Ripetizione Esame Salvi

Sommario

[Processi e thread 1](#_Toc139999204)

[Funzioni rientranti e non rientranti 2](#_Toc139999205)

[Come si inviano e ricevono i segnali 2](#_Toc139999206)

[System call open e close 2](#_Toc139999207)

[System call read e write 3](#_Toc139999208)

[System call lseek 3](#_Toc139999209)

[Process Control Block (PCB) 4](#_Toc139999210)

[Immagine di un processo 4](#_Toc139999211)

[Stati di un processo 5](#_Toc139999212)

[PID 5](#_Toc139999213)

[Setuid e setgid 5](#_Toc139999214)

[Sticky bit 6](#_Toc139999215)

[Processi zombie 6](#_Toc139999216)

[Hard e soft link (link simbolici) 6](#_Toc139999217)

[Dup2 7](#_Toc139999218)

[Differenza tra signal e sigaction 7](#_Toc139999219)

[Tipi di file 8](#_Toc139999220)

[TCP e UDP 8](#_Toc139999221)

[Pipe 8](#_Toc139999222)

[FIFO 11](#_Toc139999223)

[Mutex 12](#_Toc139999224)

[Variabili di condizione 12](#_Toc139999225)

[Semafori 13](#_Toc139999226)

[Famiglia exec 14](#_Toc139999227)

[Socket 15](#_Toc139999228)

[Segnali 15](#_Toc139999229)

# Processi e thread

Un **processo** è un programma in esecuzione che utilizza un insieme di risorse, le quali sono allocate dal kernel. I processi hanno un proprio spazio di indirizzamento e **sono isolati l’uno dall’altro**; infatti **per permettere la comunicazione** tra loro vengono utilizzati alcuni meccanismi (**socket**, **pipe** ecc.). Un **thread** è una sequenza di istruzioni all’interno di un processo, che può essere un po’ vista come una “sotto-unità” di un processo. I thread **condividono lo stesso spazio di indirizzamento del processo di cui fanno parte**, compreso il codice e i dati. I thread, come i processi, sono gestiti dal SO, ma richiedono meno risorse rispetto ai processi. Inoltre, ogni thread ha il proprio stack ed il suo stato di esecuzione. I thread possono condividere le stesse risorse, ma devono essere sincronizzati correttamente per evitare problemi di concorrenza (mutex, semafori, var. cond.). Vengono utilizzati per eseguire attività parallele, in modo da migliorare l’efficienza.

# Funzioni rientranti e non rientranti

Una funzione è detta **rientrante** se può essere eseguita simultaneamente, in modo sicuro, da **più thread di esecuzione** nello stesso processo. Con modo sicuro si intende che la funzione ottiene i risultati previsti indipendentemente dallo stato di esecuzione di qualsiasi altro thread. Una funzione rientrante può essere interrotta in qualsiasi momento e riavviata in un momento successivo senza la perdita di dati. Le funzioni rientranti usano variabili locali o assicurano protezione dei dati globali facendone una copia locale. Le funzioni **non rientranti**, invece, sono funzioni che **non possono essere condivise** da più task, **a meno che** non venga assicurata la mutua esclusione mediante l’utilizzo di strutture di **sincronizzazione** (come semafori).

# Come si inviano e ricevono i segnali

I processi possono inviare segnali in modi diversi a seconda del contesto specifico, ma generalmente coinvolgono l’utilizzo di mezzi di comunicazione o protocolli di trasmissione. **All’interno di un SO**, i processo comunicano attraverso **pipe**, **socket** **locali** e **segnali** (**kill**(), **raise**(), **signal**(), **sigaction**()), mentre nei contesti di **comunicazioni di rete**, i processi possono inviare segnali attraverso **protocolli di trasmissione** come **TCP** o **UDP**.

# System call open e close

#include <fcntl.h>

int open (const char \*path, int oflag, /\*mode\_t mode\*/…);

La system call **open** permette di aprire o di creare file:

* **path** è il nome del file da creare o aprire;
* **oflag** può assumere diversi valori, come ad esempio **O\_RDONLY**, che significa che il file viene aperto in lettura, **O\_WRONLY**, che significa che il file viene aperto in scrittura, **O\_RDWR**, che invece significa aprire il file sia in lettura che scrittura. Esistono poi altri oflag opzionali, come **O\_TRUNC**, che tronca il file a lunghezza zero, se esiste e se è aperto in una modalità che comprende la scrittura, o anche **O\_CREAT**, che crea il file se non esiste;
* il **terzo argomento**, invece, viene utilizzato solo quando si crea un file e permette di **specificare i permessi di accesso allo stesso file**, come S\_IRUSR, S\_IWUSR, S\_IRWXU ecc.

