int sem_wait(size_t *sem):

- Decrementa il semaforo puntato da `sem`. Se il valore del semaforo è maggiore di zero, il decremento procede e la funzione ritorna immediatamente.
  Se il semaforo ha attualmente il valore zero, la chiamata si blocca finché non non diventa possibile eseguire il decremento (ovvero, il valore del semaforo sale sopra lo zero) oppure un gestore del segnale interrompe la chiamata.
- Valore di ritorno: in caso di successo ritorna 0, altrimenti viene ritornato -1 e settato `errno`

int sem_post(size_t *sem):

- Incrementa il semaforo puntato da `sem`. Se di conseguenza il valore del semaforo diventa maggiore di zero, allora un altro processo o thread bloccato in una chiamata a `sem_wait(..)` verrà riattivato e procederà a bloccare il semaforo.
- Valore di ritorno: in caso di successo ritorna 0, altrimenti viene ritornato -1 e settato `errno`

int sem_destroy(size_t *sem):

- Distrugge il semaforo unnamed all'indirizzo puntato da `sem`.
  Solo un semaforo inizializzato da `sem_init(..)` deve essere distrutto utilizzando `sem_destroy(..)`. La distruzione di semafori su cui altri processi o thread sono attualmente bloccati (a causa di una chiamata a `sem_wait(..)`) produce un comportamento indefinito.
- Valore di ritorno: in caso di successo ritorna 0, altrimenti viene ritornato -1 e settato `errno`

~ Semafori Named ~

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value):

sem_t *sem_open(const char *name, int oflag):

- Crea un nuovo semaforo named (primo prototipo) o ne apre uno già esistente (secondo prototipo). Tale semaforo viene identificato da `name`.
  L'argomento `ofleg` specificano i flags che controllano l'operazione:
  - `O_CREAT`: Crea il semaforo se non esiste ancora. Il proprietario (User ID) del semaforo è impostato all'ID effettivo del processo chiamante. Il gruppo (Group ID) è impostato al Group ID effettivo del processo.

    - `O_EXCL`: Se usata insieme a `O_CREAT`, questa chiamata fallisce se il semaforo name esiste già. Questo garantisce che il chiamante sia il creatore effettivo del semaforo (creazione atomica).
      Se `O_CREAT` è specificato in oflag, allora devono essere forniti altri due argomenti:
    - mode: Specifica i permessi da assegnare al nuovo semaforo (come in `open(..)`). Questi permessi sono modificati dalla `umask` del processo.
    - value: Specifica il valore iniziale del nuovo semaforo. Se si crea un semaforo per la mutua esclusione, value dovrebbe essere 1. Se si crea un semaforo per la sincronizzazione (attesa), value dovrebbe essere 0.
      Se `O_CREAT` non è specificato, `sem_open()` controlla solo i primi due argomenti e ignora mode e value, tentando di accedere a un semaforo esistente.
- Valore di ritorno: in caso di successo restituisce l'indirizzo del nuovo semaforo (per permettere altre operazioni su tale semaforo), mentre in caso di errore restituisce `SEM_FAILED` e setta `errno`

int sem_close(sem_t *sem):

- Chiude il semaforo named riferito da `sem`, consentendo di liberare tutte le risorse che il sistema ha assegnato al processo chiamante per questo semaforo
- Valore di ritorno: in caso di successo ritorna 0, altrimenti viene ritornato -1 e settato `errno`

int sem_unlink(const char *name):

- Rimuove il semaforo named riferito da `name`. Il nome del semaforo viene immediatamente rimosso mentre viene definitivamente distrutto una volta che tutti i processi che hanno tale semaforo aperto lo chiudono
- Valore di ritorno: in caso di successo ritorna 0, altrimenti viene ritornato -1 e settato `errno`

~ Mutex ~

int pthread_mutex_init(pthread_mutex_t *restrict mutex, const *pthread_mutexattr_t attr):

- Inizializza il mutex riferito da `mutex` con gli attributi specificati da `attr`.
  Se `attr == NULL` allora vengono utilizzati gli attributi di default per il mutex.
  È possibile inizializzare staticamente un mutex usando la macro `PTHREAD_MUTEX_INITIALIZER`: `pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;`
  Questo equivale a chiamare `pthread_mutex_init(..)` con `attr` uguale a `NULL`, ma non esegue alcun controllo degli errori a runtime.
  Tentare di inizializzare un mutex che è già stato inizializzato comporta un

Comportamento Indefinito (Undefined Behavior).
(`mutex` : Un puntatore a un'area di memoria allocata che conterrà la struttura del mutex; `attr` : Un puntatore a un oggetto p `thread_mutexattr_t` che definisce le proprietà del mutex)

- Valore restituito: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

int pthread_mutex_destroy(pthread_mutex_t *mutex):

- Distrugge l'oggetto mutex puntato da `mutex` .(viene portato ad uno stato non inizializzato, infatti un mutex distrutto può essere re-inizializzato successivamente con `pthread_mutex_init()`)
  E' sicuro distruggere un mutex solamente quando è SBLOCCATO e nessun thread è bloccato in attesa di esso.
  La distruzione libera qualsiasi risorsa del sistema (se presente) che era stata allocata per il mutex. Non libera, tuttavia, la memoria occupata dalla variabile `pthread_mutex_t` stessa (se questa era stata allocata dinamicamente con malloc, spetta all'utente fare la free() dopo la destroy).
- Valore di ritorno: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

int pthread_mutex_lock(pthread_mutex_t *mutex):

