Sistemi Operativi

Fork

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){

        int N=3, i, pid;
        for (i=0; i<N; i++){

                pid = fork();
                if (pid==0){
                        printf("Sono il FIGLIO con PID: %d\n", getpid());
                        i=3;
                }
        }
        printf("Istruzione successiva \n");
        return 0;
```

La system call ***fork*** è una funzione che divide il processo chiamante e genera un nuovo processo figlio con un nuovo process id (PID). Il processo figlio avrà una copia uno ad uno dell'area di memoria del processo padre e quindi di tutte le variabili allocate (compresa una propria area heap). Per operare nel figlio bisogna utilizzare un if(pid==0) perchè il valore di ritorno della funzione fork assegnerà 0 alla variabile nell'area di memoria del figlio. Se il PID è uguale a -1 allora si è verificato un errore, altrimenti è il processo padre.

wait(NULL) con questa funzione il processo padre aspetta che un figlio termini

Execl

```c
if( pid == -1 ) {
        perror("fork fallita");
        exit(1);
}

if( pid == 0 ) {

        printf("Sono il processo figlio, con PID %d\n", getpid());
        printf("Command line passata al main():\n");
        printf("argv[0]: %s\n", argv[0]);
        printf("argv[1]: %s\n", argv[1]);
        printf("argv[2]: %s\n", argv[2]);
        printf("Attendo 3 secondi\n");

        for(i=0; i<3; i++) {
                sleep(1);
                printf(".\n");
         }

        execl("/bin/cp", "cp", argv[1], argv[2], NULL);

        perror("Se arrivo in questo punto, qualcosa è andato storto...\n");

        kill(getppid(), SIGKILL);
        exit(1);
```

La funzione exec sostituisce il processo in uso avviando un nuovo file eseguibile e sostituendo la sua area dati con quella del nuovo eseguibile.

Risorse IPC

Comunicazione tra processi mediante strutture dati rese disponibili dal kernel

- Memoria condivisa (SHM : shared memory segments)
- Semafori (SEM: semaphore arrays)
- Code di messaggi (MSG: queues)

Oggetti (o risorse) IPC

Ogni risorsa IPC è identificata da un valore **univoco nel sistema**, denominato chiave (IPC key)

Una IPC key può essere

- "Cablata" nel codice (problemi di consistenza)
- Generata dal sistema mediante la primitive **ftok**
- **IPC_PRIVATE** (equivale a 0), costante usata per creare una nuova entry priva di chiave (in questo caso, la risorsa è **accessibile solo al processo creatore ed ai suoi figli**)

Due processi che usano gli **stessi parametri** ottengono le stesse chiavi

- Si evita di cablare una chiave nel programma, utilizzando invece un **percorso** (distinto da altri programmi)

```
key_t ftok(char * path, char id);

key_t mykey;
mykey = ftok("./eseguibile", 'a');
```

```
int …ctl (int desc, …, int cmd, …);
```

desc: indica il descrittore della risorsa;

cmd: specifica del comando da eseguire:

- IPC_RMID: rimozione della risorsa indicata
- IPC_STAT: richiede informazioni statistiche sulla risorsa indicata
- IPC_SET: richiede al sistema la modifica di un sottoinsieme degli attributi della risorsa (es. permessi di accesso)

Shared Memory Shm

Creazione della SHM
Collegamento alla SHM
Uso della SHM
Scollegamento della SHM
Eliminazione della SHM

1) `int shmget(key_t key, int size, int flag)`

- **key**: chiave per identificare la SHM in maniera univoca nel sistema
- **size**: dimensione in byte della memoria condivisa
- **flag**: intero che specifica la modalità di creazione e dei permessi di accesso (IPC_CREAT, IPC_EXCL, permessi)

Restituisce un identificatore numerico per la memoria in caso di successo (descrittore), oppure -1 in caso di fallimento

2) `void* shmat(int shmid, const void *shmaddr, int flag)`

Collega il segmento di memoria allo spazio di indirizzamento del chiamante.

- **shmid**: identificatore del segmento di memoria
- **shmaddr**: indirizzo dell'area di memoria del processo chiamante al quale collegare il segmento di memoria condivisa. Se 0, un valore opportuno viene automaticamente scelto.
- **flag**: opzioni (IPC_RDONLY per collegare in sola lettura)

Restituisce l'indirizzo del segmento collegato, oppure -1 in caso di fallimento.

3) `int shmctl(int ds_shm, int cmd, struct shmid_ds * buff)`

Invoca l'esecuzione di un comando su una SHM

ds_shm: descrittore della memoria condivisa su cui si vuole operare
cmd: specifica del comando da eseguire:

- **IPC_STAT**
- **IPC_SET**
- **IPC_RMID**: marca da eliminare, rimuove solo quando non vi sono più processi attaccati
- **SHM_LOCK**: impedisce che il segmento venga swappato o paginato

---

Semafori

1) `int semget(key_t key, int nsems, int semflg);`

2) `int semctl(int semid, int semnum, int cmd );`

```
key_t chiave_sem=IPC_PRIVATE;
//richiesta di 2  semafori ed
  inizializzazione
sem=semget(chiave_sem,2,IPC_CREAT|0664);
//Inizializzazione dei due semafori
semctl(sem,0,SETVAL,val1);
semctl(sem,1,SETVAL,val2);
```

- La semget definisce un ARRAY di più semafori (2 nell'esempio precedente)

- Ogni semaforo dell'array è una struttura dati che comprende tra i suoi campi i seguenti:

```
unsigned short semval;   /* valore semaforo*/
unsigned short semzcnt;  /* # proc che aspettano 0 */
unsigned short semncnt;  /* # proc che aspettano incr.*/
pid_t sempid;            /* proc dell'ultima op. */
```

... e su questi agisce la primitiva **semop**

`int semop(int semid, struct sembuf *sops, unsigned nsops);`

Ognuno degli ***nsops*** elementi, riferiti dal puntatore ***sops***, specifica un'operazione da compiere sul semaforo. L'operazione è descritta da una struttura, ***struct sembuf***, la quale include i seguenti campi:

```
struct sembuf {
  unsigned short sem_num;  /* numero di semaforo */
  short sem_op;            /* operazione da compiere */
  short sem_flg;           /* flags */
}
```

Due sono i valori che puo' assumere ***sem_flg*** : **IPC_NOWAIT** e **SEM_UNDO**. Se si specifica **SEM_UNDO**, l'operazione sarà annullata nel momento in cui il processo che la ha eseguita termina.

3) `semctl(id_sem, num_sem ,IPC_RMID);`

```
27
28      int id_sem = semget(k_sem, 1, IPC_CREAT|0664); // QUANTI SEMAFORI HO?
29
30      if(id_sem < 0) {
31              perror("Errore SEMGET");
32              exit(1);
33      }
34
35
36      int * ptr = shmat(id_shmem, 0, 0);
37
38      if(ptr == (void *)-1) {
39              perror("Errore SHMAT");
40              exit(1);
41      }
42
43
44      semctl(id_sem,0,SETVAL,1);
45      printf("SEMAFORO %d INIZIALIZZATO AD 1\n", id_sem);
46
47
48      *ptr = 0;
49      printf("VALORE INIZIALE: %d\n", *ptr);
50
51
52      int i;
53      for(i=0; i<2; i++) {    // CREA 2 PROCESSI FIGLI
54
55              pid_t pid = fork();
56
57              if(pid < 0) {
58                      perror("Errore FORK");
59                      exit(1);
60              }
61
62              if(pid==0) {
63                      // PROCESSO FIGLIO
64                      printf("Process[%d] PID=%d creato\n", i, getpid());
65                      int j;
66                      for(j=0; j<100; j++) {
67
68                              Wait_Sem(id_sem, 0);
69
70                              //INIZIO SEZIONE CRITICA
71
72                              int tmp = *ptr;
73                              printf("Processo %d ha letto\n", i);
74
75                              // CORRETTO: *ptr = tmp + 1;
76                              // SBAGLIATO: *ptr++;
77                              // SBAGLIATO: *(ptr)++;
78                              // CORRETTO: (*ptr)++;
79
80                              *ptr = tmp + 1;
81
82                              printf("Processo %d ha incrementato\n", i);
83
84                              //FINE SEZIONE CRITICA
85
86                              Signal_Sem(id_sem, 0);
87
88                      }
89
90                      exit(0);
91              }
92      }
93
```

