%d interi
%ld long
%c char

RANDOM
<stdlib.h>

srand(time(null)) imposta il seme iniziale per generare i numeri casuali.
t*ime:* restituisce l'ora corrente; in pratica ad ogni esecuzione il valore di time cambia perchè il tempo scorre quindi l'argomento di srand è sempre diverso
int rand()%n
restituisce un numero casuale tra 0 e n-1
[NB. se non si imposta srand ad ogni esecuzione vengono generati sempre gli stessi numeri casuali]

ALLOCAZIONE DINAMICA
<stdlib.h>

void *malloc(size_t size)
retituisce l'indirizzo dell'area dimemoria creata di dimensione size
[NB. es. per allocare 10 interi: int *punt=(int*)malloc(10*sizeof(int));]

void *calloc(size_t nitems, size_t size)
restituisce l'indirizzo dell'area di memoria appena creata
nitems: numero di elementi da allocare
size: grandezza di ogni elemento (es. sizeof(int))
[NB. è uguale alla malloc solo che inizializza tutti gli elementi creati a 0]

void *realloc(void *ptr, size_t size) rialloca un area di memoria
ptr: puntatore allare da riallocare
size: la nuova grandezza dell'allocazione puntata da ptr

void free(void *ptr) delloca l'are di memoria puntata da ptr

PROCESSI
<sys/wait.h> <unistd.h>

int fork() crea un processo figlio e restituisce il pid
se pid==0 processo figlio
se pid==1 processo padre
se pid==-1 errore nella creazione del processo

getpid() per conoscere il pid del processo corrente (get pid)

getppid() per conoscere il pid del padre (get parental pid)

exit(stato) provoca la terminazione del processo che la invoca restituendo al padre un messaggio dato dallo stato (es exit(2))
[oss. esiste anche **_exit(stato)** che termina comunque il processo ma fa delle operazioni in meno come per esempio non svuota il buffer dell'I/O]

wait(int *status) il padre attende che il figlio finisca l'esecuzione (int status; wait(&status);)
ritorna il pid del processo che termina l'esecuzione

status: conterrà lo stato di terminazione del processo (essendo in esadecimale bisogna traslarlo di 8 byte con lo shift verso destra [>>8] oppure con la macro WEXITSTATUS(status) stesso effetto) per fare un controllo su tale stato bisogna fare il cast a char essendo di 8 bit (es.(char)st==0)
se stato==0 terminato correttamente (o valore della exit(stato))
se stato !=0 terminato involontariamente

sleep(unsigned int second) sospende il processo per second secondi
ritorna il numero di secondi se va a buon fine
0 errore

usleep(useconds_t usec) sospende il processo (in millisecondi)
ritorna 0 se va a buon fine
!=0 errore

SEMAFORI

<sys/sem.h> <sys/ipc.h> <sys/types.h>

int semget(key_t key, int num_sem, short flags)
serve per creare un gruppo di semafori, definisce un array di più semafori.
restituisce l'identificativo IPC del set di semafori se va a buon fine
-1 errore

key: è una chiave necessaria per la generazione del set di semafori (si può usare la system call ftok() per generarlo univocamente) viene confrontato con le chiavi già esistenti nel kernel e in base al flag si opera.
key_t ftok(char * path, char id) Restituisce una chiave ottenuta combinando l'inode number e il minor device number del file indicato, il carattere indicato come secondo argomento [es: mykey= ftok (".",'a');]
[NB. si può usare utilizzare IPC_PRIVATE (equivale a 0) al posto della key, in questo modo si omette la chiave e si può accedere al set solo tramite l'identificativo restituito da semget]

num_sem: indica il numero di semafori da creare nel gruppo (es. un insieme di stampanti fanno parte dello stesso gruppo però hanno semafori diversi)