#include <unistd.h>

int close (int fildes);

La system call **close** permette di chiudere un file, a partire dal descrittore di file che viene passato come parametro. Chiude il file ritornando -1 in caso di errore, ma in generale quando un processo termina, tutti i file aperti vengono automaticamente chiusi.

# System call read e write

#include <unistd.h>

ssize\_t read(int fildes, void \*buf, size\_t nbytes);

ssize\_t write (int fildes, const void \*buf, size\_t nbytes);

La system call **read** permette di leggere da file: in particolare, **legge dal file indicato dal descrittore** di file passato come primo parametro, **nbytes byte**, ossia il terzo parametro, **in buf**, passato come secondo parametro. Dopodichè la **posizione corrente viene aggiornata**, e la function ritorna il numero di byte effettivamente letti, 0 se si è arrivati alla fine del file, -1 in caso di errore. La read **legge un numero di byte minore** di quanto richiesto **nel caso in cui si legga** **da** **terminale**, poichè di solito viene letta una riga per volta, quando si è **interrotti da un** **segnale**, ed era stata letta una quantità parziale di dati, o anche in altri casi. La system call **write**, invece, **scrive nel file indicato dal descrittore** di file passato come primo parametro, **nbytes byte a partire dalla posizione corrente di buf**. Viene **aggiornata la posizione corrente** e vengono restituiti il numero di byte effettivamente scritti, oppure -1 in caso di errore.

# System call lseek

Ad ogni file aperto è associato un valore intero non negativo, detto **offset corrente del file**, che misura il **numero di byte dall’inizio del file**.

#include <unistd.h>

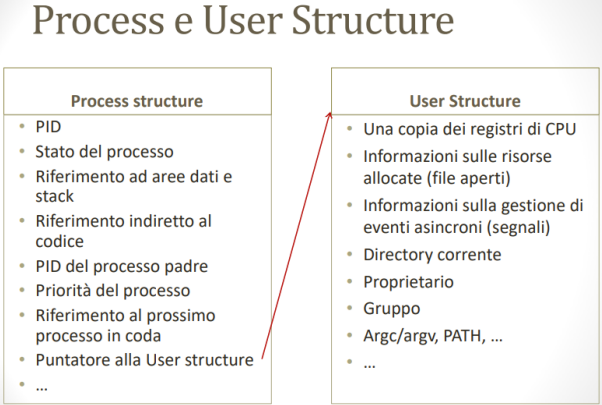
off\_t lseek (int fildes, off\_t offset, int whence);

La system call **lseek** permette di riposizionare l’offset corrente del file. In particolare, lseek ritorna il nuovo offset, oppure -1 in caso di errore. Quando il file viene aperto l’offset viene inizializzato a 0, a meno che non si specifichi l’opzione O\_APPEND. Il terzo argomento della chiamata a lseek, ossia **whence**, può assumere come valori **SEEK\_SET**, ossia l’offset viene posto a offset byte dall’inizio del file, **SEEK\_CUR**, dove viene aggiunto offset all’offset corrente, o **SEEK\_END**, ossia l’offset viene posto alla fine del file, più offset.

# Process Control Block (PCB)

Il PCB è rappresentato da due strutture dati:

* **process structure**, che contiene informazioni necessarie al sistema per la gestione del processo, a prescindere dal suo stato;
* **user structure**, che contiene informazioni necessarie solo se il processo è residente in memoria centrale.



# 

# Immagine di un processo

L’immagine di un processo è l’insieme delle aree di memoria e delle strutture dati associate al processo. Non tutta l’immagine è accessibile in modalità utente, in quanto si hanno della **parti** **anche nel** **kernel**. Ogni processo può essere soggetto a swapping, in quanto non tutta l’immagine può essere trasferita in memoria. Una parte dell’immagine è **swappabile**, mentre un’altra parte è **residente**, o **non swappabile**. In Unix l’immagine di un processo **si divide in**:

* **process structure** (kernel, non swappabile), che è una struttura dati facente parte del PCB;
* **text structure** (kernel, non swappabile), che è un elemento della text table (struttura dati globale in cui sono contenuti i puntatori ai segmenti di testo eventualmente condivisi dai processi) associato al codice del processo;
* **area dati globali utente** (user, swappabile), come suggerisce il nome è relativa all’area dei dati globali. E’ suddivisa in due segmenti, uno relativo ai dati inizializzati e l’altro relativo a dati non inizializzati.
* **stack e heap utente** (user, swappabile), che sono aree dinamiche associate al programma eseguito;
* **stack del kernel** (kernel, swappabile), che è uno stack di sistema associato al processo per le system call;
* **user structure** (kernel, swappabile), che è una struttura dati facente parte del PCB.

