  - Cosa succede dopo la `fork` dipende dalla direzione dei dati
  - I canali non utilizzati vanno chiusi
- **Esempio: parent → child**
  - Il parent chiude l'estremo di input (`close(fd[0]);`)
  - Il child chiude l'estremo di output (`close(fd[1]);`)



Pipe

- **Come utilizzare i pipe?**
  - Una volta creati, è possibile utilizzare le normali chiamate **`read`**/**`write`** sugli estremi
- **La chiamata `read`**
  - se l'estremo di ouput è aperto
    - restituisce i dati disponibili, ritornando il numero di byte
    - successive chiamate si bloccano fino a quando nuovi dati non saranno disponibili
  - se l'estremo di output è stato chiuso
    - restituisce i dati disponibili, ritornando il numero di byte
    - successive chiamate ritornano 0

Pipe

- **La chiamata `write`**
  - se l'estremo di input è aperto
    - i dati in scrittura vengono bufferizzati fino a quando non saranno letti dall'altro processo
  - se l'estremo di input è stato chiuso
    - viene generato un segnale **`SIGPIPE`**
      - ignorato/catturato: write ritorna **`-1`** e **`errno=EPIPE`**
      - azione di default: terminazione
- **Esempio: pipe1.c**
  - Due processi: parent e child
  - Il processo parent comunica al figlio una stringa, e questi provvede a stamparla

Pipe

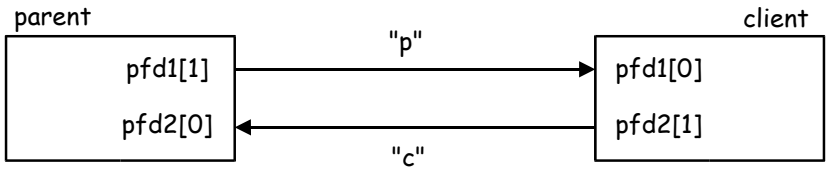
- **Chiamata `fstat`**
  - Se utilizziamo **`fstat`** su un descrittore aperto su un pipe, il tipo del file sarà descritto come fifo (macro **`S_ISFIFO`**)
- **Atomicità**
  - Quando si scrive su un pipe, la costante **`PIPE_BUF`** specifica la dimensione del buffer del pipe
  - Chiamate **`write`** di dimensione inferiore a **`PIPE_BUF`** vengono eseguite in modo atomico
  - Chiamate **`write`** di dimensione superiore a **`PIPE_BUF`** possono essere eseguite in modo non atomico
    - La presenza di scrittori multipli può causare interleaving tra chiamate **`write`** distinte

Pipe

- **Esempio: pipe2.c**
  - Consideriamo un programma che vuole mostrare il proprio output una pagina alla volta
    - Aggiungere al programma la funzionalità di more, less
    - Scrivere uno script che metta il programma in pipe con more
    - Scrivere un programma che crei un pipe con more
  - Procedura
    - si crea un pipe
    - si chiama fork
    - si chiudono nel padre e nel figlio gli opportuni descrittori
    - il figlio usa dup2 per duplicare l'estremo di input del pipe sullo standard input
      - attenzione; bisogna verificare che il descrittore non abbia già il valore scelto
    - Nota: cerca in una variabile di ambiente PAGER

Pipe per sincronizzazione

- **Esempio già visto: tellwait2.c**
  - Un programma che stampa due stringhe, ad opera di due processi diversi
  - Basato su funzioni tell, wait
- **Implementazione basata su pipe: tellwaitpipe.c**
  - Vengono aperte due pipe (TELL_WAIT)
  - Funzioni TELL: emettono un carattere "p", "c"
  - Funzion WAIT: consumano un carattere "p", "c"

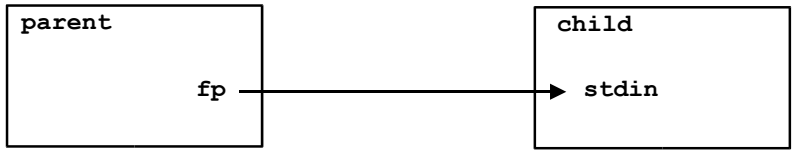


Pipe - popen

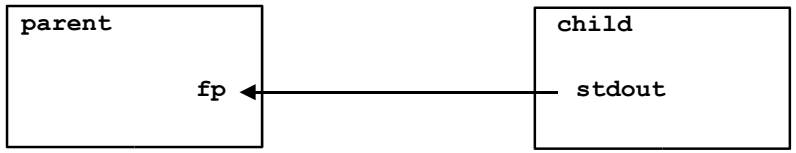
- **Funzioni:**
```
FILE *popen(char *cmdstring, char *type);
int pclose(FILE *fp);
```
- **Descrizione di `popen`:**
  - crea un nuovo processo per eseguire **`cmdstring`**
  - crea un pipe con questo processo
  - chiude le parti non usate dei pipe
  - redireziona
    - lo standard output del nuovo processo sul pipe **`(type = "r")`**
    - lo standard input dal nuovo processo sul pipe **`(type = "w")`**
- **Descrizione di `pclose`**
  - Attende con **`wait`** la terminazione del comando
  - Restituisce l'exit status

