Syscall UNIX

Gestione processi/esecuzione

Come viene identificato un processo?
Viene identificato da un numero intero chiamato *pid* (process identifier) che è di tipo pid_t per garantire portabilità.
`getpid()` restituisce il pid del processo corrente.
A cosa serve la syscall getppid?
`getppid()` restituisce il pid del parent del processo corrente.
Come viene creato un processo in UNIX
Il processo inizia con una chiamata di sistema `fork()` che crea una copia identica del processo chiamante. Entrambi condividono lo stesso codice, dati e spazio degli indirizzi.
Successivamente, il processo figlio può eseguire una chiamata di sistema `exec()`, che sovrascrive l'immagine del processo con un nuovo programma. In questo modo il processo figlio può essere completamente diverso dal processo padre.
Il processo padre può aspettare che il processo figlio termini o continuare a eseguire altre attività indipendentemente dal figlio.
Quale effetto ha la syscall fork?
La `fork` crea un nuovo processo duplicando il processo chiamante, l'unica cosa che cambia è il valore di ritorno: al figlio ritorna 0, al padre ritorna il pid del figlio. Se fallisce ritorna -1 al padre e non crea il figlio. Nota che non viene creato un nuovo processo da zero.
Come termina un processo UNIX?
Può terminare in diversi modi:
- chiamando `exit()`, che termina il processo e libera le risorse;
- chiamando `return` nel main(), simile a exit();
- ricevendo un segnale di terminazione, come `SIGTERM` o `SIGKILL`;
- a causa di un errore fatale, ad esempio segmentation fault.
Quale è lo scopo della syscall wait (o waitpid)?
La syscall `wait` è usata per aspettare un cambiamento di stato nel figlio del processo chiamante (terminazione, fermato da un segnale, ripristinato da un segnale...), ottenere informazioni sul figlio e rilasciare le risorse alla sua terminazione. Passando come argomento -1 aspetta la terminazione di tutti i figli.
Cosa succede alla terminazione di un processo orfano?
Se un processo rimane *orfano* (il padre è terminato prima di lui) viene associato al processo init (pid 1), il quale fa da raccoglitore di tutti i processi orfani. Quando un processo orfano termina, init si occupa di fare la wait in modo da liberare le risorse correttamente (gestisce eventuali processi zombie).
Cosa sono i processi zombie e perché esistono?
Se un figlio termina e il padre non ha fatto la syscall wait, allora il figlio diventa un *processo zombie*. Il kernel mantiene un set di informazioni minimali sul processo zombie per consentire al processo padre di fare una wait.
Come si lancia l'esecuzione di programmi? La syscall execve
1. Viene clonato un nuovo processo grazie alla syscall fork().
2. Il processo figlio prepara gli argomenti per il nuovo programma (nome del programma da eseguire,

eventuali argomenti).
3. Il processo figlio imposta un ambiente per il nuovo programma (variabili d'ambiente, percorso di ricerca per eseguibili, altre informazioni necessarie per l'esecuzione del programma).
4. Il processo figlio esegue la chiamata di sistema `execve()`, passando il percorso del nuovo programma e gli argomenti. La syscall execve() sostituisce l'immagine del processo figlio con quella del nuovo programma.
5. Una volta completata la chiamata ad execve(), il controllo viene passato al nuovo programma. Il processo figlio ora eseguirà il codice del programma specificato.

Le funzioni sorelle nella libreria C: exec*

- `execl()` e `execv()`: queste funzioni accettano una lista di argomenti o un vettore di argomenti per specificare il programma da eseguire e i suoi eventuali argomenti.
- `execle()` e `execve()`: simili a execl() e execv(), ma consentono di specificare anche l'ambiente del nuovo programma attraverso un array di stringhe.
- `execlp()` e `execvp()`: queste funzioni cercano l'eseguibile specificato nei percorsi definiti nella variabile d'ambiente PATH, semplificando la specifica del percorso completo dell'eseguibile.