```
void Wait_Sem (int id_sem, int numsem)    {
      struct sembuf sem_buf;
      sem_buf.sem_num=numsem;
      sem_buf.sem_flg=0;
      sem_buf.sem_op=-1;  // 1 PER Signal_Sem
      semop(id_sem,&sem_buf,1);   //semaforo rosso
}
```

Produttori e Consumatori

Pur esistendo un problema (potenziale) di mutua esclusione nell'utilizzo del buffer comune,

la soluzione impone un **ordinamento** nelle operazioni dei due processi.

E' necessario che produttori e consumatori si scambino segnali per indicare rispettivamente l'avvenuto deposito e prelievo.

Ciascuno dei due processi deve attendere, per completare la sua azione, l'arrivo del segnale dell'altro processo.

DEVONO SCAMBIARSI DELLE INFORMAZIONI, TRAMITE IL CANALE MEMORY SHARED PER INVIARE SEGNALI, SI UTILIZZA IL SEMAFORO

Due categorie di processi:
- **Produttori**, che depositano un messaggio su di una risorsa condivisa
- **Consumatori**, che prelevano il messaggio dalla risorsa condivisa

Vincoli:
- Il produttore **non può produrre** un messaggio prima che qualche consumatore abbia **prelevato** il messaggio precedente
- Il consumatore **non può prelevare** alcun messaggio fino a che un produttore non l'abbia **depositato**

Per la sincronizzazione dei processoproduttore e consumatore si utilizzano **due semafori**:

1. **SPAZIO_DISP**: semaforo bloccato da un produttore prima di una produzione, e sbloccato da un consumatore in seguito ad un consumo (**VALORE INIZIALE: 1**)
2. **MSG_DISP**: semaforo sbloccato da un produttore in seguito ad una produzione, e bloccato da un consumatore prima del consumo (**VALORE INIZIALE: 0**)

```
void Consumatore(msg * ptr_sh, int sem){

        msg mess;

        Wait_Sem(sem, MSG_DISP);

        // Prelievo del messaggio
        mess = *ptr_sh;
        printf("Messaggio letto: <%d> \n", mess);

        Signal_Sem(sem, SPAZIO_DISP);

}
```

```
 void Produttore(msg * ptr_sh, int sem){

         msg mess;

         Wait_Sem(sem, SPAZIO_DISP);

         // Produzione di valore_prodotto ...
         printf ("Produzione in corso…");
         mess = valore_prodotto;

         *ptr_sh = mess;

         Signal_Sem(sem, MSG_DISP);
 }
```

La coda è implementata mediante i seguenti campi:

- **buffer[DIM]** – array di elementi di tipo msg (tipo del messaggio depositato dai produttori) contenente i valori prodotti;
- **testa** - tipo intero. Si riferisce alla posizione del primo elemento libero in testa; in altre parole rappresenta il primo elemento disponibile per la memorizzazione del messaggio prodotto. L'elemento di testa è **buffer[testa-1]**;
- **coda** – tipo intero. L'elemento puntato si riferisce alla posizione dell'elemento di coda della coda. In altre parole l'elemento di coda è **buffer[coda]**
- Anche le variabili **testa** e **coda** devono essere condivise!!

Nell'ipotesi in cui vi siano **più produttori** e **più consumatori** che accedono allo stesso buffer, le operazioni di deposito e prelievo devono essere eseguite rispettivamente in **mutua esclusione**, ed essere quindi programmate come sezioni critiche.

A tal fine, bisogna introdurre due nuovi semafori:
- **MUTEX_C** per le operazioni di consumo
- **MUTEX_P** per le operazioni di produzione
- Entrambi inizializzati a 1

```
void Produttore(int* testa, msg* buffer, int sem) {

        Wait_Sem(sem, SPAZIO_DISP);
        Wait_Sem(sem, MUTEX_P);     // inizio sez. critica

        // Produzione di valore_prodotto . . .
        buffer[testa] = valore_prodotto;
        testa = (++testa) % DIM;     // gestione circolare

        Signal_Sem(sem, MUTEX_P);  // fine sez. critica
        Signal_Sem(sem, NUM_MESS);       // nelem=nelem+1
}
```

```
void Consumatore(int* coda, msg* buffer, int sem){
        msg mess;

        Wait_Sem(sem, NUM_MESS);
        Wait_Sem(sem, MUTEX_C);     // inizio sez. critica

        // Consumo
        mess = buffer[coda];
        coda = (++coda) % DIM;      // gestione circolare
        printf("Messaggio letto: <%d> \n",mess);

        Signal_Sem(sem, MUTEX_C);  // fine sez. critica
        Signal_Sem(sem, SPAZIO_DISP);  // nelem=nelem-1
}
```

POOL DI BUFFER CON VETTORE DI STATO

La gestione del pool di buffer avviene mediante due vettori:

- **buffer[DIM]** – array di elementi di tipo msg (tipo del messaggio depositato dai produttori) contenente i valori prodotti;
- **stato[DIM]** - array di elementi di tipo intero. Il valore i-simo, stato[i], può assumere i seguenti tre valori:
  - VUOTO – la cella buffer[i] non contiene alcun valore prodotto;
  - PIENO – la cella buffer[i] contiene un valore prodotto e non ancora consumato;
  - IN_USO – il valore della cella buffer[i] contiene un valore in uso da un processo attivo, consumatore o produttore.

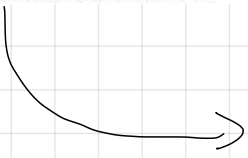
Per la gestione della mutua esclusione nell' accesso al buffer di stato, si possono utilizzare due mutex, **MUTEXP** e **MUTEXC**, uno per i produttori e uno per i consumatori, entrambi inizializzati a 1.

I soliti due semafori, **SPAZIO_DISP** e **MSG_DISP**, usati per la sincronizzazione delle operazioni di produzione e consumo
- SPAZIO_DISP inizializzato a DIM (o al numero di produttori, se minore o uguale a DIM)
- MESS_DISP inizializzato a 0

Con questa soluzione, i produttori (o i consumatori) accedono alla sezione critica gestita da **MUTEXP** (o **MUTEXC**) solo per acquisire una cella libera tramite il vettore di stato

Una volta acquisita la cella, possono procedere concorrentemente nella produzione (o consumo)

I produttori "veloci" potranno terminare prima e segnalare prima i consumatori (e viceversa)



```
void Produttore(int* stato, msg* buffer, int sem) {
        int indice = 0;
        Wait_Sem(sem, SPAZIO_DISP); // attende spazio

        Wait_Sem(sem, MUTEXP); // sez. critica
        // determina l'indice del primo elemento VUOTO
        while (indice<=DIM && stato[indice]!=VUOTO)
                indice++;
        stato[indice]=IN_USO;
        Signal_Sem(sem, MUTEXP); // sez. critica

        // Produzione di valore_prodotto . . .
        buffer[indice]=valore_prodotto;
        stato[indice]=PIENO;

        Signal_Sem(sem, MSG_DISP); // segnala i cons
```

```
void Consumatore(int* stato, msg* buffer, int sem) {
        int indice = 0;
        Wait_Sem(sem, MSG_DISP); // attende messaggio