- Blocca l'oggetto mutex riferito da `mutex` .
  Se il mutex è attualmente SBLOCCATO (unlocked), il thread chiamante ne diviene il proprietario (owner) e il mutex viene impostato nello stato BLOCCATO (locked). La funzione ritorna immediatamente.
  Se il mutex è già BLOCCATO da un altro thread, il thread chiamante viene sospeso (blocked) e messo in attesa. Il thread attenderà finché il mutex non diverrà disponibile. L'operazione di sblocco e acquisizione è atomica.
- Valore di ritorno: Valore restituito: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

int pthread_mutex_unlock(pthread_mutex_t *mutex):

- Rilascia l'oggetto mutex riferito da 'mutex'. Quando un mutex diventa disponibile, se ci sono thread in attesa , lo scheduler ne seleziona uno per risvegliarlo.
  Quando un mutex diventa disponibile, se ci sono thread in attesa (blocked in pthread_mutex_lock), lo scheduler ne seleziona uno per risvegliarlo.
- Valore di ritorno: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

~ Barriere ~

int pthread_barrier_init(pthread_barrier_t *restrict barrier, const pthread_barrierattr_t *restrict attr, unsigned int count):

- Alloca le risorse necessarie all'utilizzo della barriera riferita da `brrier` e la inizializza con gli attributi riferiti da `attr`. Se `attr` è NULL, vengono usati gli attributi di default per la barriera; l'effetto è lo stesso di passare un oggetto attributi inizializzato con i valori predefiniti.
  L'argomento `count` indica il numero di thread che devono chiamare `pthread_barrier_wait(..)` prima che questi vengano contemporaneamente sbloccati dalla barriera (valore specificato in `count` deve essere maggiore di zero).
- Valore di ritorno: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

int pthread_barrier_wait(pthread_barrier_t *barrier):

- Sincronizza i threads partecipanti alla barriera puntata dal puntatore `barrier`.
  Il thread che chiamante si blocca fino a che un numero di thread pari a `count`, specificato in `pthread_barrier_init(..)` non ha chiamato la `pthread_barrier_wait(..)`.
  Quando l'ultimo thread necessario raggiunge la barriera (cioè ha chiamato la `pthread_barrier_wait(..)`) tutti i thread in attesa (sleep) vengono rilasciati (svegliati) e la barriera viene reimpostata automaticamente al suo stato iniziale, pronta per essere utilizzata in un ciclo successivo senza dover chiamare nuovamente `pthread_barrier_init(..)`
- Valore di ritorno: in caso di successo restituisce `PTHREAD_BARRIER_SERIAL_THREAD` ad UN SOLO thread, arbitrario, tra quelli che erano bloccati dalla barriera e 0 a tutti gli altri thread. Mentre in caso di insuccesso viene restituito un intero (diverso da zero) e settato `errno`
- ATTENZIONE: Il valore `PTHREAD_BARRIER_SERIAL_THREAD` serve a eleggere un **"Capitano Temporaneo"** o un **"Responsabile delle Pulizie"** tra i thread, senza dover usare altri mutex o logiche complesse basate sugli ID dei thread.
  Immagina un algoritmo iterativo (che gira in un ciclo `while`) diviso in fasi, tipico del calcolo scientifico o del rendering grafico:

  - **Fase Parallela:** Tutti i thread calcolano un pezzo di dati.
  - **Sincronizzazione:** Bisogna aspettare che tutti abbiano finito.
  - **Fase Seriale (Il problema):** Prima di ricominciare il ciclo, bisogna fare un'operazione unica sui dati condivisi (es. resettare un contatore globale, scambiare i buffer di memoria, scrivere un checkpoint su disco, sommare i risultati parziali).

  Chi esegue la Fase Seriale?

  - **Se la fanno tutti:** Disastro. Avresti Race Conditions (tutti scrivono insieme) o operazioni duplicate (il file viene scritto N volte).

- **Se la fa solo il Thread 0:** Funziona, ma devi cablare nel codice il concetto di "Thread 0". E se il Thread 0 è lento o bloccato?
- **Se usi un Mutex:** Aggiungi overhead inutile.

La Soluzione: L'Eletto
Qui entra in gioco `PTHREAD_BARRIER_SERIAL_THREAD`. Quando la barriera "cade" (cioè l'ultimo thread è arrivato), la funzione `pthread_barrier_wait` ritorna:
`0` a tutti i thread tranne uno.
`PTHREAD_BARRIER_SERIAL_THREAD` a **un solo thread** (solitamente l'ultimo che è arrivato, ma lo standard dice "uno arbitrario"). Questo thread che riceve tale valore sarà quello che deve essere incaricato di eseguire l'operazione **Seriale** (cioè l'operazione reset,cleanup, o di aggregazione dei risultati parziali).
Questo ti permette di eseguire codice seriale in modo sicuro ed efficiente **all'interno** del flusso parallelo.

int pthread_barrier_destroy(pthread_barrier_t *barrier):