Gestione file system

Cosa è la current working directory? chdir/getcwd

La *current working directory* è la directory attualmente attiva all'interno della quale un utente o un processo sta lavorando.

- `chdir` corrisponde al comando shell `cd` ed è una syscall che permette di cambiare la current working directory di un processo.
- `getcwd` è una syscall che ritorna il pathname assoluto della current working directory del processo chiamante.

Creazione e cancellazione di directory mkdir/rmdir

- `mkdir` è una syscall che crea una directory al path specificato (primo parametro) con i relativi permessi (secondo parametro).
- `rmdir` è una syscall che rimuove una directory vuota. Se la directory non è vuota la chiamata fallisce. Per rimuovere una directory non vuota, bisogna prima eliminare ricorsivamente il contenuto.

Quali sono e come funzionano le syscall per creare link fisici e simbolici?

`link` è la syscall per creare link fisici.
`symlink` è la syscall per creare link simbolici.
Entrambe prendono come argomenti il path del target esistente ed il path del link da creare.

readlink: la system call per "vedere" il target di un link simbolico

La syscall `readlink` permette di vedere dove punta un link simbolico, salvando il risultato in un buffer.

La "cancellazione" di un file in UNIX: qual è la particolarità della syscall unlink?

La syscall `unlink` rimuove il nome di un file. Elimina il file solamente se non esistono altri nomi che si riferiscono ad esso e se tale file non è in uso da altri processi al momento dell'esecuzione del comando.

Perché c'è anche la syscall rename? Non è equivalente a link(vecchio, nuovo)+unlink(vecchio)?

La syscall `rename` permette di cambiare il nome di file o directory e può essere usata per spostarli. Nonostante sia equivalente a fare link(vecchio, nuovo) e poi unlink(vecchio), presenta dei vantaggi:
- atomicità;
- gestisce eventuali problemi di accesso ai file system.

La system call per leggere le informazioni dei file: stat
La syscall `stat` viene utilizzata per ottenere informazioni su un file. Ritorna una struct che contiene numero dell'inode, dimensione, gruppo e utente proprietario, numero di link, tempi di accesso e altro.
Creare file speciali: mknod
La syscall `mknod` permette di creare file speciali come file di dispositivo, file FIFO (named pipe), ecc.
Cambiare i permessi di un file: chmod
La syscall `chmod` consente di cambiare i permessi di un file o di una directory.
Cambiare il proprietario/gruppo di un file: chown
La syscall `chown` permette di cambiare il proprietario e/o il gruppo di un file o di una directory.
access: controllo a priori se una operazione su un file è permessa. Perché questa system call è deprecata?
La syscall `access` consente di verificare i permessi di un processo verso un file o una directory.
Questa syscall è deprecata perchè introduce una race condition. Infatti il controllo dei permessi e la seguente operazione su file o directory non sono atomici e quindi potrebbe succedere che tra il controllo e l'operazione, i permessi vengano cambiati.
Quale è lo scopo della system call umask?
La syscall `umask` serve per impostare la maschera di creazione dei file di un processo, la quale determina i permessi che vengono rimossi di default quando un nuovo file o directory viene creato.
Aggiungere/togliere sottoalberi al file system: mount/umount2
UNIX ha un'unica gerarchia di file. Se ho più dischi e/o unità di massa (es. chiavetta usb) devo "innestare" nell'albero principale del filesystem il sottoalbero che rappresenta l'unità di massa. Questa operazione può essere fatta solo da root. Inoltre, questa operazione va fatta in una directory vuota.
`mount` collega un filesystem specificato a un punto di montaggio nel filesystem principale.
`umount2` scollega un filesystem montato dal filesystem principale.
Cambiare l'ampiezza di un file: truncate. Cosa succede se su un file si eseguono in successione open(..., O_CREAT), truncate, close?
La syscall `truncate` modifica la dimensione di un file a una dimensione specificata. Con length = 0 si cancella il contenuto del file.
La sequenza di operazioni è utile per creare e configurare rapidamente la dimensione di un file, assicurando che il file esista e abbia la dimensione desiderata.
La system call scomoda e per ottenere la lista dei file di una directory: getdents
La syscall `getdents` restituisce l'elenco delle directory entries di una specifica directory. Non è da utilizzare in quanto è pesantissima.
Cambiare i tempi dei file: utime
La syscall `utime` consente di modificare i timestamp di accesso e modifica di un file. Se i nuovi timestamp sono NULL, imposta l'ora corrente.
Le varianti 'l-' delle syscall: lstat, lchown, lchmod
Le varianti *l-* delle syscall operano sui link simbolici, piuttosto che sui file a cui questi link puntano.
- `lstat` : restituisce informazioni sul link simbolico stesso.
- `lchown` : modifica il proprietario/gruppo del link simbolico, non del file target.
- `lchmod` : modifica i permessi del link simbolico.