flags: indica il modo di creazione
ci sono due costanti:
IPC_CREAT usato da solo crea il set di semafori se non esistono all'interno del kernel, altrimenti restituisce l'identificativo del set esistente
IPC_EXECL viene usato insieme a IPC_CREAT e controlla l'esistenza dell'identificativo del set di semafori nel kernel, se esiste restituisce -1.
In pratica è un controllo sulla creazione che la fa fallire se già esiste l'identificativo nel kernel
[NB. si possono dare dei permessi al set di semafori durante la creazione [es. semget(key, 1, IPC_CREAT|IPC_EXECL|0664)] dove 0664 sono i permessi]

ogni gruppo è identificato dalla struttura:

```
struct semid_ds{
    struct ipc_perm sem_perm; /* info sul set di semafori come il proprietario o diritti
                                 di accesso */
    time_t sem_otime;         /* l'ultima volta che viene chiamato semctl() */
    time_t sem_ctime;         /* l'ultima volta che viene modificata la struttura */
    unsigned long int sem_nsems;} /* numero di semafori nel set */
```

[NB.questa struttura è interna al linux e il programmatore non è interessato]

Ogni semaforo del gruppo è una struttura dati, contenente:

```
struct sem {
  pid_t   sempid;       /* pid dell'ultima operazione */
  ushort  semval;       /* valore corrente del semaforo */
  ushort  semncnt;      /* numero di processi che aspettano l'incremento di semval quindi
                           aspettano che le risorse diventino disponibili*/
  ushort  semzcnt; };   /* numero di processi che aspettano semval=0 quindi è aspettano
                           di utilizzare le risorse al 100% (in modo eslusivo) */ };
```

int semop(int sem_id, struct sembuf *operazioni, int num_elementi)
serve per operare sui semafori
restituisce 0 se le operazioni sono andare a buon fine
-1 altrimenti

sem_id: indica il gruppo di semafori creati con semget
operazioni: è una struttura che indica su quale semaforo agire e quale operazione effettuare (oss. si può far riferimento a tale struttura con &operazioni)
struct sembuf sem; questa struttura contiene al suo interno i campi:
sem_num: numero del semaforo all'interno del gruppo
sem_op: l'operazione da eseguire sul semaforo sem_num all'interno del set [NB. il primo semaforo ha sem_num=0]
se sem_op <0 viene eseguita la wait()
se semval >=|sem_op| il processo si sospende subito
se semval <|sem_op| il campo semcnt viene incrementato e il processo si sospende fin quando:
semval >=|sem_op|
in questo caso il valore di
semcnt viene decrementato e il valore del
semaforo sarà semval-=|sem_op|
se sem_op >0 viene eseguita la signal() ovvero viene addizionato a s
emval sem_op semval+=sem_op questa non sospende il processo
se sem_op =0 viene eseguita la wait_for_zero
se semval =0 il processo si sospende
se semval !=0 viene incrementato il valore di semzcnt in modo da indicare che un processo è in attesa che semval=0
sem_flg: serve per compiere delle operazioni particolari
IPC_NOWAIT se si vuole mettere un processo in stato sleep ritorna un errore
SEM_UNDO in caso di errore viene riportato nello stato precedente il semaforo e continua l'esecuzione
num_elementi:indica il numero di elementi allocati per la struttura operazioni, il numero di operazioni da efettuare

i**nt semctl(int sem_id, int sem_num, int cmd, union semun arg)**
serve per impostare e ricavare valori del semaforo [...ctl control]
restituisce >0 se va a buon fine (nel caso in cui si usi con tre argomenti) o 0 (nel caso in cui si usi con 4 argomenti)
-1 errore

sem_id: indica il gruppo di semafori creati con semget
sem_num: indica il semaforo nel gruppo
cmd: indica il comando. Ci sono varie costanti che si riferiscono a varie operazioni