# Stati di un processo

* **Init**, caricamento in memoria del processo e inizializzazione delle strutture del SO;
* **Ready**, processo pronto;
* **Running**, processo in esecuzione, ossia utilizza la CPU;
* **Sleeping** (**blocked**), processo sospeso in attesa di un evento;
* **Terminated**, processo deallocato dopo la terminazione;
* **Zombie**, processo terminato ma in attesa che il padre ne rilevi lo stato di terminazione;
* **Swapped**, processo, o parte di esso, temporaneamente trasferito in memoria secondaria (su disco).

# PID

#include <unistd.h>

pid\_t getpid (void)

pid\_t getppid (void)

Il PID, process ID, è un **numero univoco assegnato ad ogni processo**, in modo da **identificarlo**. Si può ottenere il PID di un processo mediante chiamata **getpid**(), la quale appunto restituisce il PID del processo invocante. Si può anche ottenere il PID del processo padre, richiamando la function **getppid**(), che appunto restituisce il processo padre del processo invocante. Queste function hanno **sempre successo**.

# Setuid e setgid

**Ogni f**ile ha un **proprietario ed un gruppo** che lo possiede: tali informazioni si trovano rispettivamente in **st\_uid** e **st\_gid** della **struttura stat**, che memorizza **informazioni varie sui file** che vengono inseriti come parametri di input della **system call stat**. A ciascun processo vengono associati gli identificativi di **real user ID** e **real group ID,** che **identificano l’utente**, ma anche gli identificativi **effective user ID** e **effective group ID**, che **determinano i permessi di accesso ai file**. Di solito l’effective user ID coincide con il real user ID e l’effective group ID coincide con il real group ID. Visto che un programma viene eseguito con i permessi di chi lo manda in esecuzione, e non di chi lo possiede, si può utilizzare un flag (**S\_ISUID**) in **st\_mode**, ossia il **campo della struttura stat che memorizza il tipo di file**, in modo che quando un determinato file di programma viene eseguito, l’effective user ID del processo sia quello di chi possiede il file; tale flag è detto **set-user-ID**. Analogamente è possibile mandare in esecuzione un processo con un effective group ID uguale al group ID del file eseguibile; il relativo flag (**S\_ISGID**) è detto **set-group-ID**.

# Sticky bit

Lo **sticky b**it (**S\_ISVTX**) permette di richiedere al kernel che l’**immagine del segmento di testo di un processo resti allocata nell’area di swap anche dopo la sua terminazione**. Quando lo sticky bit viene abilitato su una directory, i file nella directory possono essere rinominati o cancellati solo se l’utente ha i permessi di scrittura sulla directory e se è proprietario del file, o proprietario della directory o superutente.

# Processi zombie e adottati

Un processo che termina, **non scompare** dal sistema **fino a che il padre non accetta il suo codice di terminazione**; tale processo si trova nello stato zombie. Se il padre non termina e non esegue mai una wait(), il codice di terminazione non sarà mai accettato ed il processo resterà sempre uno zombie. Un processo zombie non ha aree codice e dati allocati, quindi non usa molte risorse di sistema, ma continua ad avere un PCB. Se invece il padre termina prima del figlio, quest’ultimo diventa un processo orfano e viene adottato dal processo init (primo processo avviato dal kernel all’accensione del computer) che imposta il ppid del processo a 1, visto che init ha pid = 1.

# Hard e soft link (link simbolici)

I **link** sono scorciatoie per accedere a file o directory che vengono utilizzate frequentemente e permettono di avere più di un punto di accesso per lo stesso file o directory. Esistono **due tipi di link**:

* **soft link**, dove viene creato un file che contiene solo il nome del riferimento;
* **hard link**, dove non viene creato un file e si fa riferimento direttamente ai dati fisici.

Un link permette di accedere ad un file utilizzando un nuovo path name. Se utilizzo un **hard link** i due file fanno riferimento allo stesso **i-node**, ossia una **struttura dati che contiene informazioni relative ad un file in una directory** (tipo di file, dimensione del file, numero dell’i-node, UID e GID, permessi di accesso, ecc.). Se invece utilizzo un **soft link,** anche chiamato **link simbolico**, questo corrisponde ad un file speciale che contiene il path name del file collegato. La **tecnica degli hard link** permette di **risparmiare memoria** e consente **accessi veloci** **ai file collegati**, ma **può generare situazioni anomale**, mentre la **tecnica dei soft link** ha come **svantaggio** lo **spreco di memoria** e la **maggior lentezza nell’accesso**.