Le varianti 'f-' delle syscall: fstat, fchown, fchmod, ftruncate, fchdir
Le varianti *f-* delle syscall per identificare il file utilizzano come argomento un file descriptor (fd) invece che il path.

Gestione file

La syscall open, come funziona e perché può avere 2 o 3 parametri?
La syscall `open` apre un file e restituisce un file descriptor che rappresenta il file aperto. I due parametri obbligatori sono il path del file e la modalità di apertura. Il terzo parametro (facoltativo) è usato insieme al flag `O_CREAT` per specificare i permessi del file creato.
Duplicazione di file system: dup/dup2/dup3
Queste syscall sono utilizzate per duplicare file descriptor (fd), permettendo di ottenere un nuovo fd che si riferisce allo stesso file o risorsa del fd originale.
- `dup` : duplica un fd, restituendo il fd duplicato.
- `dup2` : duplica un fd in uno specificato.
- `dup3` : simile a dup2, ma permette di specificare flag aggiuntivi.
Cosa succede ai file aperti quando un processo esegue una fork?
Quando viene eseguita una `fork` il processo figlio ottiene copie esatte dei file descriptor aperti dal processo chiamante. Gli offset e i flag dei fd sono condivisi tra padre e figlio, quindi ogni modifica ai file (anche avanzamento della lettura) viene vista da entrambi i processi.
Cosa succede ai file aperti quando un processo esegue una exec?
Durante una chiamata a `exec` i file descriptor aperti nel processo originale rimangono aperti nel nuovo programma, a meno che non siano stati contrassegnati con il flag `FD_CLOEXEC` (close-on-exec).
Scopo della syscall close
La syscall `close` serve a chiudere un file descriptor terminato l'utilizzo, liberando così le risorse associate.
Qual è l'uso delle syscall read e write?
La `read` legge dati da un file tramite il suo fd, inserendo i dati letti in un buffer. La `write` scrive dati su un file tramite il suo fd, prendendoli da un buffer. Queste syscall sono fondamentali per le operazioni di I/O a basso livello.
Come vengono gestiti i buffer multipli di readv e writev?
Le syscall `readv` e `writev` permettono di leggere e scrivere dati usando buffer multipli, migliorando l'efficienza delle operazioni di I/O. I buffer multipli vengono gestiti passando come argomento un array di struct `iovec` , descriventi i singoli buffer (puntatore e lunghezza).
Come funzionano pread e pwrite e che senso hanno nei programmi multithread?
`pread` e `pwrite` sono syscall di I/O posizionale, utilizzate per leggere e scrivere dati da file senza modificare l'offset del file. In un contesto multithread permettono a diversi processi di accedere contemporaneamente a diverse parti di un file senza interferire l'uno con l'altro.
Qual è il funzionamento di lseek? Come vengono gestite le lseek oltre la fine del file?
La syscall `lseek` permette di spostare l'offset di un file. Se l'offset supera la fine del file:
- in caso di lettura non viene restituito nulla (EOF),
- in caso di scrittura il file viene esteso con byte a zero fino a raggiungere il nuovo offset.
Quali flag possono essere gestiti con fcntl?
La syscall `fcntl` può gestire vari flag per controllare il comportamento dei file descriptor.
- *Flag dei fd*: `F_GETFD` ottiene i flag del fd, `F_SETFD` imposta i flag del fd, `FD_CLOEXEC` chiude il fd durante una exec.
- *Flag dei file*: `F_GETFL` ottiene i flag del file, `F_SETFL` imposta i flag del file, `O_APPEND` aggiunge i dati alla fine del file, `O_NONBLOCK` operazioni di I/O non bloccanti, `O_SYNC` scrive i dati in modo sincrono.
- *Altri flag*: `F_DUPFD` duplica il fd con un valore minimo specificato, `F_DUPFD_CLOEXEC` duplica il