        Wait_Sem(sem, MUTEXC); // sez. critica
        // determina l'indice del primo elemento PIENO
        while (indice<=DIM && stato[indice]!=PIENO)
                indice++;
        stato[indice]=IN_USO;
        Signal_Sem(sem, MUTEXC); // sez. critica

        // consumo del messaggio
        msg valore_consumato = buffer[indice];
        stato[indice]=VUOTO;

        Signal_Sem(sem, SPAZIO_DISP); // segnala i prod
```

Una sola coppia
produttore consumatore

2 mutex,
SPAZIO_DISPONIBILE =1
MESS_DISPONIBILE =0

```c
void produttore(int * p, int ds_sem) {

        printf("produttore è fermo prima di wait\n");
        Wait_Sem(ds_sem, SPAZIO_DISPONIBILE);
        printf("produttore si sblocca dopo la wait\n");


        sleep(2);

        *p = rand() % 100;

        printf("Il valore prodotto = %d\n", *p);


        Signal_Sem(ds_sem, MESSAGGIO_DISPONIBILE);
}

void consumatore(int * p, int ds_sem) {

        printf("consumatore è fermo prima di wait\n");
        Wait_Sem(ds_sem, MESSAGGIO_DISPONIBILE);
        printf("consumatore si sblocca dopo la wait\n");


        sleep(2);
        printf("Il valore consumato = %d\n", *p);


        Signal_Sem(ds_sem, SPAZIO_DISPONIBILE);
}
```

Pull di buffer con testa e coda con mutua esclusione con produttori e consumatori con stessa dimensione (velocità)

SPAZIO_DISP
MESSAGGIO DISP
MUTEXC
METEXP

```c
void produttore(struct prodcons * p, int ds_sem) {

    //printf("produttore è fermo prima di wait\n");
    Wait_Sem(ds_sem, SPAZIO_DISPONIBILE);
    //printf("produttore si sblocca dopo la wait\n");


    Wait_Sem(ds_sem, MUTEX_P);


    sleep(2);

    // genera valore tra 0 e 99
    p->buffer[p->testa] = rand() % 100;

    printf("Il valore prodotto = %d\n", p->buffer[p->testa]);

    p->testa = (p->testa+1) % DIM_BUFFER;


    Signal_Sem(ds_sem, MUTEX_P);

    Signal_Sem(ds_sem, MESSAGGIO_DISPONIBILE);
}

void consumatore(struct prodcons * p, int ds_sem) {

    //printf("consumatore è fermo prima di wait\n");
    Wait_Sem(ds_sem, MESSAGGIO_DISPONIBILE);
    //printf("consumatore si sblocca dopo la wait\n");

    Wait_Sem(ds_sem, MUTEX_C);


    sleep(2);

    printf("Il valore consumato = %d\n", p->buffer[p->coda]);

    p->coda = (p->coda + 1) % DIM_BUFFER;


    Signal_Sem(ds_sem, MUTEX_C);

    Signal_Sem(ds_sem, SPAZIO_DISPONIBILE);
}
```

Produttori e Consumatori con Pull di Buffer
con vettore di stato, produttori e consumatori con velocità differenti tutelati

SPAZIO_DISP
MESSAGGIO DISP
MUTEXC
METEXP

```c
void produttore(struct prodcons * p, int ds_sem) {

    int indice = 0;


    Wait_Sem(ds_sem, SPAZIO_DISPONIBILE);


    Wait_Sem(ds_sem, MUTEX_P);

    while(indice <= DIM_BUFFER && p->stato[indice] != BUFFER_VUOTO) {
        indice++;
    }

    p->stato[indice] = BUFFER_INUSO;

    //qui devo rilasciare il mutex per i produttori...ERRORE se non lo faccio!!!!
    Signal_Sem(ds_sem, MUTEX_P);


    sleep(2);

    // genera valore tra 0 e 99
    p->buffer[indice] = rand() % 100;

    printf("Il valore prodotto = %d\n", p->buffer[indice]);


    p->stato[indice] = BUFFER_PIENO;

    Signal_Sem(ds_sem, MESSAGGIO_DISPONIBILE);
}
void consumatore(struct prodcons * p, int ds_sem) {

    int indice = 0;


    Wait_Sem(ds_sem, MESSAGGIO_DISPONIBILE);


    Wait_Sem(ds_sem, MUTEX_C);

    while(indice <= DIM_BUFFER && p->stato[indice] != BUFFER_PIENO) {
        indice++;
    }

    p->stato[indice] = BUFFER_INUSO;

    //qui devo rilasciare il mutex per i consumatori...ERRORE se non lo faccio!!!!
    Signal_Sem(ds_sem, MUTEX_C);


    sleep(2);

    printf("Il valore consumato = %d\n", p->buffer[indice]);


    p->stato[indice] = BUFFER_VUOTO;

    Signal_Sem(ds_sem, SPAZIO_DISPONIBILE);
}
```

Lettori e Scrittori

Due categorie di processi:
- **Lettori**, che leggono un messaggio su di una risorsa condivisa
- **Scrittori**, che scrivono il messaggio dalla risorsa condivisa

Vincoli:
1. i processi **lettori** possono accedere **contemporaneamente** alla risorsa
2. i processi **scrittori** hanno accesso **esclusivo** alla risorsa
3. i **lettori** e **scrittori** si **escludono mutuamente** dall'uso della risorsa

Le operazioni di lettura sono "protette" dalle procedure di **Inizio_Lettura()** e **Fine_Lettura()**,

Le operazioni di scrittura sono " protette " da **Inizio_Scrittura()** e **Fine_Scrittura()**.

Un processo **lettore attende** solo se la risorsa è **occupata** da un processo scrittore e un processo **scrittore** può accedere alla risorsa solo se questa è **libera**.

Questa particolare strategia di sincronizzazione può tuttavia provocare condizioni di **attesa indefinita (STARVATION)** per i processi scrittori.

Starvation degli scrittori

Una variable condivisa **NUM_LETTORI** viene usata per contare il numero di lettori che contemporaneamente accedono alla risorsa

Solo quando NUM_LETTORI = 0, gli scrittori possono accedere (uno alla volta) alla risorsa

Si utilizzano **2 semafori**:

- MUTEXL per gestire l'accesso alla variabile NUM_LETTORI in mutua esclusione, da parte dei lettori (inizializzato a 1)
- SYNCH per garantire la mutua esclusione tra i processi lettori e scrittori e tra i soli processi scrittori (inizializzato a 1)

```
void Inizio_Lettura(int sem) {
      Wait_Sem(sem, MUTEXL);
      Num_Lettori++;
      if (Num_Lettori==1) Wait_Sem (sem, SYNCH);
      Signal_Sem(sem, MUTEXL);
}


void Fine_Lettura(int sem) {
      Wait_Sem(sem, MUTEXL);
      Num_Lettori--;
      if (Num_Lettori==0) Signal_Sem (sem, SYNCH);
      Signal_Sem(sem, MUTEXL);
}
```

```
void Inizio_Scrittura(int sem) {
      Wait_Sem(sem, SYNCH);
}

void Fine_Scrittura(int sem) {
      Signal_Sem(sem, SYNCH);
}
```

Starvation degli scrittori: mentre uno scrittore è sospeso sul semaforo SYNCH, altri lettori possono accedere alla risorsa, ritardando indefinitamente lo scrittore

Starvation di entrambi

Anche per gli scrittori si può ottenere un comportamento analogo, introducendo una variabile **NUM_SCRITTORI**

In questo caso occorrono 4 semafori, tutti inizializzati a 1:

- MUTEXL per gestire l'accesso alla variabile NUM_LETTORI in mutua esclusione, da parte dei lettori
- MUTEXS per gestire l'accesso alla variabile NUM_SCRITTORI in mutua esclusione, da parte degli scrittori
- MUTEX per gestire l'accesso in mutua esclusione alla risorsa condivisa da parte degli scrittori
- SYNCH per garantire la mutua esclusione tra i processi lettori e scrittori