# Dup2

#include <unistd.h>

int dup2 (int filedes, int filedes2)

La system call **dup2**() permette di **duplicare i descrittori di file**: in particolare prende in input due descrittori di file, copiando quello passato come primo argomento (filedes) in quello passato come secondo (filedes2**). Nel caso in cui filedes2 sia attualmente attivo**, **lo chiude** **e lo fa riferire allo stesso file a cui fa riferimento filedes**. Il descrittore di file originale e quello copiato condividono lo stesso puntatore interno al file e le stesse modalità di accesso. Dup2() ritorna il nuovo descrittore se ha successo, altrimenti -1. Questa system call viene spesso **utilizzata nelle pipe per redirezionare l’input o l’output** di un processo **verso la** stessa (**pipe**), in modo da consentire la **comunicazione tra i processi**.

# Differenza tra signal e sigaction

#include <signal.h>

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

int sigaction (int signo, const struct sigaction \*act, struct sigaction \*oact);

La funzione **signal**() fornisce lo strumento per istruire il kernel ad eseguire una determinata azione quando il processo chiamante riceve un determinato segnale. Prende in input due argomenti:

* **signo**, che è un intero che rappresenta il nome del segnale occorso;
* **func**, che è un puntatore ad una funzione e può avere come valore **SIG\_IGN**, che indica al sistema di ignorare il segnale, **SIG\_DFL**, che indica al sistema di associare l’azione di default, o l’indirizzo di una funzione da chiamare quando occorre il segnale (**gestore del segnale**).

La function **sigaction**(), invece, sostituisce la funzione signal() nelle versioni più recenti di Unix. Prende in input il numero del segnale **signo**, il puntatore **act**, che se è non nullo significa che stiamo modificando l’azione del segnale, ed infine un altro puntatore **oact**, che se non nullo, fa tornare al sistema l’ultima azione effettuata per quel segnale nello stesso puntatore oact. In generale **signal**() è una funzione **più** **semplice** **e meno flessibile** **rispetto a sigaction**(), poichè quest’ultima ha come argomento due puntatori ad una **struct**, la quale permette di specificare vari comportamenti e opzioni aggiuntive per la gestione dei segnali. Tramite il campo **sa\_mask** di questa struct, ad esempio, si può **specificare l’insieme dei segnali aggiunti alla maschera dei segnali.**

# Tipi di file

Esistono **diversi tipi di file**:

* **regolari**, che sono il tipo di file più comune, contenente dati in una qualche forma (ad es. di testo o binari);
* **directory**, la quale contiene nomi di altri file e puntatori alle informazioni su tali file;
* **file speciali a blocco**, utilizzati per rappresentare dispositivi che consistono in un insieme di blocchi ad indirizzamento casuale (dischi);
* **file speciali a caratteri**, usati per rappresentare dispositivi che costituiscono flussi di caratteri (terminali, stampanti e interfacce di rete);
* **FIFO**, utilizzati per la comunicazione tra processi;
* **Socket**, che è un tipo di file utilizzato per la comunicazione su rete tra processi;
* **Link simbolici**, ossia i soft link.

# TCP e UDP

**TCP** (**Transmission Control Protocol**) e **UDP** (**User Datagram Protocol**) sono **protocolli di comunicazione utilizzati nelle socket** per trasferire dati tra processi che si trovano su host differenti. Le **socket sono analoghe a dei “varchi”**, infatti **un processo riceve/invia messaggi tramite la socket**. Il **TCP** è un protocollo **orientato alla connessione**, ossia, prima di inviare dati, il TCP **stabilisce una connessione tra il processo mittente e il processo destinatario**, garantendo un **trasporto affidabile tra i due**. Il TCP, inoltre, implementa il **controllo della congestione**, ossia **regola la velocità di trasmissione dei dati per evitare il sovraccaricamento della rete**, ma anche il **controllo di flusso**, ossia il **processo mittente evita di sovraccaricare il processo destinatario** con una quantità eccessiva di dati. Per quanto riguarda il **protocollo UDP**, invece**, non è richiesta l’instaurazione di una connessione** prima di trasmettere dati, di conseguenza l’UDP è chiaramente **meno affidabile del TCP**. L’UDP, inoltre, **non offre neanche funzionalità come il controllo di flusso ed il controllo della congestione**. In generale l’**UDP** è un protocollo **più rapido ma meno affidabile**, in quanto non garantisce la consegna dei dati e non ha funzionalità aggiuntive come invece ha il TCP.