- Distrugge l'oggetto barriera puntato da `barrier`, rilasciando qualsiasi risorsa interna allocata in precedenza da `pthread_barrier_init(..)`. Condizione per una distruzione in sicurezza è distruggere una barriera solamente se non c'è alcun thread bloccato su di essa(nessun thread è fermo dentro la chiamata `pthread_barrier_wait(..)`)
- Valore di ritorno: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

~ Condition Variables ~

int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr):

- Inizializza la condition variable puntata da `cond` usando gli attributi specificati in `attr`. Se `attr` è NULL, vengono usati gli attributi di default per la condition variable.
  È possibile inizializzare staticamente una condition variable usando la macro `PTHREAD_COND_INITIALIZER`: `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
  Questo equivale all'inizializzazione dinamica con `attr` uguale a NULL, ma non esegue controlli di errore a runtime.
- Valore di ritorno: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex):

- Blocca l'esecuzione del thread chiamante fino a che la condition variable `cond` non viene segnalata (finché non avviene la condizione che può far risvegliare quel therad).

Questa funzione esegue tre operazioni fondamentali in una sequenza specifica:

- Rilascio Atomico del Mutex: La funzione rilascia il mutex riferito da `mutex` e contestualmente sospende il thread chiamante (lo mette in coda di attesa sulla condition variable). Questa operazione è ATOMICA: non esiste intervallo di tempo in cui il mutex è rilasciato ma il thread non è ancora in attesa (questo previene il problema del "Lost Wakeup").
- Attesa (Sleeping): Il thread rimane in stato BLOCCATO finché un altro thread non chiama `pthread_cond_signal(..)` o `pthread_cond_broadcast(..)` sulla stessa variabile di condizione. Durante questo tempo, il mutex è libero e può essere acquisito da altri thread.
- Riacquisizione del Mutex: Prima di ritornare al chiamante (cioè appena il thread viene svegliato), la funzione tenta automaticamente di RI-ACQUISIRE (Lock) il mutex.
  Di conseguenza, quando `pthread_cond_wait(..)` ritorna, il thread chiamante POSSIEDE nuovamente il mutex (è locked).

- Valore di ritorno: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

int pthread_cond_signal(pthread_cont_t *cond):

- Sblocca (risveglia) almeno un thread che è attualmente bloccato sulla condition variable puntata da `cond`.
  Se ci sono più thread in attesa su `cond`, la politica di scheduling determina quale thread specifico viene sbloccato. Se non ci sono thread in attesa su `cond`, la funzione non ha alcun effetto.
  Quando un thread viene sbloccato dalla signal, non riprende immediatamente l'esecuzione dal punto successivo alla wait. Invece, esso transita dallo stato di attesa sulla condizione allo stato di attesa sul MUTEX associato. Il thread deve competere per riacquisire il lock del mutex prima di poter ritornare dalla chiamata `pthread_cond_wait(..)`.
  La funzione `pthread_cond_signal(..)` può essere chiamata dal thread corrente sia che esso possieda il mutex associato alla condizione, sia che non lo possieda. Tuttavia, chiamarla mentre si possiede il mutex è pratica comune per garantire una predicibilità dello scheduling.
- Valore di ritorno: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

int pthread_cond_broadcast(pthread_cont_t *cond):

- Sblocca (risveglia) tutti i thread che sono attualmente bloccati sulla condition variable puntata da `cond`.
  Se non ci sono thread in attesa su `cond'` la funzione non ha alcun effetto.
  Poiché ogni thread che esce da `pthread_cond_wait(..)` deve riacquisire il MUTEX

associato prima di ritornare, i thread risvegliati dal broadcast NON riprendono l'esecuzione tutti contemporaneamente. Invece:
    - Tutti i thread passano dallo stato "Waiting on Condition" allo stato "Waiting on Mutex".
    - I thread competono per acquisire il mutex uno alla volta.
    - Il thread che ottiene il lock ritorna dalla wait, esegue il suo codice critico, e (si spera) rilascia il mutex, permettendo al prossimo thread risvegliato di procedere. L'ordine in cui i thread acquisiscono il mutex dipende dalla politica di scheduling (es. priorità, FIFO).
- Valore di ritorno: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

int pthread_cond_destroy(pthread_cont_t *cond):

- Distrugge la variabile di condizione puntata da `cond`, rilasciando qualsiasi risosrsa interna allocata precedentemente da `pthread_cont_init(..)`.
  E' sicuro distruggere una condition variable solo se NESSUN thread è attualmente bloccato su di essa (ovvero, nessun thread è fermo dentro una chiamata `pthread_cond_wait(..)`)
- Valore di ritorno: in caso di successo restituisce 0, altrimenti un altro valore intero e setta `errno`

~ Processi ~

pid_t fork(void):

- Crea un nuovo processo duplicando il processo chiamante. Il nuovo processo è detto child mentre il processo chiamante è detto parent (genitore).
  Il processo figlio e il processo genitore vengono eseguiti in spazi di memoria separati. Al momento della `fork()`, entrambi gli spazi di memoria hanno lo stesso contenuto.
  Scritture in memoria, mappature di file e sblocchi di file (unmapping) effettuati da uno dei processi non influenzano l'altro.
  Il processo figlio è un duplicato esatto del processo genitore, eccetto per i seguenti punti principali:
    - il child ha il proprio PID univoco e questo PID non corrisponde all'ID di alcun gruppo di processi esistente o sessione;
    - L'ID del processo genitore del child è lo stesso dell'ID del processo genitore.
    - Il bambino non eredita i blocchi di memoria del genitore
    - L'utilizzo delle risorse di processo e i contatori del tempo della CPU vengono azzerati nel figlio.
    - L'insieme dei segnali in sospeso del bambino è inizialmente vuoto
    - Il figlio non eredita le regolazioni del semaforo dal genitore