fd con il flag FD_CLOEXEC impostato, `F_GETOWN` ottiene l'ID del processo proprietario del fd (per segnali asincroni), `F_SETOWN` imposta l'ID del processo proprietario del fd, `F_GETLK` ottiene le informazioni di blocco del file, `F_SETLK` imposta un blocco del file (non bloccante), `F_SETLKW` imposta un blocco del file (bloccante).

Funzioni specifiche dei file speciali (device): uso della syscall ioctl.

La syscall `ioctl` è utilizzata per interfacciarsi con i device. È una chiamata multifunzione che permette di eseguire operazioni di controllo e configurazione, ad esempio:
- ottenere informazioni del dispositivo,
- configurare parametri del dispositivo,
- controllare operazioni specifiche.

L'opzione O_PATH di open

Quando si apre un file con l'opzione `O_PATH`, il fd risultante non può essere utilizzato per operazioni di lettura o scrittura, può essere invece usato per operazioni che non richiedono l'accesso ai dati, come fstat, fchdir...

Le varianti con suffisso -at: openat, mkdirat, fchownat, fchmodat, fstatat, unlinkat, symlinkat...

Le varianti di syscall con il suffisso *-at* accettano un fd di una directory come parametro, permettendo operazioni sui file tramite percorsi relativi rispetto a tale directory. Queste funzioni prendono un fd di una directory come primo parametro, seguito da un percorso relativo e altri parametri specifici.

Gestione utenti

UNIX prevede 3 tipi di utente e di gruppo: reale, effettivo e salvato: perché?

Il concetto di utente/gruppo reale, effettivo e salvato è fondamentale per la gestione dei permessi e la sicurezza. Questo modello consente ai programmi di operare in modo sicuro e controllato, utilizzando i privilegi minimi necessari in ogni fase dell'esecuzione.

Utente Reale (Real User ID - UID): rappresenta l'identità dell'utente che ha avviato il processo.

Utente Effettivo (Effective User ID - EUID): determina i permessi con cui il processo viene eseguito.

Utente Salvato (Saved Set-User-ID - SUID): memorizza l'EUID originale di un processo prima che venga cambiato.

Valgono le stesse suddivisioni per i gruppi.

getuid, setuid, getgid, setgid, geteuid, seteuid, getreiud, setreuid, getresuid, setresuid, getresgid, setresgid, getgroups, setgroups

Sono tutte syscall utilizzate per ottenere e impostare gli ID di utente e gruppo per i processi.
- `getuid` / `getgid`: ottenere UID/GID reale del processo chiamante.
- `setuid` / `setgid`: impostare UID/GID reale, effettivo e salvato del processo all'UID/GID specificato.
- `geteuid` / `seteuid`: ottenere/impostare l'UID effettivo del processo chiamante.
- `getreuid` / `setreuid`: ottenere/impostare l'UID reale ed effettivo del processo chiamante.
- `getresuid` / `setresuid`: ottenere/impostare l'UID reale, effettivo e salvato del processo chiamante.
- `getresgid` / `setresgid`: ottenere/impostare il GID reale, effettivo e salvato del processo chiamante.
- `getgroups` / `setgroups`: ottenere/impostare la lista dei gruppi supplementari del processo chiamante.

Segnali

I segnali sono meccanismi di comunicazione tra processi usati per notificare eventi, ad esempio:
- `SIGINT` : interruzione,
- `SIGTERM` : terminazione gentile,
- `SIGKILL` : terminazione forzata (non può essere ignorato).

