GESTIONE DELLA COMUNICAZIONE LOCALE TRA PROCESSI IN UNIX:

Il primo meccanismo di comunicazione tra processi locali in ambiente Unix e' stato il meccanismo delle **pipe**. Per **processo locale** si intende un processo residente su una macchina locale e non su una macchina remota. Una pipe sostanzialmente e' un oggetto su cui e' possibile leggere e scrivere come se fosse un comune file. Per creare una pipe si utilizza la seguente funzione C (pipe()):

```
int dp[2];
int esito;
dp=pipe(dp);
if(esito==-1)
{
        Printf("\n Impossibile creare la pipe");
}
```

La funzione pipe() quindi ritorna due possibili valori: 0 o -1. Viene ritornato 0 se la creazione della pipe mediante funzione ha esito positivo, altrimenti ritorna -1 se tale esito e' negativo. Una volta creata la pipe, e' possibile leggere da essa o scrivere su di essa. Per svolgere tali mansioni si usano rispettivamente le seguenti funzioni:

READ per la lettura;
WRITE, per la scrittura;

Per esempio, possiamo pensare di creare un processo figlio da un processo padre utilizzando la primitiva FORK(). Tale primitiva permette di creare un nuovo processo, chiamato per l'appunto **processo figlio** da un processo in esecuzione che diventa **processo padre**. Tale primitiva ritorna 0 se la creazione avviene con successo, oppure restituisce un -1 in caso di errore. A titolo di esempio si ha:

```
int pid;
pid=fork();
if(pid==-1)
{
        Printf("\n Errore nella creazione del processo!!");
}
```

A questo punto e' possibile fare in modo che il processo padre scrivi un generico messaggio che poi il processo figlio vada a leggere.

```
int pid, esito;
int dp[2];
char messaggio[]="prova!!"
esito=pipe(pd);          //creazione pipe

//controllo correttezza creazione pipe
If(esito==-1)
{
        Printf("\n Errore nella creazione della pipe!!");
}
pid=fork();      //creazione processo

//controllo creazione processo
```

```
If(pid==-1)
{
        Printf("\n Errore nella creazione del processo!!");
}

Switch(pid)
{

  Case 0:       //processo figlio
  {
        Close(pd[1]);
        Read(pd[0], messaggio, 7);
        break;
  }

 Default:
 {
        Close(pd[0]);
        write(pd[1], messaggio, 7);
        break;
 }
```

In particolare l'istruzione **close** permette di chiudere la pipe. Nel frammento di codice precedente e' stata creata una pipe e alternativamente apro e chiude uno estremo della stessa in base all'operazione da svolgere. Purtroppo la pipe ha una capacita' assai limitata, e quindi e' necessaria una sorta di sincronizzazione tra i processi. Per essere piu' chiari, se la pipe e' vuota il processo di lettura si blocca, mentre viceversa se la pipe e' piena il processo di scrittura si blocca. L'utilizzo della pipe comporta alcuni vantaggi tra cui permette la comunicazione tra piu' processi in relazione di parentela, ed inoltre la pipe viene cancellata dalla memoria una volta che i processi cessano di esistere. Purtroppo con le classiche pipe non e' possibile far comunicare due o piu' processi non imparentati. Per questa ragione si usano le pipe **fifo**, ossia delle code classiche. Tali code vengono generate mediante la seguente funzione C:

```
int esito;
esito=mkinfo(char *path, mode);
```

dove path e' il nome della fifo, mentre mode rappresenta i possibili permessi. Tale funzione restituisce uno 0 in caso positivo oppure un valore negativo in caso di errore. Per esempio:

```
if(esito<0)
{
        Printf("\n Errore nella creazione della coda!! ");
}
```

Una volta creata la coda, bisogna chiaramente accedervi. Per prima cosa la si apre, tramite la funzione **open** che ha la seguente sintassi:

```
descrittore coda=open(nome fifo, modalita' apertura);
```

Per chiudere una coda basta usare la funzione **close**:

```
close(descrittore coda);
```

Infine per distruggere (eliminare dalla memoria) una coda si usa la funzione **unlink:**

```
unlink("nome coda");
```

Per esempio:

```
int esito, fd;
char path="./prova/coda"
esito=mkfifo(path, 0664);       //creazione coda

//controllo correttezza creazione coda
if(esito<0)
{
        Printf("\n Impossibile creare coda!!");
}

//apertura coda
If(fd=open(path, O_RDONLY)==-1)
{
        Printf("\n Errore aperture coda!!");
}
Else
{
        Read(fd, messaggio, dimensione_messaggio);      //lettura msg
        Close(fd);              //chiudi e distruggi coda
        Unlink("path");
}
```

Per superare le ovvie limitazioni delle pipe, con la versione system V di Unix, sono state introdotte nuove tecniche di comunicazione tra processi residenti su una stessa macchina, che messe insieme rappresentano la cosi' detta IPC (Inter-Process Communication). Queste tecniche sono:

1. **code di messaggi**
2. **semafori**
3. **memoria condivisa**

Iniziamo a parlare delle code di messaggi. Una coda dei messaggi e' sostanzialmente una struttura dati stabilmente residente in memoria. Per creare una coda dei messaggi e' possibile utilizzare la funzione **msgget**, come mostrato nell'esempio che segue:

```
int d_coda;

key_t chiave=30;
```

```
d_coda=msgget(chiave, IPC_CREAT|0999);

//controllo apertura coda messaggi

If(d_coda==-1)

{

        Printf("\n Errore apertura coda messaggi!!");

}
```

Una volta creata la coda dei messaggi, e' possibile spedire segnali alla coda tramite la funzione:

```
msgctl(int msg_id, int commando, struct msqid_ds *buf);
```

Invece per inviare e ricevere messaggi dalla coda e' possible utilizzare rispettivamente le seguenti funzioni:

```
msgsnd();

msgrcv();
```

Per quanto riguarda la memoria condivisa, e' possibile creare intere parti di memoria da far condividere tra due o piu' processi. Per creare una memoria condivisa si usa la seguente funzione:

```
shmget();
```

mentre per operare su una memoria condivisa si utilizza la seguente funzione:

```
shmctl();
```

Chiaramente per poter operare sulla memoria condivisa e' prima necessario attaccare tale memoria al processo in questione, cosa questa che viene svolta utilizzando la seguente funzione:

```
shmat();
```

Infine per quanto riguarda i semafori, abbiamo che essi sono dei contatori che possono essere incrementati o decrementati. Per inizializzare un determinato semaforo si usa la funzione seguente:

```
sem_init();
```

Il primo argomento deve essere un puntatore all'oggetto semaforo, il secondo parametro (PSHARED) indica se il semaforo e' utilizzato nell'ambito dello stesso processo (valore uguale a 0) oppure se viene utilizzato anche da processi esterni (valore diverso da 0), ed infine il terzo parametro che viene utilizzato come valore di inizializzazione. Tramite la funzione:

sem_wait();

e' possibile sospendere il thread chiamante fintantoché il valore del semaforo e' diverso da zero. Utilizzando la funzione:

sem_trywait();

e' possibile decrementare il valore del semaforo, mentre viceversa utilizzando la seguente funzione:

sem_post();

tale valore viene incrementato di un'unita'. Infine per disallocare le risorse associate ad un semaforo si utilizza la funzione:

sem_destroy();