    - Il figlio non eredita i blocchi dei record associati al processo dal suo genitore
    - Il figlio non eredita i timer dal genitore
    - Il figlio non eredita le operazioni di I/O asincrone in sospeso dal suo genitore né eredita alcun contesto di I/O asincrono dal suo genitore
      Dopo una fork() riuscita, entrambi i processi (genitore e figlio) continuano l'esecuzione dall'istruzione successiva alla chiamata fork().
- Valore di ritorno: in caso di successo il PID del processo figlio viene restituito al processo padre ed al processo figlio viene ritornato zero. In caso di insuccesso viene ritornato -1 al processo padre, non viene creato alcun figlio e viene settato `errno`

int execl(const char *path, const char *arg0, ... /*, (char *) NULL */):

- Sostituisce l'immagine del processo corrente con una nuova immagine di processo.
  Il codice del programma corrente viene cancellato dalla memoria, viene caricato il codice del nuovo eseguibile specificato da `path`, lo Stack, lo Heap e i dati globali vengono reinizializzati per il nuovo programma.
  Il PID (Process ID) RIMANE LO STESSO. Non viene creato un nuovo processo, ma il processo esistente si "trasforma".
- Argomenti:
    - `args`: Il percorso COMPLETO del file binario da eseguire
    - `arg0`: Per convenzione, il primo argomento deve essere il nome del file associato al programma che viene eseguito
    - `...`: Una serie variabile di puntatori a stringhe (const char *) che costituiscono gli argomenti da passare al nuovo programma
    - `NULL`: La lista degli argomenti DEVE essere terminata tassativamente da un puntatore NULL castato a (char *)
- Valore di ritorno: la funzione ritorna solamente se si verifica un errore, in quel caso ritorna -1 e setta `errno`, altrimenti non ritorna (poiché il codice chiamante è stato sostituito, non esiste più alcuna istruzione "successiva" da eseguire nel vecchio programma).

pid_t wait(int *wstatus):

- Sospende l'esecuzione del processo chiamante fino a che UNO dei sui figli non termina, se un processo figlio è già terminato al momento della chiamata allora ritorna immediatamente.
  La `wait(..)` permette la Sincronizzazione, cioè permette al genitore di aspettare la fine del lavoro del figlio e permette anche la **Resource Reaping** (Raccolta), ovvero permette di recuperare lo stato di terminazione del figlio e liberare le risorse del kernel associate ad esso.
  Senza tale chiamata un figlio terminato rimane in stato Zombie.
- Argomenti:

    - `wstatus`: puntatore ad interno che se diverso da `NULL` memorizza informazioni sulla causa della terminazione del figlio, se invece è `NULL` allora tali informazioni vengono scartate
- Valore di ritorno: in caso di successo ritorna il PID del processo figlio terminato, altrimenti ritorna -1
  (Questa chiamata di sistema viene utilizzata per attendere il cambiamento di stato di un processo figlio del processo chiamante e ottenere informazioni sul processo child il cui stato è cambiato.
  Un cambiamento di stato è considerato come: il child terminato; il child è stato fermato da un segnale; oppure il child è stato ripreso da un segnale. Nel caso di un child terminato, l'esecuzione di una `wait(..)` consente al sistema di rilasciare le risorse associate al child; se non viene eseguita un'attesa, il child terminato rimane nello stato "zombie")

int pipe(int pipefd[2]):

int pipe(int pipefd[2], int flags)

- Crea una pipe, un canale dati unidirezionale che può essere utilizzato per la comunicazione tra processi (IPC). L'array `pipefd` è utilizzato per ritornare due file descriptors che riferiscono alle estremità della pipe:
    - `pipefd[0]`: si riferisce all'estremità di lettura della pipe
    - `pipefd[1]`: su riferisce all'estremità di scrittura della pipe
      I dati scritti all'estremità di scrittura della pipe vengono memorizzati nel buffer dal kernel finché non vengono letti dall'estremità di lettura della pipe.
      Se `flags == 0` allora le due dichiarazioni sono identiche, ma i valori di `flags` possono essere sottoposti a OR bit a bit per ottenere comportamenti diversi (per i valori vedere terminale).
- Valore di ritorno: in caso di successo viene ritornato 0, altrimenti viene ritornato -1, settato `errno` e l'array `pipefd` rimane invariato

int close(int fd):

- Chiude un descrittore di file, in modo che non faccia più riferimento ad alcun file e possa essere riutilizzato.
  Tutti i blocchi di record contenuti nel file a cui erano associati e di proprietà del processo vengono rimossi indipendentemente dal descrittore di file utilizzato per ottenere il blocco.
  Se `fd` è l'ultimo descrittore di file che fa riferimento alla descrizione del file aperto sottostante , le risorse associate al file descriptor aperto vengono liberate; se il descrittore di file è stato l'ultimo riferimento a un file che è stato rimosso utilizzando `unlink(..)`, il file viene eliminato
- Valore di ritorno: in caso di successo ritorna zero, altrimenti ritorna -1 e setta `errno`

int mkfifo(const char *pathname, mode_t):