```
void Inizio_Lettura(int sem) {
      Wait_Sem(sem, MUTEXL);
      Num_Lettori++;
      if (Num_Lettori==1) Wait_Sem (sem, SYNCH);
      Signal_Sem(sem, MUTEXL);
}


void Fine_Lettura(int sem) {
      Wait_Sem(sem, MUTEXL);
      Num_Lettori--;
      if (Num_Lettori==0) Signal_Sem (sem, SYNCH);
      Signal_Sem(sem, MUTEXL);
}
```

```
void Inizio_Scrittura(int sem) {
      Wait_Sem(sem, MUTEXS);
      Num_Scrittori++;
      if (Num_Scrittori==1) Wait_Sem(sem, SYNCH);
      Signal_Sem(sem, MUTEXS);
      Wait_Sem(sem, MUTEX);
}

void Fine_Scrittura(int sem) {
      Signal_Sem(sem, MUTEX);
      Wait_Sem(sem, MUTEXS);
      Num_Scrittori--;
      if (Num_Scrittori==0) Signal_Sem (sem, SYNCH);
      Signal_Sem(sem, MUTEXS);
}
```

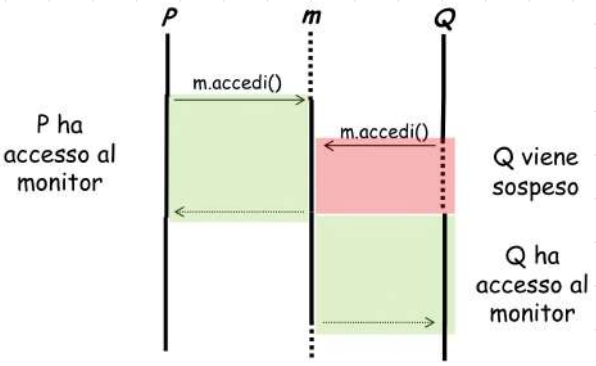
```
17 void InizioLettura(int sem, Buffer* buf){
18
19         Wait_Sem(sem, MUTEXL); //Indica ai lettori che sto iniziando a leggere, incremento
20                                // numlettori in mutua esclusione
21         buf->numlettori = buf->numlettori + 1;
22
23         if (buf->numlettori == 1) //se si tratta del primo lettore blocca gli scrittori
24             Wait_Sem(sem, SYNCH);
25
26         Signal_Sem(sem, MUTEXL); //Rilascia il mutex per far entrare altri lettori
27 }
28
29 void FineLettura(int sem, Buffer* buf){
30
31         Wait_Sem(sem, MUTEXL); //Indica ai lettori che sto terminando la lettura, decremento
32                                // numlettori in mutua esclusione
33         buf->numlettori = buf->numlettori - 1;
34
35         if (buf->numlettori == 0) //se sono l'ultimo lettore devo rilasciare la risorsa per gli scrittori
36                 Signal_Sem(sem, SYNCH);
37
38         Signal_Sem(sem, MUTEXL); //rilascio il mutex per altri lettori che vogliono iniziare la lettura
39 }
40
41 //Procedure di inizio e fine scrittura
42
43 void InizioScrittura(int sem, Buffer* buf){
44
45         Wait_Sem(sem,MUTEXS); //Indica agli scrittori che sto iniziando a scrivere, incremento
46                               // numscrittori in mutua esclusione
47         buf->numscrittori = buf->numscrittori + 1;
48
49             if (buf->numscrittori == 1) // se si tratta del primo scrittore blocca i lettori
50             Wait_Sem(sem, SYNCH);
51
52         Signal_Sem(sem,MUTEXS); //Rilascia il mutex per far entrare altri scrittori per potersi mettere
   in attesa
53         Wait_Sem(sem,MUTEX); //Blocco eventuali scrittori per la scrittura vera e propria
54 }
55
56 void FineScrittura(int sem, Buffer* buf){
57
58         Signal_Sem(sem,MUTEX); //Rilascio il mutex per gli scrittori che devono scrivere
59         Wait_Sem(sem,MUTEXS); //Indica agli scrittori che sto terminando la scrittura, decremento
60                               // numscrittori in mutua esclusione
61         buf->numscrittori = buf->numscrittori - 1;
62
63         if (buf->numscrittori == 0) //se sono l'ultimo scrittore devo rilasciare la risorsa per i lettori
64                 Signal_Sem(sem, SYNCH);
65
66         Signal_Sem(sem,MUTEXS); //rilascio il mutex per altri scrittori che vogliono iniziare la scrittura
```

Monitor

Il monitor è un **costrutto sintattico** che associa un insieme di operazioni a una struttura dati (risorsa) condivisa tra più processi

E' stato introdotto per **facilitare la programmazione strutturata** di problemi in cui è necessario controllare l'**assegnazione** di una o più risorse tra più processi concorrenti mediante apposite **discipline** di gestione.

Il monitor viene utilizzato per definire tipi di **risorse condivise**



Lo scopo è quello di controllare l'assegnazione di una risorsa tra processi concorrenti in accordo a determinate politiche di gestione, secondo due livelli di controllo:

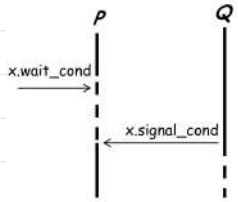
1. **un solo processo alla volta abbia accesso alle variabili locali (competizione);**
2. **una disciplina di scheduling che stabilisca l'ordine con il quale i processi hanno accesso alla risorsa (cooperazione).**

Al fine di garantire che un solo processo alla volta abbia accesso alle variabili locali, si impone che

- le **funzioni di accesso** al monitor vengano eseguite in modo **mutuamente esclusivo**;
- i processi che invocano una funzione di un oggetto monitor mentre un altro processo è attivo nell'istanza, devono essere **ritardati**

Signal and Wait

Signal_and_wait prevede che il processo P risvegliato riprenda immediatamente l'esecuzione e che Q venga sospeso;



Q viene sospeso per evitare che possa modificare nuovamente la condizione di sincronizzazione.

La semantica *signal_and_wait* privilegia il processo **segnalato** rispetto al **segnalante**.

Il segnalato viene risvegliato viene **subito** risvegliato al momento della *signal_cond*, e il possesso del mutex del monitor è **ceduto al segnalato**.

Al risveglio, il segnalato è **certo di trovare vera** la condizione per la quale è stato risvegliato. Quindi, lo schema tipico dell'invocazione di una *wait_cond* è all'interno di un «if»:

```
if (!B) {                  //B è la condizione di sincronizz.
    cv.wait_cond();        //cv è una var. condition abbinata
}                          //alla condizione B
<...accesso alla risorsa...>
```

Signal and urgent wait (Monitor Hoare)

Per poter continuare, il processo **segnalante** attende che il segnalato esca dal monitor («..._and_wait»), ponendosi **in attesa sul mutex del monitor**. Dovrà quindi competere con eventuali altri **processi sopraggiunti** nel frattempo.

La soluzione di Hoare è un caso particolare di signal_and_wait, detta *signal_and_urgent_wait*

Prevede che il processo Q abbia la **priorità** su ogni altro processo che intende entrare nel monitor

Ciò si può ottenere sospendendo il processo Q su un'apposita coda (*urgent_queue*), separata dal mutex.