Le syscall kill e signal hanno nomi fuorvianti: cosa fanno?
La syscall `kill` invia un segnale a un processo o a un gruppo di processi. La syscall `signal` permette di impostare un gestore per un segnale.

sigaction è più flessibile di signal. Cosa permette in più?
La syscall `sigaction` imposta un gestore per un segnale, con più opzioni di configurazione rispetto a `signal` :
- bloccare specifici segnali durante l'esecuzione del gestore,
- gestire i segnali con informazioni aggiuntive,
- uso di flags e opzioni.

Le funzioni di gestione dei segnali.
Altre funzioni per gestire i segnali in un processo:
- `raise` invia un segnale al processo chiamante.
- `pause` sospende l'esecuzione del processo fino alla ricezione di un segnale.
- `sigwait` attende la ricezione di uno dei segnali specificati in un insieme.

Cosa sono le funzioni signal-safe? Perché esistono funzioni unsafe?
I segnali possono interrompere l'esecuzione del codice in qualsiasi momento. Le funzioni *signal-safe* sono quelle che possono essere utilizzate in un gestore di segnali senza causare comportamenti imprevedibili o non sicuri. Le funzioni *unsafe* possono causare race condition, stati incoerenti o deadlock. Queste esistono perché alcune operazioni non possono garantire la sicurezza e la consistenza quando vengono interrotte.

La definizione di insiemi di segnali sigsetops
Un dato di tipo `sigset_t` rappresenta un insieme di segnali. Su questi insiemi sono definite delle operazioni (sigsetops), implementate attraverso le seguenti funzioni:
- `sigemptyset` inizializza un insieme di segnali come vuoto.
- `sigfillset` inizializza un insieme di segnali includendo tutti i segnali.
- `sigaddset` aggiunge un segnale a un insieme di segnali.
- `sigdelset` rimuove un segnale da un insieme di segnali.
- `sigismember` controlla se un segnale è presente nell'insieme di segnali.

Mascheramento e attesa di segnali (sigsuspend, sigprocmask, sigpending)
Mascheramento e attesa sono tecniche per controllare come i segnali vengono gestiti da un processo.
- `sigprocmask` modifica la maschera dei segnali del processo, ovvero l'insieme dei segnali che il processo blocca.
- `sigsuspend` sospende l'esecuzione del processo fino alla ricezione di un segnale, modificando temporaneamente la maschera dei segnali.
- `sigpending` interroga i segnali bloccati e in sospeso per il processo.

Comunicazione fra processi (locali)

pipe
Una *pipe* è un meccanismo di comunicazione che consente a un processo di inviare dati a un altro processo in modo sequenziale e unidirezionale. Si genera usando la funzione `pipe` , che crea una coppia di fd, uno per la lettura ([0]) e uno per la scrittura ([1]). Di default, le operazioni di lettura e

scrittura su una pipe sono bloccanti. Le pipe anonime sono usate tra processi con una relazione padre-figlio.

named pipe

Le *named pipe (o FIFO)* possono essere usate da processi non correlati. Sono rappresentate come file nel filesystem, permettendo ai processi di utilizzarle tramite il path del file. Una named pipe si crea utilizzando la funzione `mkfifo`. Dopo la creazione i processi possono gestirla come un normale file con open, read, write e close. La named pipe va chiusa e riaperta se si raggiunge la fine del file.

UNIX sockets

Le *UNIX sockets* sono un meccanismo di comunicazione tra processi all'interno dello stesso sistema operativo. A differenza delle pipe, possono gestire comunicazioni bidirezionali e sono più flessibili, supportando sia connessioni datagramma (`SOCK_DGRAM`, tipo UDP) che stream (`SOCK_STREAM`, tipo TCP).

Attesa di eventi

Ieri: select, pselect

Le funzioni `select` e `pselect` permettono di monitorare più fd per vedere se sono pronti per operazioni di lettura, scrittura o se ci sono eccezioni, senza bloccare il programma.