- Crea un file speciale FIFO con nome al percorso `pathname`. `mode` specifica i permessi di FIFO che vengono applicati al file creato modificati dalla `umask` del processo nel modo consueto: i permessi del file creato sono (mode & ~`umask`).
  Un file speciale FIFO è simile a una pipe, detta **Named Pipe** , solo che viene creato in modo diverso. Invece di essere un canale di comunicazione anonimo, una named pipe viene inserita nel file system chiamando `mkfifo(..)`.
  Una volta creato un file speciale FIFO in questo modo, qualsiasi processo può aprirlo per la lettura o la scrittura, allo stesso modo di un file ordinario.
  Tuttavia, deve essere aperto su entrambe le estremità contemporaneamente prima di poter procedere a eseguire qualsiasi operazione di input o output su di esso.
  L'apertura di una FIFO per la lettura normalmente si blocca finché un altro processo non apre la stessa FIFO per la scrittura e viceversa.
- Valore di ritorno: in caso di successo ritorna 0, in caso di pipe già creata ritorna `EEXIST`, mentre nel caso di insuccesso ritorna -1 e setta `errno`

INIZIO DIGRESSIONE: **umask**

NOME
umask - imposta la maschera di creazione dei file (User File Creation Mask)

DESCRIZIONE
La umask (User Mask) è una variabile di ambiente del processo che funge da "filtro sottrattivo" per i permessi di sicurezza.
Ogni volta che un programma tenta di creare un nuovo file (tramite open, mkdir, mkfifo) e specifica dei permessi ideali (es. 0666 rw-rw-rw-), il sistema operativo applica la umask per rimuovere certi permessi prima di scrivere il file su disco.
La logica è: "Il programma chiede il permesso X, ma se la umask vieta X, il risultato sarà: Niente permesso".

FORMULA MATEMATICA
I permessi finali sono calcolati con l'operazione bitwise:

Permessi Finali = (mode & ~umask)

In termini logici semplici:
Permessi Richiesti DAL PROGRAMMA
MENO (sottratti da)
Permessi Vietati DALLA UMASK
UGUALE
Permessi Effettivi SU DISCO

ESEMPIO PRATICO (Il caso classico)
Immaginiamo di chiamare mkfifo con permessi 0666 (lettura/scrittura per tutti). Supponiamo che la umask corrente sia 0022 (valore standard su Linux).

1. Richiesta (mode): 0666 (rw- rw- rw-) [Binario: 110 110 110]

2. Umask (divieto): 0022 (--- -w- -w-) [Binario: 000 010 010]
   (La umask dice: "Vieta la scrittura al Gruppo e agli Altri")
3. Calcolo (~umask): Il NOT di 0022 diventa 111 101 101.
4. Operazione AND:
   110 110 110 (0666)
   111 101 101 (NOT 0022)

---

110 100 100 (0644)

Risultato Finale: 0644 (rw- r-- r--). Il proprietario può leggere/scrivere. Gli altri possono solo leggere. (User può leggere e scrivere, Group e Other possono solo leggere)

UTILIZZO DA TERMINALE
Il comando `umask` senza argomenti mostra la maschera corrente.
$ umask
0022

VALORI COMUNI
0000 -> Nessun filtro. I file nascono esattamente come richiesti (pericoloso).
0022 -> Vieta la scrittura a "Gruppo" e "Altri" (Standard).
0077 -> Vieta tutto a "Gruppo" e "Altri". I file sono privati (rw-------).

FINE DIGRESSIONE

int unlink(const char *pathname):

- Elimina un nome dal file system. Se quel nome era l'ultimo collegamento a un file e nessun processo ha il file aperto, il file viene eliminato e lo spazio che stava utilizzando viene reso disponibile per il riutilizzo.
  Se il nome era l'ultimo collegamento a un file ma tutti i processi hanno ancora il file aperto, il file rimarrà esistente finché non verrà chiuso l'ultimo descrittore di file che fa riferimento ad esso.
  Se il nome si riferisce a un collegamento simbolico, il collegamento viene rimosso.
  Se il nome si riferisce a un socket, FIFO o dispositivo, il nome viene rimosso, ma i processi che hanno l'oggetto aperto possono continuare a utilizzarlo.
- Valore di ritorno: In caso di successo viene restituito 0, altrimenti restituisce -1 e viene settato `errno`

int shm_open(const char *name, int oflag, mode_t mode):

- Crea e apre un nuovo oggetto di memoria condivisa POSIX oppure ne apre uno esistente. Un oggetto di memoria condivisa POSIX è in effetti un handle che può essere utilizzato da processi non correlati per mappare tramite `mmap(..)` la stessa regione di memoria condivisa.