Pipe - popen

- **`type = "w"`**



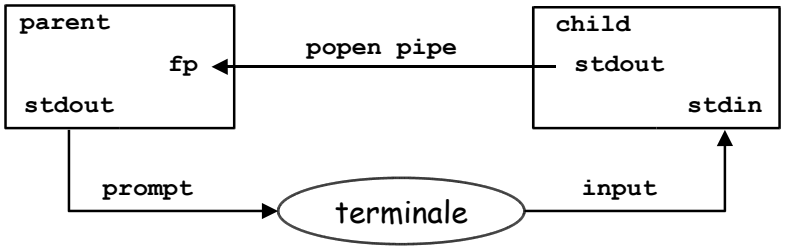
- **`type = "r"`**



- **Nota: `cmdstring` è eseguita tramite `"/bin/sh -c"`**

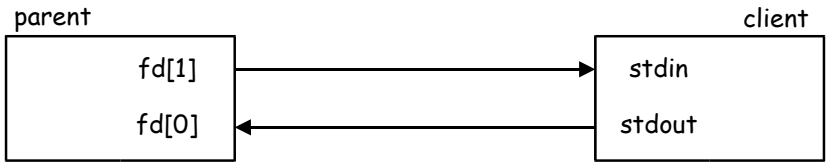
Pipe - popen

- **Esempio: popen2.c**
  - Rielaborazione del programma precedente utilizzando la funzione popen
- **Esempio: popen1.c**
  - Si consideri un'applicazione che scrive un prompt su standard output e legge una linea da standard input
  - Vogliamo inframezzare un "filtro" sullo standard input



Ulteriore esempio - Coprocessi

- **Cos'è un coprocesso?**
  - Un filtro UNIX è un processo che legge da stdin e scrive su stdout
  - Normalmente i filtri UNIX sono connessi linearmente in una pipeline
  - Se lo stesso processo legge da stdin e scrive su stdout si parla di coprocesso



Ulteriore esempio - Coprocessi

- **Esempio: add2.c, pipe4.c**
  - Il primo programma prende due numeri dallo stdin e lo scrive su stdout
  - Il secondo programma
    - legge da stdin una stringa
    - la passa allo stdin del coprocesso
    - stampa l'output ottenuto dal coprocesso
  - Nota:
    - Perché **`add2.c`** utilizza le system call **`read`**, **`write`**, invece delle funzioni di **`stdio`**?
    - Perché **`stdio`**, se **`stdin`** non è un terminale, utilizza buffer di grandi dimensione
    - **`setvbuf(stdin, NULL, _IOLBF, 0);`** per line-buffering

Named pipe

- **Pipe "normali"**
  - possono essere utilizzate solo da processi che hanno un "antenato" in comune
  - motivo: unico modo per ereditare descrittori di file
- **Named pipe**
  - permette a processi non collegati di comunicare
  - utilizza il file system per "dare un nome" al pipe
  - chiamate **`stat, lstat`**
    - Utilizzando queste chiamate su pathname che corrisponde ad un fifo, la macro **`S_ISFIFO`** restituirà **`true`**)
  - la procedura per creare un fifo è simile alla procedura per creare file

Named pipe - FIFO

- **System call:**
  `int mkfifo(char* pathname, mode_t mode);`
  - crea un FIFO dal **`pathname`** specificato
  - la specifica dell'argomento **`mode`** è identica a quella di **`open`** **`(O_RDONLY, O_WRONLY, O_RDWR,`** **etc)**
- **Come funziona un FIFO?**
  - una volta creato un FIFO, le normali chiamate **`open`**, **`read`**, **`write`**, **`close`**, possono essere utilizzate per leggere il FIFO
  - il FIFO può essere rimosso utilizzando **`unlink`**
  - le regole per i diritti di accesso si applicano come se fosse un file normale

Named pipe - FIFO

- **Chiamata `open`:**
  - File aperto senza flag O_NONBLOCK
    - Se il file è aperto in lettura, la chiamata si blocca fino a quando un altro processo non apre il FIFO in scrittura
    - Se il file è aperto in scrittura, la chiamata si blocca fino a quando un altro processo non apre il FIFO in lettura
  - File aperto con flag O_NONBLOCK
    - Se il file è aperto in lettura, la chiamata ritorna immediatamente
    - Se il file è aperto in scrittura, e nessun altro processo è stato aperto in lettura, la chiamata ritorna un messaggio di errore