```
int produci  enter_monitor( &(pc->m) );

        printf("Ingresso monitor - produzione\n");

        if( pc->buffer_occupato == 1 ) {

                printf("Sospensione - produzione\n");
                wait_condition( &(pc->m), VARCOND_PRODUTTORI );
                printf("Riattivazione - produzione\n");
        }

        pc->buffer = valore;
        pc->buffer_libero = 0;
        pc->buffer_occupato = 1;

        printf("Produzione (%d)\n", valore);

        signal_condition( &(pc->m), VARCOND_CONSUMATORI );

        leave_monitor( &(pc->m) );

        printf("Uscita monitor - produzione\n");
}

int Consuma(struct ProdCons * pc) {

        int valore;

        enter_monitor( &(pc->m) );

        printf("Ingresso monitor - consumazione\n");

        if( pc->buffer_libero == 1 ) {

                printf("Sospensione - consumazione\n");
                wait_condition( &(pc->m), VARCOND_CONSUMATORI );
                printf("Riattivazione - consumazione\n");
        }

        valore = pc->buffer;
        pc->buffer_libero = 1;
        pc->buffer_occupato = 0;

        printf("Consumazione (%d)\n", valore);

        signal_condition( &(pc->m), VARCOND_PRODUTTORI );

        leave_monitor( &(pc->m) );

        printf("Uscita monitor - consumazione\n");

        return valore;
```

Produci e consuma nel modello signal and wait.

```c
/*******************IMPLEMENTAZIONE DELLE PROCEDURE**********************/

void init_monitor (Monitor *M,int num_var){

    int i;

    //alloca e inizializza il mutex per l'accesso al monitor
    M->mutex=semget(IPC_PRIVATE,1,IPC_CREAT|0664);

    semctl(M->mutex,0,SETVAL,1);


    //alloca e inizializza il semaforo per la coda urgent
    M->urgent_sem=semget(IPC_PRIVATE,1,IPC_CREAT|0664);

    semctl(M->urgent_sem,0,SETVAL,0);


    //alloca e inizializza i semafori con cui realizzare le var.condition
    M->id_conds=semget(IPC_PRIVATE,num_var,IPC_CREAT|0664);

    for (i=0;i<num_var;i++)
         semctl(M->id_conds,i,SETVAL,0);


    //alloca un contatore per ogni var.condition, più un contatore per la coda urgent
    M->id_shared=shmget(IPC_PRIVATE,(num_var+1)*sizeof(int),IPC_CREAT|0664);

    printf("(num_var+1)*sizeof(int) = %d\n", (num_var+1)*sizeof(int));

    //effettua l'attach all'array di contatori appena allocato
    M->cond_counts=(int*) (shmat(M->id_shared,0,0));

    printf("M->cond_counts %p\n", M->cond_counts);

    M->num_var_cond = num_var;

    M->urgent_count = M->cond_counts + M->num_var_cond;

    printf("M->urgent_count %p\n", M->urgent_count);

    //inizializza i contatori per le var.condition e per la coda urgent
    for (i=0; i<num_var; i++)
        M->cond_counts[i]=0;

    *(M->urgent_count)=0;
}
void wait_condition(Monitor* M,int id_var){

#ifdef DEBUG_
    if(id_var<0 || id_var>=M->num_var_cond) {
        printf("<%d> -Monitor- errore nell'invocazione della wait (idvar=%d)\n", getpid(), id_var);
    }
#endif

#ifdef DEBUG_
    printf("<%d> -Monitor- invocata la wait sulla condition numero %d\n", getpid(), id_var);
#endif

      M->cond_counts[id_var]=M->cond_counts[id_var]+1;

      if( *(M->urgent_count) > 0 ) {
#ifdef DEBUG_
         printf("<%d> -Monitor- signal sulla coda urgent \n", getpid());
#endif
         Signal_Sem(M->urgent_sem,0);
      } else {
#ifdef DEBUG_
         printf("<%d> -Monitor- signal sul mutex del monitor \n", getpid());
#endif
         Signal_Sem(M->mutex,0);
      }

      Wait_Sem(M->id_conds,id_var);

      M->cond_counts[id_var]=M->cond_counts[id_var]-1;
}
void signal_condition(Monitor* M,int id_var){

#ifdef DEBUG_
    if(id_var<0 || id_var>=M->num_var_cond) {
        printf("<%d> -Monitor- errore nell'invocazione della signal (idvar=%d)\n", getpid(), id_var);
    }
#endif

#ifdef DEBUG_
    printf("<%d> -Monitor- tentativo di signal; n.ro proc. in attesa sulla cond. n. %d = %d\n",
getpid(), id_var,M->cond_counts[id_var]);
#endif

    (*(M->urgent_count))++;

    if(M->cond_counts[id_var]>0) {

            Signal_Sem(M->id_conds,id_var);

#ifdef DEBUG_
            printf("<%d> -Monitor- invocata la signal sulla condition numero %d\n", getpid(), id_var);
#endif

#ifdef DEBUG_
            printf("<%d> -Monitor- processo in attesa sulla coda urgent \n", getpid());
#endif

            Wait_Sem(M->urgent_sem,0);

#ifdef DEBUG_
            printf("<%d> -Monitor- processo uscito dalla coda urgent \n", getpid());
#endif
```

Sign and Wait

Sono già nel monitor a questo punto

```
wait_cond(){
    condcount++;
    signal(mutex);
    wait(condsem);
}

signal_cond(){
    if (condcount>0){
        condcount--;
        signal(condsem);
        wait(mutex);
    }
}
```

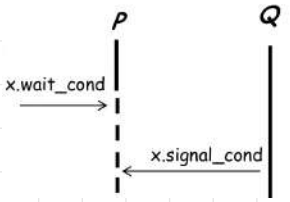
Sign and urg Wait

```
wait_cond(){
    condcount++;
    if(urgentcount>0)
        signal(urgent);
    else
        signal(mutex);
    wait(condsem);
    condcount--;
}

signal_cond(){
    urgentcount++;
    if(condcount>0) {
        signal(condsem);
        wait(urgent);
    }
    urgentcount--;
}
```

Signal and Continue

Signal_and_continue (detto anche *wait and notify*) privilegia il processo **segnalante** rispetto al **segnalato**. Il processo Q segnalante prosegue la sua esecuzione, mantenendo l'accesso esclusivo al monitor, dopo aver risvegliato P



Q prosegue l'esecuzione dopo aver risvegliato P

Il processo P segnalato **non viene attivato subito** al momento della *signal_cond*. Invece, viene trasferito alla coda associata all'**ingresso del monitor** (compete per il mutex del monitor).

Nel frattempo che P possa acquisire di nuovo il mutex, può verificarsi che il processo Q, oppure altri processi sopraggiunti nel frattempo (e che precedono l'esecuzione di P) modifichino le **variabili locali** del monitor e la **validità della condizione di sincronizzazione**

Quando P avrà acquisito il monitor, esso dovrà **verificare nuovamente** la condizione di sincronizzazione prima di proseguire. Quindi, con una politica signal_and_continue lo schema tipico di uso della primitiva wait_cond è all' interno di un «while»:

- **while (!B) { cv.wait_cond() }**
  <...accesso alla risorsa...>

```c
void init_monitor (Monitor *M,int num_var){

    int i;

    //alloca e inizializza il mutex per l'accesso al monitor
    M->mutex=semget(IPC_PRIVATE,1,IPC_CREAT|0664);

    semctl(M->mutex,0,SETVAL,1);


    //alloca e inizializza i semafori con cui realizzare le var.condition
    M->id_conds=semget(IPC_PRIVATE,num_var,IPC_CREAT|0664);

    for (i=0;i<num_var;i++)
         semctl(M->id_conds,i,SETVAL,0);


    //alloca un contatore per ogni var.condition
    M->id_shared=shmget(IPC_PRIVATE,num_var*sizeof(int),IPC_CREAT|0664);


    //effettua l'attach all'array di contatori appena allocato
    M->cond_counts=(int*) (shmat(M->id_shared,0,0));

    M->num_var_cond = num_var;



    //inizializza i contatori per le var.condition
    for (i=0; i<num_var; i++)
        M->cond_counts[i]=0;