IPC_STAT legge i dati dell'insieme di semafori, copiando il contenuto della relativa struttura semid_ds all'indirizzo specificato nella union buf (sem_num vene ignorato)
IPC_SET permette di modificare i permessi ed il proprietario dell'insieme. Modificando i valori della struttura semid_ds puntata da buf della union (sem_num viene ignorato)
IPC_RMID Rimuove tutti i semafori del set dal kernel. (il campo sem_num viene ignorato)
GETALL serve per ottenere i valori correnti (semval di sem) di tutti i semafori in un set (sem_num viene ignorato)
GETNCNT Restituisce il numero di processi attualmente in attesa di risorse (il campo semncnt della struct sem) (arg viene ignorato)
GETPID Restituisce il PID del processo che ha effettuato l'ultima chiamata semop (campo sempid) (arg viene ignorato)
GETVAL Restituisce il valore di un singolo semaforo all'interno del set (campo semval di sem) (arg viene ignorato)
GETZCNT Restituisce il numero di processi attualmente in attesa di utilizzo delle risorse del 100% (campo semzcnt) (arg viene ignorato)
SETALL Imposta tutti i valori dei semafori del set contenuti nei campi dell'array della union aggiornando sem_ctime di semid_ds
SETVAL Inizializza il semaforo al valore passato dall'argomento val della union, viene aggiornato il campo sem_ctime di semid_ds

union semun: è una struttura simile alla struct solo che quando si invoca un campo tutti gli altri sono inaccessibili
tale struttura contiene i campi:
int val: usato per impostare il semaforo viene impostato da SETVAL
struct *buf: usato per IPC_STAT e IPC_SET Rappresenta una copia della struttura dati interna al semaforo utilizzato nel kernel
array: Un puntatore utilizzato da GETALL e SETALL. Deve puntare a una serie di valori interi da utilizzare nella creazione o il recupero di tutti i valori del semaforo in un set.
__buf e __pad: sono due dati che vengono usati all'interno del kernel a cui il programmatore non è interessato

MEMORIA CONDIVISA

<sys/shm.h> <sys/ipc.h> <sys/types.h>
In linux i processi creati sono indipendenti e non condividono nessun tipo di risorsa, l'unico modo è creare un'area condivisa

int shmget(key_t key, int size, int flag)
crea una memoria condivisa tra processi [shm shared memory]
resituisce l'identificativo dell'area di memoria se va a buon fine
-1 errore
key: chiave per identificare la SHM in maniera univoca nel sistema [NB. valgono le stesse cose di semget]
size: dimensione in byte della memoria condivisa
flag: uguale a semget quindi IPC_CREAT e IPC_EXECL, con i permessi da aggiungere

void* shmat(int shmid,const void *shmaddr, int flag)
collega il segmento di memoria allo spazio di indirizzamento del chiamante
restituisce l'indirizzo del segmento collegato se va a buon fine, bisogna fare il cast essendo void*
-1 errore
shmid: identificatore del segmento di memoria dato da shmget

shmaddr: indirizzo dell area di memoria del processo chiamante al quale collegare il segmento di memoria condivisa. Se 0, viene automaticamente scelto.
f*lag*: IPC_RDONLY per collegare in sola lettura

int shmctl(int ds_shm, int cmd, struct shmid_ds * buff)
serve per invocare l'esecuzione di un comando sulla shared memory
restituisce >0 se va a buon fine
-1 errore
ds_shm: descrittore della memoria condivisa su cui si vuole operare dato da shmget
cmd: i comandi possibili (come semctl)
IPC_STAT Recupera la struttura shmid_ds per un segmento, e lo memorizza l'indirizzo dell'argomento buf
IPC_SET Imposta il valore del membro ipc_perm della struttura shmid_ds per un segmento. Prende i valori dalla argomento buf.
IPC_RMID Segna un segmento per la rimozione, viene rimosso solo quando non vi sono più processi attaccati
SHM_LOCK impedisce che il segmento venga swappato o paginato
buff: è una struttura che contiene le informazioni sulle operazioni da fare (come semctl)

```
struct shmid_ds {
    struct ipc_perm shm_perm;       /* permessi */
    int     shm_segsz;              /* dimensione del segmento (bytes) */
    time_t  shm_atime;              /* ultima volta collegato shmat() */
    time_t  shm_dtime;              /* ultima volta scollegato shmdt() */
    time_t  shm_ctime;              /* ultima volta aggiornato shmat() */
    unsigned short  shm_cpid;       /* pid del creatore */
    unsigned short  shm_lpid;       /* pid dell'ultimo operatore */
    short   shm_nattch;             /* no. di operatori collegati */
    unsigned short   shm_npages;    /*
    unsigned long   *shm_pages;      area privata non accessibile
    struct vm_area_struct *attaches; */                 };
```