Il funzionamento di `shm_open(..)` è analogo a quello di `open(..)`. `name` specifica l'oggetto di memoria condivisa da creare o aprire. (Per l'uso portatile, un oggetto di memoria condivisa dovrebbe essere identificato da un nome della forma `/somename`: cioè, una stringa terminata da null composta fino a NAME_MAX (cioè, 255) caratteri costituita da una barra iniziale, seguita da uno o più caratteri, nessuno dei quali sono barre.)
Dopo un `shm_unlink(..)` riuscito, i tentativi di `shm_open(..)` di un oggetto con lo stesso nome falliscono (a meno che non sia stato specificato `O_CREAT`, nel qual caso viene creato un nuovo oggetto distinto)

- Valore di ritorno: in caso di successo ritorna un file descriptor, mentre in caso di insuccesso ritorna -1 e setta `errno`

int shm_unlink(const char *name):

- Esegue l'operazione inversa rispetto a `shm_opne(..)`, rimuovendo un oggetto creato in precedenza da `shm_open(..)`.
  L'operazione di `shm_unlink(..)` è analoga a `unlink(..)`: rimuove il nome di un oggetto di memoria condivisa e, una volta che tutti i processi hanno demappato l'oggetto, dealloca e distrugge il contenuto della regione di memoria associata.
- Valore di ritorno: in caso di successo ritorna 0, mentre in caso di insuccesso ritorna -1 e setta `errno`

int ftruncate(int fd, off_t length):

- Permette che il file riferito da `fd` venga troncato ad una dimensione di `length` byte precisa.
  Se il file in precedenza era più grande di questa dimensione, i dati extra vengono persi.
  Se in precedenza il file era più corto, viene esteso e la parte estesa viene letta come byte nulli ('\0').
  L'offset del file non viene modificato.
  Se la dimensione è cambiata, i campi `st_ctime` e `st_mtime` (rispettivamente, ora dell'ultima modifica di stato e ora dell'ultima modifica) per il file vengono aggiornati e i bit di modalità `set-user-ID` e `set-group-ID` possono essere cancellati.
  Con `ftruncate(..)`, il file deve essere aperto per la scrittura.
- Valore di ritorno: in caso di successo ritorna 0, in caso di insuccesso ritorna -1 e setta `errno`

void *mmap(void addr[.length], sise_t length, int prot, int flags, int fd, off_t offeset):

- Crea una nuova mappatura nello spazio degli indirizzi virtuali del processo chiamante. L'indirizzo di partenza per la nuova mappatura è specificato in `addr`. L'argomento `length` specifica la lunghezza della mappatura (che deve essere maggiore di 0).

Se `addr` è NULL, il kernel sceglie l'indirizzo (allineato alla pagina) in cui creare la mappatura; questo è il metodo più portabile per creare una nuova mappatura.
Se `addr` non è NULL, il kernel lo prende come un suggerimento su dove posizionare la mappatura: su Linux, il kernel sceglierà un confine di pagina vicino (ma sempre superiore o uguale al valore specificato da `/proc/sys/vm/mmap_min_addr`) e tenterà di creare lì la mappatura.
Se esiste già un'altra mappatura, il kernel sceglie un nuovo indirizzo che può dipendere o meno dal suggerimento.

L'indirizzo della nuova mappatura viene restituito come risultato della chiamata.

Il contenuto di una mappatura di file viene inizializzato utilizzando `length` byte che iniziano a offset `offset` nel file (o altro oggetto) a cui fa riferimento il file descriptor `fd`. `offset` deve essere un multiplo della dimensione della pagina restituita da `sysconf(_SC_PAGE_SIZE)`.

Dopo il ritorno della chiamata a `mmap()` , il file descriptor, `fd`, può essere chiuso immediatamente senza invalidare la mappatura.

L'argomento `prot` descrive la protezione di memoria desiderata della mappatura (e non deve entrare in conflitto con la modalità aperta del file).

L'argomento flag determina se gli aggiornamenti alla mappatura sono visibili ad altri processi che mappano la stessa regione e se gli aggiornamenti vengono trasferiti al file sottostante (vedere pagina del manuale per i flags).

- Valore di ritorno: ritorna un puntatore all'area mappata in caso di successo , mentre in caso di errore ritorna `MAP_FAILED` e setta `errno`

int munmap(void addr[.length], size_t length):

- Questa chiamata di sistema elimina le mappature per l'intervallo di indirizzi specificato e fa sì che ulteriori riferimenti agli indirizzi all'interno dell'intervallo generino riferimenti di memoria non validi.
  La regione viene inoltre automaticamente non mappata al termine del processo. D'altra parte, la chiusura del descrittore di file non annulla la mappatura della regione.
  L'indirizzo `addr` deve essere un multiplo della dimensione della pagina (ma non è necessario che la lunghezza lo sia).
  Tutte le pagine contenenti una parte dell'intervallo indicato non sono mappate e i successivi riferimenti a queste pagine genereranno `SIGSEGV`.
  Non si tratta di un errore se l'intervallo indicato non contiene pagine mappate.
- Valore di ritorno: in caso di successo ritorna 0, in caso di errore invece ritorna -1 e viene settato `errno` (probabilmente a `EINVAL`)

pid_t getpid(void):

- Ritorna il PID del processo chiamante

pit_t getppid(void):

- Ritorna l'ID del processo padre del processo chiamante

~ Segnali ~

int kill(pid_t pid, int sig):

- Questa system call può essere utilizzata per inviare qualsiasi segnale a qualsiasi gruppo di processi o processo.
  Se `pid` è:
  - = 0 : `sig` viene inviato a ogni processo nel gruppo di processi del processo chiamante.
  - = -1 : `sig` viene inviato a tutti i processi per i quali il processo chiamante ha l'autorizzazione a inviare segnali, ad eccezione del processo 1 (init)
  - < -1 : `sig` viene inviato a ogni processo del gruppo di processi il cui ID è `pid`.
    Se `sig` è 0, non viene inviato alcun segnale, ma vengono comunque eseguiti controlli di esistenza e autorizzazione; questo può essere utilizzato per verificare l'esistenza di un ID di processo o di un ID di gruppo di processi che il chiamante è autorizzato a segnalare.