#ifdef DEBUG_
    printf("Monitor inizializzato con %d condition variables. Buona Fortuna ! \n",num_var);
#endif
}



void enter_monitor(Monitor * M){

#ifdef DEBUG_
    printf("<%d> Tentativo di ingresso nel monitor... \t",getpid() );
#endif

    Wait_Sem(M->mutex,0);

#ifdef DEBUG_
    printf("<%d> Entrato nel monitor \n",getpid() );
#endif
}


void leave_monitor(Monitor* M){

#ifdef DEBUG_
    printf("<%d> Uscito dal monitor  \n", getpid());
    printf("<%d> -Monitor- signal sul mutex del monitor \n", getpid());
#endif
}
```

```c
#ifdef DEBUG_
    printf(" \n Il Monitor è stato rimosso ! Arrivederci \n", getpid());
#endif
}
void wait_condition(Monitor* M,int id_var){

#ifdef DEBUG_
    if(id_var<0 || id_var>=M->num_var_cond) {
        printf("<%d> -Monitor- errore nell'invocazione della wait (idvar=%d)\n", getpid(), id_var);
    }
#endif

#ifdef DEBUG_
    printf("<%d> -Monitor- invocata la wait sulla condition numero %d\n", getpid(), id_var);
#endif

      M->cond_counts[id_var]=M->cond_counts[id_var]+1;


#ifdef DEBUG_
        printf("<%d> -Monitor- signal sul mutex del monitor \n", getpid());
#endif

      Signal_Sem(M->mutex,0);

#ifdef DEBUG_
        printf("<%d> -Monitor- wait sul semaforo %d del monitor \n", getpid(),id_var);
#endif

      Wait_Sem(M->id_conds,id_var);

#ifdef DEBUG_
        printf("<%d> -Monitor- wait sul mutex del monitor \n", getpid());
#endif

        Wait_Sem(M->mutex,0);
}
void signal_condition(Monitor* M,int id_var){

#ifdef DEBUG_
    if(id_var<0 || id_var>=M->num_var_cond) {
        printf("<%d> -Monitor- errore nell'invocazione della signal (idvar=%d)\n", getpid(), id_var);
    }
#endif

#ifdef DEBUG_
    printf("<%d> -Monitor- tentativo di signal; n.ro proc. in attesa sulla cond. n. %d = %d\n",
  getpid(), id_var,M->cond_counts[id_var]);
#endif


   if(M->cond_counts[id_var] > 0){
        M->cond_counts[id_var]--;
        #ifdef DEBUG_
        printf("<%d> -Monitor- signal sul semaforo %d\n", getpid(), id_var);
        #endif
        Signal_Sem(M->id_conds,id_var);
   }
}


int queue_condition(Monitor * M, int id_var){
        return M->cond_counts[id_var];
```

```c
#ifndef __MONITOR_H
#define __MONITOR_H

typedef struct {

//id del semaforo per realizzare il mutex del monitor
  int mutex;

//numero di variabili condition
  int num_var_cond;

//id del gruppo sem associati alle var.cond
  int id_conds;

//id della memoria condivisa per i contatori delle variabili condition
  int id_shared;

//array delle variabili condition_count
  int *cond_counts;

} Monitor;

//monitor e numero di variabili condition
void init_monitor (Monitor*, int);
void enter_monitor(Monitor*);
void leave_monitor(Monitor*);
void remove_monitor(Monitor*);
void wait_condition(Monitor*,int);
void signal_condition(Monitor*,int);
int queue_condition(Monitor*,int);


#endif
```

```c
wait_cond(){
    condcount++;
    signal(mutex);
    wait(condsem);
    wait(mutex);
}

signal_cond(){
    if(condcount>0) {
         condcount--;
         signal(condsem);
    }
}
signal_all(){
    while(condcount>0) {
         condcount--;
         signal(condsem);
    }
}
```

//MONITOR//
Produttori e Consumatori

```c
#include "procedure.h"

#include <unistd.h>
#include <stdio.h>

void Produci(struct ProdCons * pc, int valore) {

        enter_monitor( &(pc->m) );

        printf("Ingresso monitor - produzione\n");

        while( pc->numero_liberi == 0 ) {

                printf("Sospensione - produzione\n");
                wait_condition( &(pc->m), VARCOND_PRODUTTORI );
                printf("Riattivazione - produzione\n");
        }


        int i = 0;
        while( i<DIM && pc->stato[i] != LIBERO ) {
                i++;
        }


        pc->stato[i] = IN_USO;
        pc->numero_liberi--;

        leave_monitor( &(pc->m) );


        // ...operazione lenta...
        sleep(2);

        pc->buffer[i] = valore;

        printf("Produzione - posizione %d, valore %d\n", i, valore);


        enter_monitor( &(pc->m) );

        pc->stato[i] = OCCUPATO;
        pc->numero_occupati++;

        signal_condition( &(pc->m), VARCOND_CONSUMATORI );

        leave_monitor( &(pc->m) );

        printf("Uscita monitor - produzione\n");
}
```

```c
int Consuma(struct ProdCons * pc) {

        int valore;

        enter_monitor( &(pc->m) );

        printf("Ingresso monitor - consumazione\n");

        while( pc->numero_occupati == 0 ) {

                printf("Sospensione - consumazione\n");
                wait_condition( &(pc->m), VARCOND_CONSUMATORI );
                printf("Riattivazione - consumazione\n");
        }


        int i = 0;
        while( i<DIM && pc->stato[i] != OCCUPATO ) {
                i++;
        }


        pc->stato[i] = IN_USO;
        pc->numero_occupati--;

        leave_monitor( &(pc->m) );


        // ...operazione lenta...
        sleep(2);

        valore = pc->buffer[i];

        printf("Consumazione - posizione %d, valore %d\n", i, valore);


        enter_monitor( &(pc->m) );

        pc->stato[i] = LIBERO;
        pc->numero_liberi++;

        signal_condition( &(pc->m), VARCOND_PRODUTTORI );

        leave_monitor( &(pc->m) );

        printf("Uscita monitor - consumazione\n");

        return valore;
}
```

Implementazione delle funzioni produci e consuma nel modello di monitor signal and continue

//MONITOR//
Lettori e Scrittori con starvation di entrambi

```
int Leggi(struct LettScritt * ls) {

        int valore;

        enter_monitor( &(ls->m) );

        printf("Lettura - ingresso monitor\n");

        while( ls->numero_scrittori > 0 ) {

                printf("Sospensione - lettura\n");
        // Nel caso di signal and continue, sarò segnalato da uno scrittore prima
        // o poi, quindi posso lasciare il monitor senza segnalare nessun lettore
                wait_condition( &(ls->m), VARCOND_LETTORI );
                printf("Riattivazione - lettura\n");
        }

        ls->numero_lettori++;

        printf("Numero lettori ++ : %d\n", ls->numero_lettori);

        leave_monitor( &(ls->m) );



        // ...operazione lenta...
        sleep(2);
        valore = ls->buffer;

        printf("Lettura - valore [%d]\n", valore);



        enter_monitor( &(ls->m) );

        ls->numero_lettori--;

        printf("Numero lettori -- : %d\n", ls->numero_lettori);


        if( ls->numero_lettori == 0 ) {

                printf("Lettura - signal su scrittori\n");

                signal_condition( &(ls->m), VARCOND_SCRITTORI );
        }

        leave_monitor( &(ls->m) );

        printf("Lettura - uscita monitor\n");

        return valore;
```

int scrivi

```
        enter_monitor( &(ls->m) );

        printf("Scrittura - ingresso monitor\n");

        while (ls->numero_lettori > 0 || ls->numero_scrittori > 0) {

                printf("Scrittura - sospensione\n");
                wait_condition( &(ls->m), VARCOND_SCRITTORI );
                printf("Scrittura - riattivazione\n");
        }

        ls->numero_scrittori++;

        printf("Numero scrittori ++ : %d\n", ls->numero_scrittori);

        leave_monitor( &(ls->m) );