int shmdt (char * shmaddr)
rimuove il segmento quando non è più necessario, in realtà quello che fa è decrementare la variabile shm_nattach di uno, quindi, a differenza di IPC_RMID anche se i processi sono collegati vengono staccati, la rimozione del segmento dal kernel avviene solo quando shm_nattch =0

esempio di creazione shm

```
key_t chiave = ftok("./eseguibile", 'k');
int ds_shm = shmget(chiave,1024,IPC_CREAT|IPC_EXCL|0664); //se tolgo IPC_EXECL non ho bisogno di richiamare shmget sotto
char * p; char iniz[] = "Questa è una prova";
if(ds_shm >= 0) {
        // la risorsa è stata appena creata
        p = (char*) shmat(ds_shm, NULL, 0);
        strncpy(p, iniz, sizeof(iniz)); // inizializza
} else {
        // la risorsa già esiste
        ds_shm = shmget(chiave, 1024, 0664);  // se la risorsa esiste già bisogna richiamare di nuovo shmget senza IPC_CREAT E IPC_EXECL
        p = (char*) shmat(ds_shm, NULL, 0);}
...
printf("Contenuto SHM: %s\n", p);
```

SCAMBIO DI MESSAGGI

<aya/types.h> <sys/ipc.h> <sys/msg.h>
Lo scambio di messaggi tra processi in UNIX avviene solo indirettamente tramite una mailbox

int msgget(key_t key, int msgflg)
istanzia una coda di messaggi
restituisce l'identificativo della mailbox se va a buon fine
-1 errore
key: identificativo univoco della mailbox, valgono le stesse cose di shmget e semget
msgflag: è uguale al campo flag di semget e shmget, se si vuole usare una coda già istanziata msgflag=0
[NB. Dimensione della coda e numero massimo di messaggi sono decisi autonomamente dal sistema operativo]

int msgsnd(int msqid, struct msgbuf * msgp, int msgsz, int msgflg)
consente l'invio (e accodamento) di un messaggio verso la mailbox
restituisce 0 se va a buon fine
-1 errore
msqid: identificativo della coda in cui scrivere il messaggio, dato da msgget
*msgbuf *msgp*: puntatore alla struttura in cui il messaggio è contenuto
struct msgbuf {
long message_type;
char message_text [MAX_SIZE]; }
message_type: consente di selezionare ed estrarre un messaggio nella coda in qualsiasi posizione, assumendo il valore del messaggio selezionato
se assume il valore del pid del mittente si può realizzare una comunicazione indiretta simmetrica ??
message_text: contiene il messaggio e può essere di qualsisi tipo ex. char
msgsz: contiene la dimensione in byte del messaggio. Dato dalla dimensione di msgbuf - la dimensione del campo message_type [sizeof(struct mymsgbuf)-sizeof(long)]
msgflg: consente di avere delle caratteristiche per il produttore (mittente)
se msgflg=0 la send blocca il processo se la mailbox è piena
se msgflg=IPC_NOWAIT la send ritorna -1 e non accoda il messaggio se la mailbox è piena il processo non si blocca