# Pipe

#include <unistd.h>

int pipe (int fd[]);

Il termine pipe è utilizzato per indicare situazioni in cui si **connette un flusso di dati da un processo ad un altro**. Le **pipe anonime** sono un **canale di comunicazione**, mantenuto a livello kernel, che appunto **unisce due processi**; è **unidirezionale** e **permette la comunicazione solo tra processi con un antenato comune**. Una pipe presenta **due lati di accesso**, **ciascuno associato ad un descrittore** di file: il **lato di lettura** è acceduto invocando **read**(), mentre il **lato di scrittura** è acceduto invocando **write**(). In caso di scrittura, memorizza il suo input in un buffer, di dimensione massima **PIPE\_BUF**. Quando un processo ha finito di utilizzare un lato di una pipe, chiude il descrittore con **close**(). La **system call** **pipe**() crea una pipe anonima e restituisce due descrittori di file: lato di lettura fd[0], che significa aperto in lettura, e lato di scrittura fd[1], che significa aperto in scrittura. Se un **processo legge da una pipe**:

* **read**() restituisce 0, che indica la fine dell’input, dopo che sono stati letti tutti i dati;
* se la pipe è vuota e il lato di scrittura è ancora aperto, il processo si sospende fino a che diventa disponibile qualche input;
* se il processo tenta di leggere più byte di quelli presenti, read() restituisce il numero di byte effettivamente letti.

Se, invece, un **processo scrive su una pipe**:

* se il lato di lettura è chiuso, **write**() fallisce e allo scrittore è inviato un segnale SIGPIPE, la cui azione di default fa terminare il ricevente;
* se scrive meno byte di quelli che una pipe può contenere (PIPE\_BUF), write() viene eseguita in modo atomico (cioè l’intero buffer dati viene scritto nella pipe in un’unica operazione, senza essere interrotto);
* se scrive più byte di quelli che una pipe può contenere, invece, non c’è garanzia di atomicità.

Insieme al comando pipe() vengono spesso utilizzate la **system call fork**(), in modo da creare un **canale IPC dal genitore al figlio o viceversa**, la **system call dup2**, per **duplicare i descrittori di file**, e la **system call exec**, per **eseguire un programma**. Un esempio classico di utilizzo delle pipe è:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/types.h>

#define read 0

#define write 1

int main() {

int dscr[2];

pipe(dscr);

pid\_t pr1;

pr1=fork();

if(pr1){

close(dscr[read] );

dup2(dscr[write], 1);

close(dscr[write]);

execlp("ls ","ls ","-R",NULL);

}

else if(!pr1){

close(dscr[write] );

dup2(dscr[read], 0);

close(dscr[read]);

execlp("grep","grep",argv[1],NULL);

}

exit(0);

}

int dscr[2] = descrittori di file della pipe;

pipe(dscr) = restituisce due descrittori di file, uno per la lettura (dscr[read]) e uno per la scrittura (dscr[write]);

if(pr1) = se è processo padre;

if(!pr1) = se è processo figlio;

close(dscr[read]) = chiudiamo il descrittore di lettura;

dup2(dscr[write],1) = duplichiamo il descrittore di scrittura “dscr[write]” per il descrittore relativo allo STDOUT;

close(dscr[write]) = chiudiamo il descrittore di scrittura;

execlp(“ls”,”ls”,”-R”,NULL) = eseguo il comando “ls -R”;

dup2(dscr[read],0) = duplichiamo il descrittore di lettura “dscr[read]” per il descrittore relativo allo STDIN;

# FIFO

#include <sys/stat.h>

int mkfifo (const char \*filename, mode\_t mode);