        // ...operazione lenta...
        sleep(1);
        ls->buffer = valore;

        printf("Scrittura - valore [%d]\n", valore);



        enter_monitor( &(ls->m) );

        ls->numero_scrittori--;

        printf("Numero scrittori -- : %d\n", ls->numero_scrittori);


        if( queue_condition( &(ls->m), VARCOND_SCRITTORI ) ) {

                printf("Scrittura - signal su scrittori\n");

                signal_condition( &(ls->m), VARCOND_SCRITTORI );

        } else {

                /*
                 *  NOTA: questa soluzione è applicabile solo al caso di
                 *  monitor con semantica signal-and-continue (signal all).
         *  Questo perchè lo scrittore dopo aver fatto la prima signal
         *  CONTINUA la sua esecuzione!
                 */
                printf("Scrittura - signal all su lettori\n");

                signal_all( &(ls->m), VARCOND_LETTORI );
         }


        leave_monitor( &(ls->m) );
```

Code di Messaggi (Modello ad ambiente locale)

- la cooperazione si realizza mediante lo **scambio diretto di messaggi** per mezzo di primitive che il S.O. deve rendere disponibili
- Il naturale supporto fisico al modello sono i sistemi di elaborazione con architettura distribuita.

SI DISTINGUONO PER

tipo di sincronizzazione dei processi comunicanti

indirizzamento, ovvero per la modalità con cui si designano la provenienza e la destinazione

Send sincrona, receive bloccante

- Sia il sender sia il receiver rimarranno in attessa fin quando il messaggio non sarà consegnato
- Denominato anche "**rendezvous**"

Send asincrona, receive bloccante

- Il Sender procede nella sua esecuzione dopo aver inviato il messaggio
- Il Receiver rimane in attesa fin quando non arriva il messaggio

Send asincrona, receive non bloccante

- Nessuna delle due parti rimane in attesa

Send

Si consideri un processo che esegue una send.
Vi sono due possibilità:

- Il processo che sta eseguendo la send rimane in attesa fino a che il messaggio è stato ricevuto (**Send sincrona**);
- Il processo che sta eseguendo la send continua la sua esecuzione senza attendere l'avvenuta consegna del messaggio (**Send asincrona**).

Recive

In maniera analoga, per un processo che esegue la receive:

- Se il messaggio è stato inviato dal sender, il processo lo riceve e continua l'esecuzione.
- Se il messaggio non è ancora arrivato il receiver può:
  - rimanere in attesa fino a che il messaggio non è stato ricevuto (**Receive Bloccante**);
  - continuare la sua esecuzione senza attendere l'avvenuta consegna del messaggio (**Receive Non Bloccante**)

UNIX supporta il modello nel quale lo scambio di messaggi non avviene direttamente tra il mittente e il destinatario, ma avviene tra un utente e una *mailbox (comunicazione indiretta)*

UNIX System V mette a disposizione unicamente le primitive

- Send asincrona
- Receive (bloccante e non bloccante)

Una *mailbox* può essere vista come una coda di messaggi. Essa é caratterizzata da:

- una *chiave* (analoga a quella dei segmenti di memoria condivisa);
- Un *proprietario* (l'utente che la istanzia);
- Un *gruppo* di appartenenza;
- Un insieme di *protezioni* (indicate dalla solita stringa con 3 numeri a 3 bit)

int msgget(key_t k, IPC_CREAT|0664)

int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)

Invia un messaggio sulla coda msqid. Il messaggio é contenuto all'interno del buffer *msgbuf*:

```
struct msgbuf {
      long message_type;
      char message_text [MAX_SIZE];
}
```

Il campo ***message_type*** identifica il ***tipo*** del messaggio: esso diventa molto importante quando accoppiato con **funzioni receive *selettive***

- Permette di estrarre un **particolare messaggio** dalla coda (il cui campo ***tipo*** contenga il valore specificato), anche se non si trova in cima
- Se *message_type* assume il valore del **pid** del mittente possiamo realizzare una **comunicazione indiretta simmetrica**

Il campo ***msgsz*** rappresenta la dimensione (in byte) del messaggio

- la dimensione di msgbuf **meno** la dimensione del campo tipo

Il campo ***msgflg*** é un flag che ci permette di specificare la semantica dell'operazione di send

- Se *msgflg* = 0 la send blocca il processo se la mailbox è piena
- Se *msgflg* = IPC_NOWAIT, e la **mailbox è piena**, la send ritorna -1 e non accoda il messaggio (NOTA: msgsnd() è **sempre asincrona**, tranne quando la mailbox è piena)

La funzione restituisce 0 se non si sono verificati errori

```
result = msgsnd(msqid, (void*)&msg, sizeof(msg)-sizeof(long), 0);
```

int msgrcv(int msqid, void *msgp, int msgsz, long msgtyp, int msgflg);

- Consuma un messaggio dalla mailbox identificata da *msqid*
- Il buffer contenente il messaggio sarà puntato da *msgp*
- Il campo messaggio di tale buffer avrà dimensione *msgsz*

Il campo *msgtyp* (tipo del messaggio) svolge importanti funzioni:

- *Se msgtyp* = 0 viene prelevato il primo messaggio della coda (ovvero quello inviato da più tempo);
- Se *msgtyp* > 0 viene prelevato il primo messaggio dalla coda il cui campo *tipo* sia pari al valore di *msgtyp*
- Se *msgtyp* < *0* viene prelevato il primo messaggio dalla coda il cui campo *tipo* abbia una valore minore o uguale a |*msgtyp*|

Il campo *msgflg*

- se impostato a 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 impostato a IPC_NOWAIT, la ricezione non é bloccante: se non ci sono messaggi viene restituito -1

int msgctl (msqid, IPC_RMID, 0)

Per la lettura della coda (senza consumo) IPC_STAT

Per modificare le caratteristiche della coda IPC_SET

Errori da non fare

Campo "tipo" impostato a zero.

- Il valore 0 è un valore riservato per usi speciali, e non va utilizzato nei messaggi definiti dal programmatore. Ad esempio, nella primitiva msgrcv(), il quarto parametro ("tipo") deve essere 0 quando si vuole che il processo possa ricevere messaggi di qualunque tipo.

Campo "tipo" non impostato.

- In tal caso, il valore del campo tipo è imprevedibile e arbitrario (dipende dal sistema operativo, dall'hw, e dal compilatore).

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>

#include "header.h"


int main() {

        printf("Processo P3 avviato\n");
        struct msg_calc m_r;
        float media_cum[2];
        //int n_intermedio[2];

        key_t queue  = ftok(FTOK_PATH_Q, FTOK_CHAR_Q);

        int id_queue = msgget(queue,IPC_CREAT|0644);

        printf("ID QUEUE: %d\n",id_queue);
        if(id_queue < 0) {
                perror("Msgget fallita");
                exit(1);
        }


        int i;
        for(i = 0; i < 22;i++){
                msgrcv(id_queue,(void *)&m_r,sizeof(struct msg_calc)-sizeof(long),0,0);
                printf("Ricevuto messaggio dal processo <%lu> ,con valore <%f>\n",m_r.processo,m_r.numero);
                if(m_r.processo == P1){
                        //n_intermedio[P1-1]++;
                        media_cum[P1-1]+=m_r.numero / 11;
                }else if(m_r.processo == P2){
                        //n_intermedio[P2-1]++;
                        media_cum[P2-1]+=m_r.numero / 11;
                }else{
                        printf("Processo non riconosciuto\n");
                }
        }

        for(i =0; i < 2;i++){
                printf("<Media %d = %f>\n",i+1,media_cum[i]);
        }
        return 0;
}
```

Pthreads

Gestione dei Thread: creazione, distruzione e join di thread;

Gestione dei Mutex: creazione, distruzione, lock e unlock di variabili di mutua esclusione (mutex) per la gestione di sezioni critiche;

Gestione delle Condition Variables: creazione, distruzione, wait e signal su variabili condition definite dal programmatore.

