Chiamate di sistema per la gestione di processi

- **Identificatori di processo**

```
#include <sys/types.h>      // typedef long    pid_t;
                            // typedef long    uid_t;
                            // typedef long    gid_t;
pid_t       getpid(void);   //pid del chiamante;

pid_t       getppid(void);  //pid del padre

uid_t       getuid(void);   //real-user-id del chiamante

uid_t       geteuid(void);  //effective-user-id del chiamante

gid_t       getgid(void);   //real-group-id del chiamante

gid_t       getegid(void);  //effective-user-id del chiamante
```

- **Creazione di un processo (1)**

int fork()

Crea un nuovo processo.
Al processo padre ritorna il PID del figlio (il padre puo' avere piu' figli)
Al processo figlio ritorna 0 (un processo puo' avere 1 solo padre → getppid)

Sequenza di operazioni di fork:

- alloca una entry nella tabella dei processi
- assegna un unico ID al figlio
- copia il contesto del padre
- incrementa il nr. di riferimenti ai file associati con il processo
- ritorna l'ID del figlio al padre, al figlio 0

NB:

1) Limiti (in Area U): nr. di processi da creare.
2) L'offsett dei file e' in File Table, che e' condivisa! se un figlio esegue un lseek, modifica il padre.
3) Il padre viene copiato ma: il tempo viene inzializzato a 0

Esempio:

```
main()
{
     switch(f=fork()) {
     case -1:  printf("errore\n");
               exit(1);
     case  0:  printf("sono il figlio. pid=%d\n", getpid());
               exit(0);
     }
     printf("ID del figlio = %d\n",f);
}
```

- **Creazione di un processo (2)**

int vfork()

Non copia lo spazio di indirizzamento: versione efficiente di fork!!
Attenzione: modifica l'ambiente del padre!!

- **Terminazione di un processo**

void exit(int status)

Chiude tutti i descrittori aperti, rilascia memoria.
terminazione normale:
    return(0)-return(1) //definisce uno 'exit status'
    void exit(int status) //chiamata di sistema
    void _exit(int status) //funzione di libreria
terminazione anormale:
    void abort(void) //genera il segnale SIGABRT (trap?)
    ricezione di segnale

- **Attesa della terminazione**

Segnale SIGCHLD. Exit status raccolto dal padre. Wait e waitpid ritornano il pid del processo

pid_t wait(int *exitstatus);//blocca il chiamante finche' un figlio termina

pid_t waitpid(pid_t pid, int *exitstatus, int options); //blocca il padre
//finche' termina pid

#include <sys/resource.h>
pid_t wait3(int *stat, int option, struct usage *usage); //usage
// descrive risorse

pid_t wait4(pid_t pid, int options, struct usage *usage) ;

- **Modifica dello spazio di indirizzamento**

EXEC: chiamate di sistema che invoca un altro programma, sovra-scrivendo lo spazio di memoria di un processo con una copia di un programma eseguibile.
NB: L'INDIRIZZO DI RITORNO E' PERSO!

Operazione:
* accede al file (ricava l'inode)
* verifica che il file sia eseguibile
* verifica i permessi
* legge il file header
* legge il file e ricopre lo spazio del processo. Quando il kernel alloca una regione di codice per exec, verifica se il codice e' condiviso. Se non e' condiviso, assegna una nuova regione.
* rilascia l'inode

Vediamo in dettaglio le sei varianti:

int execl(char *path, char *arg0, ..., char *argN, NULL)

arg0 = nome file
arg1...argN = parametri

Esempio:

```
main()
{
     printf("Prova della execl");
     execl("/bin/echo", "echo", "di", "un", "programma",NULL);
     printf("errore!\n");
}
```

NB: Chiusura file! Se chiudo tutti i file:
for(i=0; i<20; i++) close(i);
prima di chiamare la execl, perdo lo standard output.

Posso fare:
for(i=0; i<20; i++) fcntl(i,F_SETFD,1);
Chiude i file solo se exec ha avuto successo.

Altri tipi di exec:
int execv(char *path, char *argv[])
int execle(char *path, char *arg0,...,NULL, char *envp[])
int execve(char *path, char *argv[], char *envp[])
int execlp(char *file, char *arg0,..., *argn, NULL)
int execvp(char *file, char *argv[])

NB: gli ultimi due ricercano con PATH

- **Esecuzione di comandi di shell**

#include <stdlib.h>
int system(const char *str)

```
int status;
if ((status=system(“who”)) <0) syserr(“system”):
```

. . .
→ Es. shell

IPC

Scopi di IPC:

- scambiare dati tra processi.
- sincronizzare l'esecuzione di processi

Un esempio primitivo sono i segnali. Il messaggio scambiato e' solo il nr. del segnale.

Altri meccanismi:
PIPE : sincronizzazione
MESSAGGI : scambio di dati
MEMORIA CONDIVISA : condivisione di memoria virtuale
SEMAFORI : sincronizzazione
SOCKET: comunicazione di rete

- **PIPE**

Canale di comunicazione tra processi. Implementato come file nei quali i blocchi sono messi in coda circolare (in System V). Quindi: allocazione di i-node (solo blocchi diretti); UFDT; file table; buffer Cache.