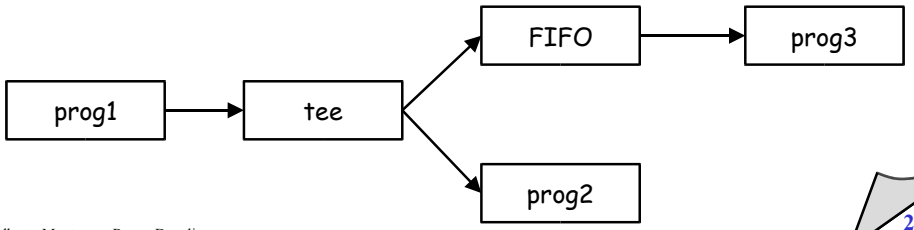
Named pipe - FIFO

- **Chiamata `write`**
  - se nessun processo ha aperto il file in lettura
    - viene generato un segnale **`SIGPIPE`**
      - ignorato/catturato: write ritorna **`-1`** e **`errno=EPIPE`**
      - azione di default: terminazione
- **Atomicità**
  - Quando si scrive su un pipe, la costante **`PIPE_BUF`** specifica la dimensione del buffer del pipe
  - Chiamate **`write`** di dimensione inferiore a **`PIPE_BUF`** vengono eseguite in modo atomico
  - Chiamate **`write`** di dimensione superiore a **`PIPE_BUF`** possono essere eseguite in modo non atomico
    - La presenza di scrittori multipli può causare interleaving tra chiamate **`write`** distinte

Named pipe

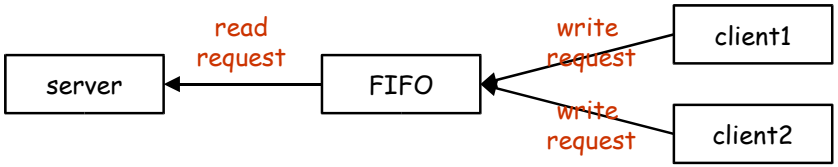
- **Utilizzazioni dei FIFO**
  - Utilizzati dai comandi shell per passare dati da una shell pipeline ad un'altra, senza passare creare file intermedi
- **Esempio:**

```
mkfifo fifo1
prog3 < fifo1 &
prog1 | tee fifo1 | prog2
```



Named pipe

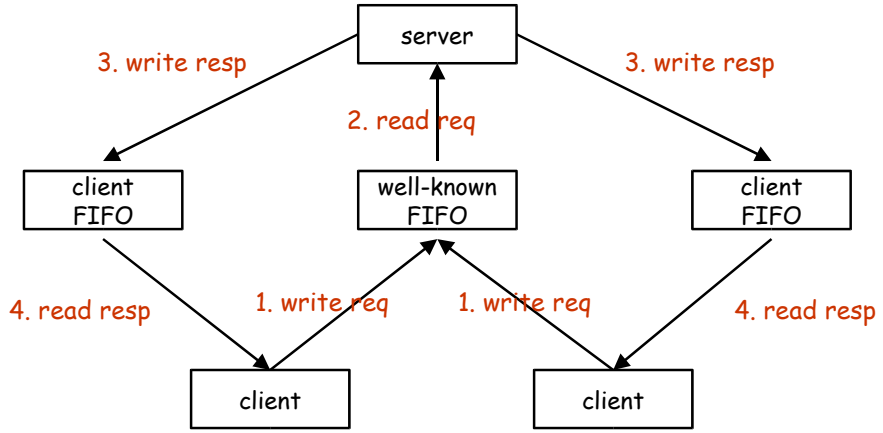
- **Utilizzazioni dei FIFO**
  - Utilizzati nelle applicazioni client-server per comunicare
- **Esempio:**
  - Comunicazioni client → server
    - il server crea un FIFO
    - il pathname di questo FIFO deve essere "well-known" (ovvero, noto a tutti i client)
    - i client scrivono le proprie richieste sul FIFO
    - il server leggere le richieste dal FIFO



Named pipe

- **Problema: come rispondere ai client?**
  - Non è possibile utilizzare il "well-known" FIFO
    - I client non saprebbero quando leggere le proprie risposte
  - Soluzione:
    - i client spediscono il proprio process id al server
    - i client creano un FIFO per la risposta, il cui nome contiene il process ID, e lo aprono in lettura
    - il server aprono in scrittura il client FIFO
    - il server scrive sul canale FIFO
  - Problemi:
    - Il server deve catturare SIGPIPE
      - il client può andarsene prima di leggere la risposta
    - Il server deve aprire in lettura il proprio FIFO
      - altrimenti, quando l'ultimo client termina, il server leggerà EOF

Named pipe



Esercizio

- **Descrizione**
  - riscrivere l'esercizio produttore-consumatore utilizzando una named pipe
  - utilizzare la pipe come buffer
  - il produttore scrive interi sulla pipe
  - il consumatore li stampa
  - utilizzare più produttori