  Affinché un processo abbia l'autorizzazione a inviare un segnale, deve essere privilegiato oppure l'ID utente reale o effettivo del processo di invio deve essere uguale all'ID utente impostato reale o salvato del processo di destinazione.
  Nel caso di `SIGCONT` è sufficiente che i processi di invio e ricezione appartengano alla stessa sessione.
- Valore di ritorno: in caso di successo ritorna 0, altrimenti -1 e viene settato `errno`

int pthread_kill(pthread_t thread, int sig):

- Invia il segnale `sig` al thread `therad` (therad che esegue lo stesso processo del chiamante).
  Il segnale viene indirizzato in modo asincrono al thread.
  Se `sig` è 0, non viene inviato alcun segnale, ma viene comunque eseguito il controllo degli errori.
- Valore di ritorno: in caso di successo ritorna 0, altrimenti ritorna un numero di errore e nessun segnale viene inviato

int raise(int sig):

- Invia il segnale `sig` al processo o al thread chiamante.
  In un programma single-thread è equivalente a:
  `kill(getpid(), sig)`

In un programma miltithreaded è equivalente a:
`pthread_kill(pthread_self(), sig)`

Se il segnale provoca la chiamata di un gestore, `raise(..)` tornerà solo dopo il ritorno del gestore del segnale.

- Valore di ritorno: in caso di successo ritorna 0, altrimenti un valore diverso da zero

pthread_t pthread_self(void):

- Restituisce l'ID del thread chiamante.
- Valore di ritorno: ha sempre successo e restituisce l'ID del thread chiamante

int sigqueue(pid_t pid, int sig, const union sigval value):

- Invia il segnale specificato in `sig` al processo il cui PID è specificato in `pid`.
  I permessi richiesti per inviare un segnale sono gli stessi di `kill(..)`.
  Come con `kill(..)`, il segnale nullo (0) può essere utilizzato per verificare se esiste un processo con un dato PID.
  L'argomento valore viene utilizzato per specificare un elemento di dati di accompagnamento (un numero intero o un valore puntatore) da inviare con il segnale e ha il seguente tipo:

```
 union sigval {
   int   sival_int;
   void *sival_ptr;
};
```

Se il processo ricevente ha installato un gestore per questo segnale utilizzando il flag `SA_SIGINFO` su `sigaction(..)`, allora può ottenere questi dati tramite il campo `si_value` della struttura `siginfo_t` passato come secondo argomento al gestore. Inoltre, il campo `si_code` di quella struttura verrà impostato su `SI_QUEUE`.

- Valore di ritorno: in caso di successo viene ritornato 0 e indicato che il segnale è stato messo in coda al processo di ricezione con successo. In caso di errore viene restituito -1 e viene settato `errno`

int pthread_sigqueue(pthread_t thread, int sig, const union sigval value):

- Simile alla `sigqueue(..)`, invia un segnale ad un thread dello stesso processo del thread chiamante.
- Argomenti:
  - `thread` : ID di un thread nello stesso processo del chiamante
  - `sig` : specifica il segnale da inviare
  - `value` : specifica dati che accompagnano il segnale
- Valore di ritorno: in caso di successo ritorna 0, altrimenti un numero

int sigaction(int signum, const struct sigaction *_Nullable restrict act, struct sigaction *_Nullable restrict oldact):

- Questa system call viene utilizzata per modificare l'azione intrapresa da un processo alla ricezione di un segnale specifico.
  `signum` specifica il segnale e può essere qualsiasi segnale valido tranne `SIGKILL` e `SIGSTOP`.
  Se `act` non è NULL, la nuova azione per il segnale `signum` viene installata da `act`.
  Se `oldact` non è NULL, l'azione precedente viene salvata in `oldact`.
  La struttura `struct sigaction` è definita come segue:

```
struct sigaction {
  void     (*sa_handler)(int);
  void     (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t   sa_mask;
  int        sa_flags;
  void     (*sa_restorer)(void);
};
```

  Su alcune architetture è coinvolta un'unione: non assegnare sia a `sa_handler` che a `sa_sigaction`.
  Il campo `sa_restorer` non è destinato all'uso applicativo (POSIX non specifica un campo sa_restorer), per ulteriori dettagli sullo scopo di questo campo possono essere trovati in `sigreturn(..)`.
  `sa_handler` specifica l'azione da associare a `signum` e possono essere varie (vedi valori da terminale).
  Se `SA_SIGINFO` è specificato in `sa_flags`, allora `sa_sigaction` (invece di `sa_handler`) specifica la funzione di gestione del segnale per `signum` (questa funzione riceve tre argomenti, come descritto di seguito).
  `sa_mask` specifica una maschera di segnali che deve essere bloccata (vale a dire aggiunta alla maschera di segnale del thread in cui viene richiamato il gestore del segnale) durante l'esecuzione del gestore del segnale.
  Inoltre, il segnale che ha attivato il gestore verrà bloccato, a meno che non venga utilizzato il flag `SA_NODEFER`.
  `sa_flags` specifica un insieme di flag che modificano il comportamento del segnale.
  Tale comportamento è formato dall'OR bit a bit di zero o più dei segnali specificati alla pagina del manuale (vedi da terminale).
- Valore di ritorno: in caso di successo ritorna 0, altrimenti ritorna -1 e setta `errno`

int pause(void):