- PIPE senza nome (FIFO) → pipe(.);
- PIPE con nome (PIPE) → mkfifo; open(.); oppure mknod; open;

int pipe(int pfd[2])

File descriptors: pfd[0] → lato lettura; pfd[1] → lato scrittura

- Operazione di scrittura: scrive su pfd[1] - si blocca quando la pipe si riempie - EOF si introduce con close(pfd[1])
- Operazione di lettura: legge da pfd[0] - aspetta se la pipe e' vuota

Uso con 1 solo processo:

```
pipe(pfd);  // Ma: si blocca se scrivo piu' di 4096 byte
write(pfd[1], "messaggio",9);
switch(n=read(pfd[0],buf,sizeof(buf))){
     case -1: errore!
     case  0: EOF
     default: printf("%s",buf);
}
```

Problema con le pipe: i descrittori in file table sono condivisi solo all'interno dello stesso processo e tra padri e progenie.

Operazioni generali con le pipe:
1- generare la pipe
2- generare un figlio
3- il figlio chiude il descrittore in scrittura e legge
4- il padre chiude il descrittore in lettura e scrive

Perche' fork e exec sono distinte?
- ➔punto 3
- ➔dup

- **DUP**

int dup(int fd)

Copia il file descriptor nel primo posto disponibile della file table. Ritorna l'fd trovato.

Perche'? Con le PIPE, condividere il terminatore che legge con 0 e quello che scrive con 1, per usare lo standard input ed output

Pipe bidirezionali? **Problemi con una pipe:** il padre scrive e legge subito ➔blocco. Soluzione: usare 2 pipe!

- **DUP2**

int dup2(int fd, int fd1)

Specifica il valore del nuovo descrittore in fd1. fd1 viene chiuso se gia' aperto. Se fd=fd1, ritorna fd senza chiuderlo

NB: `dup(fd);` equivale a `fcntl(fd, F_DUPFD, 0);`
`dup2(fd,fd1);` equivale a `close(fd2); fcntl(fd1, F_DUPFD, fd1);`
Ma: 1) dup2 e' atomica 2) errno e' diversa

- **popen/pclose**

Esegue un comando. Riunisce le operazioni di creare una pipe, di creare un processo, chiudere i descrittori, eseguire un shell per eseguire il comando e aspettare la sua terminazione

FILE *popen(const char *cmd, const char *type);

//ritorna il puntatore se ok, NULL se errore

se type = “r”, il file pointer e’ connesso allo stdout, “w” allo stdin

int pclose(FILE *fp);

//ritorna stato di terminazione se ok, -1 se errore

```
//esempio di utilizzo
#include <stdio.h>

int main()
{
        FILE *fp;
        char linea[100];

        fp=popen("ls *.h","r"); if(fp==NULL) syserr("errore in popen\n");
        while(fgets(linea,100,fp)!=NULL)
        {
                if(fputs(linea,stdout)==EOF) syserr("errore in fputs");
        }
        if(pclose(fp)==-1) syserr("errore in pclose");
        exit(0);
}
```

- **Coprocessori**

Filtri che elaborano i dati. Un processo utilizza due pipe (una per canale).

- **FIFO**: come la pipe, ma con nome, quindi condivisa da piu' processi! Vengono tipicamente usati per implementare i messaggi tra due processi.

Esempio d'uso delle fifo:

```
{
      int fd, fd1;
      mknod("nomedellafifo", S_FIFO|0777);
      switch(fork()) {
      case -1:    errore!!
      case 0:     fd=open("nomedellafifo", O_WRONLY);
                  close(1); dup(fd); close(fd);
                  execlp("who","who",NULL);
      }
      switch(fork()) {
      case -1:    errore!!
      case 0:     fd1=open("nomedellafifo", O_RDONLY);
                  close(0); dup(fd1); close(fd1);
                  execlp("wc","wc",NULL);
      }
      wait(NULL);
}
```

MESSAGGI

I processi possono scambiarsi messaggi, cioe' informazioni di qualsiasi tipo.

4 chiamate di sistema:

1) creazione messaggio (ritorna un identificatore, **msqid**)

int msgget(key_t key, int msgflg);
chiave mnemonica |
(numero intero <>0 |
accessi

Se la chiave e' 0, la coda e' privata del processo e non puo' essere condivisa.

Collisioni tra chiavi: key = UID * MSGMNI + N
MSGMNI = nr. di entry in tabella globale messaggi (kernel). es:50
N = 0...MSGMNI-1

Accessi. Esempio: msgflg = 0600|IPC_CREAT|IPC_EXCL
->permessi di lettura e scrittura del proprietario
->crea una coda nuova
->notifica gli errori

Calcolo del msqid: msqid=N+msg_perm.seq*MSGMNI
Quando la coda viene cancellata, msg_perm.seq += 1
Questo meccanismo consente di evitare che un processo che ha memorizzato un msqid tenti di accedere alla coda dopo che sia cancellata.