i**nt msgrcv (int msqid, struct msgbuf * msgp, int msgsz, long mtype, int msgflg)**
consente la ricezione di un messaggio da una mailbox
restituisce il numero di byte copiati nel buffer
-1 errore
msqid: identificativo della coda da cui prelevare il messaggio, dato da msgget
*msgbuf *msgp*: puntatore al buffer in cui è salvato il messaggio da consumare
msgsz: lunghezza del messaggio data dalla dimensione
sizeof(struct mymsgbuf)-sizeof(long)
mtype: seleziona il messaggio all'interno della coda
se mtype=0 viene prelevato il primo messaggio della coda (ovvero quello inviato da più tempo)
se mtype>0 viene prelevato il primo messaggio dalla coda il cui campo message_type sia pari al valore di mtype
se mtype<0 viene prelevato il primo messaggio dalla coda il cui campo message_type abbia una valore minore o uguale a |mtype|

msgflg:se msgflg=0 indica una ricezione bloccante, se non ci sono messaggi da consumare nella mailbox, il processo si sospende sulla msgrcv fino al giungere del messaggio
se msgflg=IPC_NOWAIT la ricezione non è bloccante, se non ci sono messaggi viene restituito -1

int msgctl (int msgqid, int cmd, struct msqid_ds * buf)
serve per invocare un comando sulla mailbox
msgqid: identificativo della coda a cui applicare il comando
cmd: il comando da applicare
IPC_STAT Recupera la struttura msqid_ds per una coda, e lo memorizza all'indirizzo dell'argomento buf (legge la coda senza consumare il messaggio)
IPC_SET Imposta il valore del membro ipc_perm della struttura msqid_ds per una coda, prende i valori dall' argomento buf (modifica le caratteristiche coda)
IPC_RMID Rimuove la coda dal kernel
*msqid_ds *buf*: è una struttura interna al kernel che viene mantenuta per ogni coda di messaggi

```
msqid_ds struct {
    struct ipc_perm msg_perm; /*contiene le informazioni dei permessi per l'accesso
                              la coda di messaggi e le informazioni sul creatore della coda */
    struct msg * msg_first;   /* puntatore al primo messaggio in coda */
    struct msg * msg_last;    /* puntatore all'ultimo messaggio nella coda */
    msg_stime time_t;         /* Ultima volta che è stato invocato msgsnd */
    msg_rtime time_t;         /* Ultima volta che è stato invocato msgrcv */
    msg_ctime time_t;         /* Ultima volta che è stata aggiornata la struttura */
    struct wait_queue * wwait;
    struct wait_queue * rwait;
    msg_cbytes USHORT;
    USHORT msg_qnum;          /* Numero di messaggi attualmente in coda */
    msg_qbytes USHORT;        /* Numero massimo di byte in coda (comprensivo di
                                 tutti i messaggi) */
    USHORT msg_lspid;         /* Pid dell'ultimo processo che ha chiamato msgsnd */
    USHORT msg_lrpid;         /* PID del processo che ha recuperato l'ultimo m
                                 essaggio */ };
```

THREAD
<pthread.h> <sys/types.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)
Crea un nuovo thread e lo rende eseguibile
restituisce 0 se va a buon fine
-1 errore
thread:(output) è un identificatore del thread creato
attr:(input) serve a settare gli attributi del thread
settato a NULL imposta gli attributi di default
start_routine:(input) puntatore (di tipo void *) alla funzione C (starting_routine) che verrà eseguita una volta che il thread è creato come run di java
arg:(input) argomento (di tipo void *) che può essere passato alla funzione C. Se la funzione richiede paramentri multipli devono essere passati via array o struct

void pthread_exit(void *value_ptr)
usata per terminare un thread esplicitamente
value_ptr: (input) indica lo stato di uscita del thread
[oss: Se usata nel programma principale (che potrebbe terminare prima di tutti i thread), gli altri thread continueranno ad eseguire]

int pthread_join(pthread_t thread_id, void **value_ptr)
blocca il chiamante finché il thread specificato non termina
ritorna 0 se va a buon fine
!=0 errore
thread_id: identificativo del thread che deve terminare per poter continuare l'esecuzione
value_ptr: restituisce lo stato di terminazione del thread thread_id