- Fa si che il processo chiamante (o thread) resti inattivo finché non viene inviato un segnale che termina il processo o provoca l'invocazione di una funzione di cattura del segnale.
- Valore di ritorno: ritorna solamente quando è stato catturato un segnale e la funzione di cattura del segnale è ritornata. In questo caso ritorna -1 ed `errno` viene settato a

`EINTR`

int sigemptyset(sigset_t *set):

- Questa consente la manipolazione di set di segnali POSIX: inizializza il set di segnali fornito da `set` a vuoto, con tutti i segnali esclusi da tale set.
- Valore di ritorno: in caso di successo ritorna 0, altrimenti -1

int sigfillset(sigset_t *set):

- Questa consente la manipolazione di set di segnali POSIX: inizializza il set di segnali `set` a pieno (full), includendo tutti i segnali
- Valore di ritorno: in caso di successo ritorna 0, altrimenti -1

int sigaddset(sigset_t *set, int signum):

- Questa consente la manipolazione di set di segnali POSIX: aggiunge il segnale `signum` al set di segnali puntato da `set`
- Valore di ritorno: in caso di successo ritorna 0, altrimenti -1

int sigdelset(sigset_t *set, int signum):

- Questa consente la manipolazione di set di segnali POSIX: elimina il segnale `signum` dal set di segnali puntato da `set`
- Valore di ritorno: in caso di successo ritorna 0, altrimenti -1

int sigprocmask(int how, const sigset_t *_Nullable restrict set, sigset_t *_Nullable restrict oldset):

- Viene utilizzato per recuperare e/o modificare la maschera del segnale del thread chiamante.
  La maschera di segnale è l'insieme dei segnali la cui trasmissione è attualmente bloccata per il chiamante.
  Il comportamento della chiamata dipende dal valore di `how`, come segue:
  SIG_BLOCK
  L'insieme dei segnali bloccati è l'unione dell'insieme corrente e di `set` passato come argomento.

  SIG_UNBLOCK
  I segnali in `set` vengono rimossi dal set corrente di segnali bloccati. È consentito tentare di sbloccare un segnale che non è bloccato.

  SIG_SETMASK
  L'insieme dei segnali bloccati è impostato come `set` passato per argomento.

  Se `oldset` non è NULL, il valore precedente della maschera di segnale viene memorizzato in `oldset`.

Se `set` è NULL, la maschera del segnale rimane invariata (ovvero, come viene ignorata), ma il valore corrente della maschera del segnale viene comunque restituito in `oldset` (se non è NULL).

L'uso di `sigprocmask(..)` non è specificato in un processo multithread; vedere `pthread_sigmask(..)`

- Valore di ritorno: in caso di successo ritorna 0, altrimenti ritorna -1 setta `errno`

int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset):

- E' identica alla funzione `sigprocmask(..)` con la differenza che il suo utilizzo nei programmi multithread è specificato esplicitamente da POSIX.1. (è MT-safe)
- Valore di ritorno: in caos di successo ritorna 0, altrimenti un numero di errore

int sigwait(const sigset_t *restrict set, int *restrict sig):

- Sospende l'esecuzione del thread chiamante fino a che uno dei segnali specificati in `set` non diventa **pendig**.
  La funzione accetta il segnale (lo rimuove dall'elenco dei segnali pendig) e restituisce il numero del segnale in `sig`.
  Il funzionamento di `sigwait(..)` è lo stesso della `sigwaitinfo(..)`, tranne per il fatto che:
  `sigwait(..)` restituisce solo il numero del segnale, anziché una struttura `siginfo_t` che descrive il segnale;
  I valori restituiti delle due funzioni sono diversi;
- Valore di ritorno: in caso di successo ritorna 0, altrimenti un numero positivo della lista in ERROR

int sigwaitinfo(const sigset_t *restrict set, siginfo_t *_Nullable restrict info):

- Sospende l'esecuzione del thread chiamante finché uno dei segnali in `set` non è **pendig** (se uno dei segnali nel set è già in sospeso per il thread chiamante, la funzione tornerà immediatamente).
  La funzione rimuove il segnale dall'insieme dei segnali pendig e restituisce il numero del segnale come risultato della sua funzione.
  Se l'argomento `info` non è NULL, il buffer a cui punta viene utilizzato per restituire una struttura di tipo `siginfo_t` (vedere `sigaction(..)`) contenente informazioni sul segnale.
  Se più segnali in `set` sono pendig per il chiamante, il segnale recuperato da `sigwaitinfo(..)` viene determinato secondo le consuete regole di ordinamento (vedere `signal(..)` per ulteriori dettagli).
- Valore di ritorno: ritorna il numero di un segnale. In caso di insuccesso ritorna -1 e setta `errno`