- `select` permette di monitorare i fd fino a quando uno o più diventano pronti per un'operazione.
- `pselect` permette inoltre di gestire il mascheramento dei segnali, il che può prevenire race condition.

Oggi: poll, ppoll

Le funzioni `poll` e `ppoll` hanno lo stesso utilizzo di *select* e *pselect*, ma in maniera più flessibile e scalabile grazie all'uso di strutture `pollfd`.

Domani: epoll (solo linux)

`epoll` è un'API di Linux per il monitoraggio di fd, progettata per essere più efficiente rispetto a *select* e *poll*, soprattutto quando si gestiscono numerosi fd.

Networking: Berkeley sockets

socket, bind, listen, accept, connect

- `socket` crea un endpoint di comunicazione e restituisce un fd.
- `bind` assegna un indirizzo al socket (server).
- `listen` mette il socket in ascolto per connessioni in entrata (server).
- `accept` accetta una connessione in entrata (server).
- `connect` stabilisce una connessione con un server (client).

send, recv, sendto, recvfrom, sendmsg, recvmsg, sendmmsg, recvmmsg, shutdown

- `send` invia dati su un socket.
- `recv` riceve dati da un socket.
- `sendto` invia un messaggio a un socket specificato (usato con UDP).
- `recvfrom` riceve un messaggio da un socket e può recuperare l'indirizzo del mittente.
- `sendmsg` invia un messaggio a un socket, con la possibilità di includere più buffer di dati.
- `recvmsg` riceve un messaggio da un socket, con la possibilità di leggere più buffer di dati.
- `sendmmsg` e `recvmmsg` sono estensioni di *sendmsg* e *recvmsg* per inviare e ricevere più messaggi in un'unica chiamata di sistema.
- `shutdown` disabilita le operazioni di invio e/o ricezione su un socket.

getsockname, getpeername, socketpair, getsockopt, setsockopt
- `getsockname` recupera l'indirizzo locale associato a un socket.
- `getpeername` recupera l'indirizzo del peer associato a un socket.
- `socketpair` crea una coppia di socket connessi tra loro.
- `getsockopt` recupera le opzioni di un socket.
- `setsockopt` imposta le opzioni di un socket.

Altre syscall

Gestione del tempo: time, gettimeofday, settimeofday, nanosleep, setitimer, getitimer
Ci sono diverse syscall per la gestione del tempo:
- `time` restituisce il tempo corrente in secondi dall'epoca Unix (1° gennaio 1970).
- `gettimeofday` recupera il tempo corrente in secondi e microsecondi dall'epoca Unix.
- `settimeofday` imposta il tempo corrente di sistema in secondi e microsecondi.
- `nanosleep` sospende l'esecuzione del chiamante per il tempo specificato in nanosecondi.
- `setitimer` imposta un timer che invia segnali periodici.
- `getitimer` recupera il valore corrente di un timer.

Memory mapping: mmap, mremap, munmap
Le funzioni di memory mapping:
- `mmap` mappa un file o un dispositivo in memoria.
- `mremap` ridimensiona o sposta una regione di memoria mappata.
- `munmap` annulla la mappatura di un file o dispositivo dalla memoria.

Usi speciali di file descriptor: eventfd, signalfd, pidfd, timerfd...
I fd possono essere utilizzati per varie operazioni, oltre alla gestione dei file. Alcuni usi speciali includono:
- `eventfd` : notificare eventi tra i processi.
- `signalfd` : ricevere segnali come dati leggibili.
- `pidfd` : monitorare il cambiamento dello stato di un processo specifico.
- `timerfd` : gestire timer e ricevere notifiche quando un timer scade.

Guardie sul file system: inotify
`inotify` è un'interfaccia che consente di monitorare in tempo reale i cambiamenti su file e directory. Funzioni principali:
- `inotify_init` crea un fd per inotify.
- `inotify_add_watch` aggiunge una sorveglianza su un file o una directory.
- `inotify_rm_watch` rimuove una sorveglianza.