[NB. un thread deve essere joinabile affinchè su di esso si possa chiamare la funzione join
bisogna modificare gli attribiti di un thread

```
pthread_attr_t attr; /* un tipo thread attribute */
pthread_attr_init(&attr); /* Inizializza la struttura dati contenente gli attributi dei
                             thread ai valori di default */
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE); /* rende
joinable un thread */
...
pthread_create(&id,&attr, start_r, (void *) data);
...
pthread_join(id, (void **) &status);
```

int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)
setta gli attributi di un thread
restituisce 0 in caso di successo
!=0 errore
attr: (input) indirizzo della struttura degli attributi da settare per il thread
dethachstate: costante che setta gli attributi in due modi:
PTHREAD_CREATE_DETACHED nessun thread può effettuare la join() sul thread stesso
PTHREAD_CREATE_JOINABLE un altro thread può effettuare la join sul thread stesso (è il valore di defaut)]

pthread prevede anche la mutua esclusione tramite determinate primitive:
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
Crea un nuovo mutex e lo inizializza come “sbloccato” (unlocked). [init=initialized]
restituisce 0 se va a buon fine
!=0 errore
mutex: (output) è un identificatore del mutex creato
mutexattr: (input) per settare gli attributi del mutex
settato a 0 attributi di default

int pthread_mutex_destroy (pthread_mutex_t *mutex)
dealloca il mutex
restituisce 0 se va a buon fine
!=0 errore
mutex: identificativo del mutex da deallocare

i**nt pthread_mutex_lock(pthread_mutex_t *mutex)**
Un thread invoca la lock su un mutex per acquisire l'accesso in mutua esclusione alla sezione critica relativa al mutex. Se il mutex è già acquisito da un altro thread, il chiamante si blocca in attesa di una unlock.

mutex: identificativo della sezione critica (mutex) in cui entrare

int pthread_mutex_trylock(pthread_mutex_t *mutex)
Analoga alla lock, ma non bloccante

restituisce 0 in caso di successo

EBUSY se il lock è bloccato da un altro thread

un altro codice d'errore altrimenti

int pthread_mutex_unlock(pthread_mutex_t *mutex)
Un thread invoca la unlock su un mutex per rilasciare la sezione critica, e per consentire quindi l'accesso ad un altro thread precedentemente bloccato.

restituisce 0 se va a buon fine

!=0 errore

int pthread_cond_init(pthread_cond_t *cond, constpthread_condattr_t *attr)
Crea una nuova Condition variable e lo inizializza come "sbloccato" (unlocked)

restituisce 0 se va buon fine

!=0 errore

cond: (output) identificativo della variabile condition creata

attr: (input) attributi sulla variabile condition

settato a 0 attributi di default

int pthread_cond_destroy(pthread_cond_t *cond)
dealloca la variabile condition

restituisce 0 se va buon fine

!=0 errore

cond: identificatore della variabie condition da deallocare

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
Blocca il thread chiamante finché non si invoca la signal sulla Condition variable

restituisce 0 se va buon fine

!=0 errore

cond: (input) identificativo della condition variable

mutex: (input) il thrad si mette in attesa sul mutex ???

int pthread_cond_signal(pthread_cond_t *cond)
Viene svegliato un thread tra quelli in attesa sulla variabile condition [NB. l thread è scelto non-deterministicamente]

restituisce 0 se va buon fine

!=0 errore

cond: (input) identificativo della condition variable

<sys/types.h> contiene le definizioni dei tipi: es. gid_t, key_t, pid_t, pthread_t, pthread_cond_t, pthread_mutex_t ecc.. (tutti i tipi che terminano con 't')

<unistd.h> contiene varie costanti e funzioni in particolare contiene: execl(), execl(), fork(), getpid(), getppid() ecc...

<sys/ipc.h> contiene tutte le costanti IPC es IPC_CREATE, IPC_EXECL ecc... contiene anche ftok()