Le FIFO sono un **meccanismo di comunicazione tra processi** (**IPC**) che consente lo **scambio di dati tra processi non in relazione tra** **loro**. Le FIFO sono implementate come **file speciali** nei sistemi operativi e, in pratica, sono un **tipo particolare di pipe**, chiamato **pipe con nome**. Si possono creare FIFO sia da riga di comando che da programma mediante **mkfifo**, che prende in **input il nome della FIFO ed i permessi ad essa assegnati**. Sulle FIFO si possono utilizzare gli **stessi comandi dei file**, come **rm** per eliminarle o **cat** per leggere ad esempio. Si può aprire una FIFO mediante system call **open**(), con l’unica differenza che un programma **non può aprirla in modalità O\_RDWR** poichè le **FIFO** vengono **utilizzate** **per** **passare** **i** **dati in una singola direzione**. Un’opzione particolare, rispetto ai file regolari, riguarda l’oflag che può avere come, appunto, opzione **O\_NONBLOCK** (non bloccante). L’aggiunta di questa opzione a O\_RDONLY permette alla chiamata ad open di ritornare con successo ed immediatamente, anche se la FIFO non è stata aperta in scrittura da alcun processo. Nel caso in cui si specificasse come oflag solo O\_RDONLY, invece, la chiamata ad open() si blocca e non ritorna fin quando un processo non aprirà la stessa FIFO in scrittura. Per quanto riguarda l’apertura in modalità scrittura, se viene specificato solo O\_WRONLY allora la chiamata ad open si blocca e attende un processo che apre la stessa FIFO in lettura, mentre se viene aggiunto O\_NONBLOCK allora ritornerà immediatamente, ma se nessun processo ha la FIFO aperta in lettura, allora open() ritornerà un valore di errore, ossia -1. Se la FIFO è aperta in **modalità bloccante** (quindi senza O\_NONBLOCK), una chiamata a **read**() attenderà fino a quando sarà disponibile qualche dato da leggere, nel caso in cui la FIFO è aperta in scrittura, e restituirà 0, se la FIFO non è aperta in scrittura. Per quanto riguarda invece una chiamata a **write**(), questa aspetterà fino a che i dati possono essere scritti, nel caso in cui la FIFO è aperta in lettura, mentre genererà un segnale **SIGPIPE** (segnale che viene inviato quando un processo tenta di scrivere su una pipe non aperta in modalità lettura), se la FIFO non è aperta in lettura. Nel caso invece di **FIFO non bloccanti**, una **read**() su una FIFO vuota restituisce un errore (**EAGAIN**, che è un errore che indica che un’**operazione non può essere completata al momento e deve essere ritentata** **successivamente**) se la FIFO è aperta in scrittura, e 0 se non lo è. Una **write**() su una FIFO non bloccante, invece, genera un segnale SIGPIPE se la FIFO non è aperta in lettura, mentre se lo è e il numero di byte da scrivere è <= **PIPE\_BUF** (costante che rappresenta la **dimensione massima dei dati che possono essere scritti o letti in modo atomico da un lato della pipe all’altro**) si controlla se c’è lo spazio per il numero di byte specificato: in caso affermativo vengono trasferiti tutti i byte, mentre in caso negativo la write() ritorna immediatamente con un errore EAGAIN. Se invece la FIFO è aperta in lettura ma il numero di byte è > PIPE\_BUF, si controlla se c’è spazio per almeno 1 byte nella FIFO e si trasferiscono tanti byte quanto è lo spazio libero. In caso di FIFO piena, invece, la write() ritorna immediatamente con l’errore EAGAIN.

# Mutex

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

Quando molteplici thread di un processo condividono la stessa memoria, è necessario che ciascun thread mantenga la coerenza dei propri dati. In particolare, quando un thread modifica una variabile che è leggibile e/o modificabile anche da altri thread, si possono generare delle inconsistenze di dati. Per **risolvere il problema dell’accesso a dati condivisi di thread diversi**, esistono i **mutex**. Un mutex è un oggetto che **possiede due stati**: **bloccato** e **non bloccato**. Nel primo caso, quando un thread blocca il mutex, gli altri thread che tentano di bloccarlo restano in attesa. Quando il thread bloccante rilascia il mutex, uno dei thread in attesa lo acquisisce. Un mutex è una **variabile di tipo pthread\_mutex\_t** che può essere **inizializzata in modo statico**, impostandola al valore della costante **PTHREAD\_MUTEX\_INITIALIZER**, oppure in **modo dinamico**, invocando **pthread\_mutex\_init**(). **Se allochiamo un mutex dinamicamente**, è necessario invocare **pthread\_mutex\_destroy**() per deallocare il mutex. Si hanno poi **pthread\_mutex\_lock**() e **pthread\_mutex\_unlock**() per bloccare e sbloccare le variabili di tipo mutex.

# Variabili di condizione

#include <pthread.h>

int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*attr);

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

Le variabili di condizione costituiscono un **meccanismo di sincronizzazione per i thread**. Le variabili di condizioni hanno **tre componenti**:

* il **predicato**, ossia la **condizione che un thread controllerà per determinare se deve attendere**;
* il **mutex**, che è il **meccanismo che protegge il predicato**;
* la **variabile di condizione**, che è il **meccanismo con cui il thread attende il verificarsi di una condizione**.

Come per i mutex, una variabile di condizione, che è di tipo **pthread\_cond\_t**, può essere **inizializzata staticamente**, mediante la costante **PTHREAD\_COND\_INITIALIZER**, o **dinamicamente**, tramite **pthread\_cond\_init**(). **In caso di allocazione dinamica**, deve essere utilizzato **pthread\_cond\_destroy**() per deallocare una variabile di condizione.