2) Controllo

int msgctl(int msqid, int cmd, struct msqid_ds * buf)

cmd = IPC_STAT : informazioni sullo stato della coda
IPC_SET : modifica i dati della struttura
IPC_RMID : cancella la coda
NB: bisogna cancellare le code dopo l'uso! altrimenti restano nel kernel

3) invio messaggi

int msgsnd(int msqid, struct msgbuf *msgp, int msgz,
int msgflg)

msgz = nr di byte significativi
msgflg = modo di esecuzione

Template per il buffer messaggi:

```
struct msgbuf{
     long mtype;    /* deve essere >0 */
     char mtext[MSGMAX];
}
```

4) ricezione messaggi

int msgrcv(int msqid, struct msgbuf * msgp, int msgsz,
long msgtype, int msgflg)

msgsz = massima dimensione desiderata per il messaggio in arrivo
msgtype = 0 → 1o msg in coda
msgtype > 0 → 1o msg in coda con mtype=msgtype
msgtype < 0 → 1o msg in coda con piu' basso mtype <= msgtype
msgflg = modalita' di esecuzione. Esempio:
- msgflg = MSG_NOERR → i messaggi > msgsz vengono troncati
- msgflg = IPC_NOWAIT → esce subito con -1 se non c'e' mess.

MEMORIA CONDIVISA

- Usata per scambiare i dati tra processi: sincronizzazioni con semafori!
- La memoria condivisa stessa e' chiamata *segmento*
- Un processo puo' accedere a molti segmenti condivisi
- Procedura:

1) un segmento viene creato fuori dallo spazio di indirizzamento
2) ogni processo che vuole accedervi esegue una chiamata di sistema per mapparlo nel proprio spazio di indirizzamento
3) l'accesso ad un segmento coniviso e' uguale a quello ad una variabile locale
4) in System V, i segmenti possono restare mappati allo spazio di indirizzamento di un processo per lungo tempo-> efficienza!

4 chiamate di sistema:

Alloca il segmento e ritorna l'identificatore :

int shmget(key_t key, int nbytes, int flags)

Attacca (mappa) il segmento al processo e ritorna l'indirizzo:

char *shmat(snd segid, char *addr, int flags)

Stacca il segmento dal processo:

int shmdt(char *addr)

Operazioni di controllo:

int shmctl(int segid, int cmd, struct shmid_ds *sbuf)

SEMAFORI

Sincronizzano l'esecuzione. Nel System V un semaforo e' composto da:
* valore del semaforo
* PID dell'ultimo processo che ha manipolato il semaforo
* nr. di processi che aspettano che il sem. aumenti
* nr. di processi che aspettano che il sem. diventi 0

3 chiamate di sistema:

int semget(key_t key, int nsems, int flags)

Crea un array di semafori o si aggancia a semafori pre-esist.
es. semget((key_t) key, 1, 0600|IPC_CREATE)

int semop(int sid, struct sembuf (*ops)[], int nops)

Manipola i valori dei semafori. Ritorna il valore dell'ultimo semaforo elaborato.
struct sembuf = array di operazioni. nops = nr di operazioni.

```
struct sembuf{
     short sem_num; /* nr del semaforo */
     short sem_op;       /* operazione */
     short sem_flg;      /* opzioni */
}
```

int semctl(int sid, int snum, int cmd, char *arg)

realizza varie operazioni sull'insieme.
snum = numero; cmd = comando.

Semplice esempio d’uso dei semafori: protezione di risorsa critica.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

main()
{
     sid=semtran(key);
     P(sid);
     Risorsa_critica;
     V(sid);
}
int semtran(int key)
{
     int sid;
     sid=semget( (key_t)key, 1, 0600|IPC_CREATE);
     return(sid);
}
void op(int sid, int op)
{
     struct sembuf sb;
     sb.sem_num = 0;
     sb.sem_op=op;
     sb.sem_flg=0;
     semop(sid, &sb, 1);
}
void P(int sid)
{    op(sid,-1)
}
void V(int sid)
{    op(sid,1);
}
```

SEGNALI

*** Un segnale puo' essere lanciato in ogni istante, tipicamente dal kernel che vuole avvisare un processo di qualche evento eccezionale**
*** I segnali non portano informazione**
*** Il ricevitore non puo' sapere chi e' il trasmettitore**
*** I segnali non possono essere usati per la sincronizzazione**

Chiamate di sistema:
#include <signal.h>

int (*signal(int sig, int (*fcn)()))()

Stabilisce la funzione da attivare in risposta al segnale.

int kill(int pid, int sig)

Manda il segnale **sid** al processo **pid**. Se pid =0, il segnale viene mandato a tutti i processi dello stesso gruppo del trasmittente.

void pause()

Aspetta un segnale

unsigned alarm(unsigned secs)

Inizializza il clock dell'allarme. Quando il clock va a zero, il kernel manda il segnale SIGALRM al processo.

Esempio: costruzione di sleep

```
void sleep(int secs)
{
     void nullfcn();

     if (signal (SIGALRM, nullfcn) == BADSIG)
          syserr("signal");
     alarm(secs);
     pause();
}
static void nullfcn() { }
```