#include <pthread.h>

int pthread\_cond\_wait(pthread\_cond\_t \*cptr, pthread\_mutex\_t \*mptr);

int pthread\_cond\_signal(pthread\_cond\_t \*cptr);

int pthread\_cond\_broadcast(pthread\_cond\_t \*cptr);

Un thread può attendere su una variabile di condizione, invocando **pthread\_cond\_wait**(). Quando la condizione si verifica, si può **risvegliare un thread** in attesa, mediante **pthread\_cond\_signal**(), oppure si possono **risvegliare tutti i thread** in attesa, tramite **pthread\_cond\_broadcast()**. Nella chiamata a pthread\_cond\_wait, bisogna passare come argomento anche il mutex associato alla variabile di condizione, il quale verrà temporaneamente rilasciato quando un processo si blocca sulla cond\_wait, in modo da permettere ad altri thread di accedere a determinati dati condivisi.

# Semafori

I semafori sono un **meccanismo di sincronizzazione tra processi o thread di un processo**. In POSIX esistono **due tipi di semafori**: quelli **basati su nome** e quelli **basati su memoria**.

#include <semaphore.h>

sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode, int value);

int sem\_close(sem\_t \*sem);

int sem\_unlink(const char \*name);

Per quanto riguarda i **semafori** **basati su nome**, la funzione **sem\_open**() **crea un nuovo semaforo**, **oppure** ne **apre** **uno già esistente**. La function prende in input il nome del semaforo, eventuali permessi se è specificato O\_CREAT nell’oflag ed il valore iniziale del semaforo. Per **chiudere** **un semaforo basato su nome**, si utilizza la funzione **sem\_close**(), mentre **per rimuoverlo** si utilizza la funzione **sem\_unlink**(), in quanto con sem\_close() viene solo chiuso ma non rimosso.

#include <semaphore.h>

int sem\_init(sem\_t \*sem, int shared, unsigned int value);

int sem\_destroy(sem\_t \*sem);

Per quanto riguarda i **semafori** **basati su memoria**, invece, vengono utilizzati **sem\_init**() e **sem\_destroy**(), rispettivamente **per creare e deallocare** un semaforo.

#include <semaphore.h>

int sem\_wait(sem\_t \*sem);

int sem\_post(sem\_t \*sem);

Su tutti i tipi di semafori, comunque, possono essere chiamate le funzioni **sem\_wait**() e **sem\_post**(), per la gestione dei meccanismi di sincronizzazione tra processi.

# Famiglia exec

int execl (const char \*path, const char \*arg0, …/\* (char \*)0 \*/);

int execle (const char \*path, const char \*arg0, … /\* (char \*)0, char \*const envp[] \*/);

int execlp (const char \*file, const char \*arg0, …/\* (char \*)0 \*/);

int execv (const char \*path, char \*const argv[]);

int execve (const char \*path, char \*const argv[], char \*const envp[]);

int execvp (const char \*file, char \*const argv[]);

Le funzioni della famiglia exec **permettono di caricare un programma dal disco**, **sostituendolo all’immagine del processo** (programma che un processo sta eseguendo) **corrente**. Quando il nuovo programma termina, anche il processo termina e non si può tornare all’immagine precedente. **Exec è una** **famiglia di primitive**, anche se **in realtà** **fanno tutte rifeirmento a** **execve**. Execve esegue il file o lo script indicato da path, passandogli la lista di argomenti indicata da argv e come ambiente la lista di stringhe indicata da envp. Le differenze tra le funzioni della famiglia exec, sono riassunte dai suffissi “**v**” e “**l**”, che stanno per **vector** e **list**. Nel **primo caso** gli **argomenti** sono **passati tramite il vettore di puntatori argv[]** **a stringhe terminate con zero**, mentre nel **secondo caso** le **stringhe degli argomenti** vengono **passate alla funzione come lista di puntatori**. Una **seconda differenza** riguarda la **modalità con cui si specifica il programma** che si vuole eseguire: con il **suffisso** “**p**” si indicano le **due funzioni che replicano il comportamento della shell**, quindi se l’argomento file non contiene “/” viene considerato come un nome di programma e viene eseguita una ricerca tra i file presenti nella lista delle directory specificate dalla variabile di ambiente PATH, mentre le **altre quattro funzioni si limitano a cercare di eseguire** **il** **file indicato dall’argomento** **path**, che viene **interpretato** **come pathname del programma**. Infine, l’**ultima differenza** riguarda la **modalità con cui viene passata la lista delle variabili di ambiente**: con il **suffisso** “**e**” **vengono indicate le funzioni che necessitano di un vettore di parametri envp**[], **analogo** **a** **quello** utilizzato per gli argomenti **a riga di comando**, **mentre le altre utilizzano il valore della variabile environ** **del processo di partenza** per costruire l’ambiente.

# Socket

**Due o più processi sullo stesso host utilizzano una qualche forma di IPC per comunicare**, mentre **due o più processi su differenti host**, connessi ad una rete di comunicazione comune, **utilizzano messaggi per comunicare**. Nei SO moderni, i servizi disponibili in rete si basano principalmente sul **modello client/server**, che **consente ai sistemi di condividere risorse** e cooperare per il raggiungimento di un obiettivo **mediante l’utilizzo di socket**. Le **socket sono analoghe a dei “varchi”,** in quanto **permettono ad un processo di inviare e ricevere messaggi**. **Un processo che vuole inviare un messaggio lo fa uscire dalla socket**, presupponendo che esista un qualche tipo di infrastruttura esterna che trasporterà il messaggio attraverso la rete fino al varco del processo di destinazione. **Per trasportare i messaggi**, vengono utilizzati dei **protocolli appositi**, che prendono il nome di **TCP e UDP** (il continuo sta nel paragrafo apposito. Tornando al modello client-server, **un server crea una socket mediante la system call socket**(), la quale **non può essere condivisa con altri processi**. Alle socket viene **assegnato un nome mediante system call bind**(): alle **socket locali si associa un filename nel file system**, mentre per le **socket di rete il nome corrisponde ad un identificatore di servizio**. A questo punto, il processo **server attende che un client si connetta alla socket mediante system call listen**(), **creando una coda di connessioni di arrivo che il server può accettare utilizzando la system call accept**(). **Quando il server invoca accept**(), **viene creata una nuova socke**t, utilizzata solamente **per comunicare con il client specifico**. Dal **lato client**, invece, un processo **crea una socket** invocando sempre la system call apposita (socket()) e successivamente **invoca connect**() **per stabilire una connessione con il server**.

# Segnali

Un segnale è la **notifica**, ad un processo, **che si è verificato un evento** e assomigliano alle interruzioni hardware, in quanto interrompono il normale flusso di esecuzione. **Un processo**, con i dovuti permessi, **può inviare un segnale ad un altro processo**, **o a se stesso**, diventando così una forma primitiva di IPC. **In conseguenza dell’arrivo di un segnale**, un processo può chiedere al kernel di:

* **ignorare il segnale**, (non fattibile per SIGKILL e SIGSTOP) cioè ignorare i segnali generati dalle eccezioni, con il comportamento del processo che in questo caso sarebbe indefinito;
* **intercettare il segnale**, ossia fornire una funzione da eseguire per un determinato segnale;
* **eseguire le azioni di default**, infatti tutti i segnali hanno una propria azione di default, come può essere la terminazione ad esempio.

**Per istruire il kernel ad effettuare una di queste tre azioni**, si utilizza la **funzione signal**() (esiste il paragrafo apposito, con anche le differenze rispetto a sigaction()).

#include <sys/types.h>

#include <signal.h>

int kill (pid\_t pid, int signo);

int raise (int signo);

Le system call kill() e raise() permettono di inviare segnali: in particolare **kill**() **permette di inviare il segnale signo al processo pid**, mentre **raise**() **permette di inviare un segnale a sé stesso**.

#include int sigprocmask (int how, const sigset\_t \*set, sigset\_t \*oset);

Uno dei **problemi** più insidiosi quando si parla di segnali è l’**occorrenza di un secondo segnale mentre si sta gestendo ancora il primo**. In questo caso bisogna prendere delle **precauzioni per evitare race condition**. Per fare ciò, si può **utilizzare una maschera dei segnali mediante chiamata di sistema sigprocmask**(), la quale **consente di specificare un insieme di segnali da bloccare** e **restituisce la lista dei segnali precedentemente bloccati**. Quest’ultima cosa viene fatta **in modo da ripristinare successivamente lo stato della maschera precedente**, nel caso in cui serva farlo. Come primo parametro della chiamata a sigprocmask() abbiamo **how** che consiste nell’**azione che si vuole effettuare**: **SIG\_BLOCK** se si vogliono **aggiungere segnali**, **SIG\_UNBLOCK** se si vogliono **rimuovere segnali**, o **SIG\_SETMASK** se si vuole **sostituire completamente la maschera corrente con una nuova maschera**. Il secondo argomento, ossia **set**, **deve contenere l’insieme dei segnali da aggiungere o rimuovere in base al parametro how**, mentre il **terzo parametro oset conterrà la maschera precedente**, nel caso in cui non sia NULL. Esistono poi delle funzioni apposite per svuotare o riempire la maschera, ma anche aggiungere o rimuovere un segnale specifico.

#include <signal.h>

int sigemptyset(sigset\_t \*set);

int sigfillset(sigset\_t \*set);

int sigaddset(sigset\_t \*set, int signo);

int sigdelset(sigset\_t \*set, int signo).