`pthread_create (thread,attr,start_r,arg)`

- Crea un nuovo thread e lo rende eseguibile.
- **`thread`** (output): di tipo `pthread_t`, è un identificatore del thread creato;
- **`attr`** (input): di tipo `pthread_attr_t`, serve a impostare gli attributi del thread;
- **`start_r`** (input): puntatore (di tipo `void *`) alla funzione C (***starting routine***) che verrà eseguita una volta che il thread è creato;
  - Firma di una starting routine di esempio: **void * foo (void *)**
- **`arg`** (input): argomento (di tipo `void *`) che può essere passato alla funzione C (ne va fatto il casting a `void *`).

`pthread_exit (status)`

- Usata per terminare un thread esplicitamente.
- Se usata nel programma principale (che potrebbe terminare prima di tutti i thread), gli altri thread continueranno ad eseguire.
- E' buona norma usarla in tutti i thread.
- **`status`** (input): indica lo stato di uscita del thread.

pthread_join(threadId, status)

Join è un modo per sincronizzare più thread, il thread chiamante aspetta che un altro thread termini.

I pthreads utilizzano il costrutto monitor signal and continue, e implementano tutte le funzionalità necessarie. Per usare la libreria è necessario includere sia <pthread.h> che la libreria dinamica che si collega nel makefile, durante il linkaggio con -lpthread.

Per ottenere il Tid

#include <sys/syscall.h>
int myid= syscall(SYS_gettid);

```c
int main() {

        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

        srand(time(NULL));

        struct monitor* m[5];//l'array di monitor per gestire 5 navi

        pthread_t gestore_molo[5];//5 gestori molo
        pthread_t viagg[10];//10 viaggiatori

        int i;

    /* TODO: Allocare 5 istanze di monitor, e attivarle con inizializza() */
        for (i = 0; i < 5; i++) {
                m[i] = malloc(sizeof(struct monitor));
                inizializza(m[i]);
        }
        // TODO: assegnare un id ad ogni nave
        for (i = 0; i < 5; i++) {
                m[i]->id_nave = rand() % 200;
        }
    /* TODO: Avviare 5 thread, facendogli eseguire la funzione gestoreMolo(),
     *     e passando ad ognuno una istanza di monitor diversa m[i]
     */
        for (i = 0; i < 5; i++) {
                printf("Creato il thread gestore n %d \n", i);
                pthread_create(&gestore_molo[i], &attr, gestoreMolo,
  (void*)m[i]);
        }

    /* TODO: Avviare 10 thread, facendogli eseguire la funzione Viaggiatori(),
     *      e passando ad ognuno una istanza di monitor diversa, da scegliere
     *      a caso con "rand() % 5"
     */
        for (i = 0; i < 10; i++){
                printf("Creato il thread viaggiatore n %d \n", i);
                pthread_create(&viagg[i], &attr, Viaggiatori, (void*)m[rand()
  %5]);
        }

    /* TODO: Effettuare la join con i thread "gestoreMolo" */
        for (i = 0; i < 5; i++) {

                pthread_join(gestore_molo[i], 0);
                        printf("Terminato il thread gestore n %d \n", i);

        }
    /* TODO: Effettuare la join con i thread "Viaggiatori" */
        for (i = 0; i < 10; i++) {
```

```c
#include "header.h"


void inizializza(struct monitor* m){

        m->molo=0;
        m->id_nave=0;

    /* TODO: Inizializzare le variabili dell'algoritmo, il mutex, e le
  variabili condition */
    m->num_scrittori = 0;
    m->num_lettori = 0;
    m->num_scrittori_wait = 0;
    m->num_lettori_wait = 0;

    pthread_mutex_init(&m->mutex, NULL);
    pthread_cond_init(&m->ok_lett_cv, NULL);
    pthread_cond_init(&m->ok_scritt_cv, NULL);
}
void rimuovi (struct monitor* m){
    /* TODO: Disattivare mutex e variabili condition */
    pthread_mutex_destroy(&m->mutex);
    pthread_cond_destroy (&m->ok_lett_cv);
    pthread_cond_destroy(&m->ok_scritt_cv);

}


//SCRITTURA. AGGIORNAMENTO DELLA POSIZIONE DEL TRENO
void scrivi_molo(struct monitor* m, int molo){

    /* TODO: Implementare qui lo schema dei lettori-scrittori con starvation
  di entrambi.
     * nella parte della SCRITTURA
     */
    pthread_mutex_lock(&m->mutex);

    while (m->num_lettori > 0 || m->num_scrittori > 0) {
        m->num_scrittori_wait++;
        pthread_cond_wait(&m->ok_scritt_cv, &m->mutex);
        m->num_scrittori_wait--;
    }

    m->num_scrittori++;

    pthread_mutex_unlock(&m->mutex);

    m->molo=molo;
    pthread_mutex_lock(&m->mutex);

        m->num_scrittori--;
```

Lettori e scrittori

```
//LETTURA. RESTITUISCE LA POSIZIONE DEL TRENO
int leggi_molo(struct monitor* m){

        int molo;
    /* TODO: Implementare qui lo schema dei lettori-scrittori con starvation
di entrambi.
     * nella parte della LETTURA
     */
    pthread_mutex_lock(&m->mutex);
    while (m->num_scrittori > 0) {
        m->num_lettori_wait++;
        pthread_cond_wait(&m->ok_lett_cv, &m->mutex);
        m->num_lettori_wait--;
    }
        m->num_lettori++;
    pthread_mutex_unlock(&m->mutex);
    molo=m->molo;
        pthread_mutex_lock(&m->mutex);

        m->num_lettori--;

        if(m->num_lettori == 0)
                pthread_cond_signal(&m->ok_scritt_cv);

        pthread_mutex_unlock(&m->mutex);
        return molo;
}
```

```
void scrivi_molo(struct monitor* m, int molo){

    /* TODO: Implementare qui lo schema dei lettori-scrittori con starvation
di entrambi.
     * nella parte della SCRITTURA
     */
    pthread_mutex_lock(&m->mutex);

    while (m->num_lettori > 0 || m->num_scrittori > 0) {
        m->num_scrittori_wait++;
        pthread_cond_wait(&m->ok_scritt_cv, &m->mutex);
        m->num_scrittori_wait--;
    }

    m->num_scrittori++;

    pthread_mutex_unlock(&m->mutex);

    m->molo=molo;
    pthread_mutex_lock(&m->mutex);

        m->num_scrittori--;

        if(m->num_scrittori_wait > 0) {

        pthread_cond_signal(&m->ok_scritt_cv);

        } else {

                pthread_cond_broadcast(&m->ok_lett_cv);
        }

        pthread_mutex_unlock(&m->mutex);
}
```

Produttore e consumatore

```
66 void InizioConsumo(struct ProdCons * pc){
67
68         pthread_mutex_lock(&pc->mutex);
69
70         while (pc->ok_consumo==0)
71                 pthread_cond_wait(&pc->ok_cons_cv, &pc->mutex);
72 }
73
74 void FineConsumo(struct ProdCons * pc){
75
76         pc->ok_produzione = 1;
77         pc->ok_consumo = 0;
78
79         pthread_cond_signal(&pc->ok_prod_cv);
80
81         pthread_mutex_unlock(&pc->mutex);
82 }
83
84
85 void InizioProduzione(struct ProdCons * pc){
86
87         pthread_mutex_lock(&pc->mutex);
88
89         while (pc->ok_produzione==0)
90                 pthread_cond_wait(&pc->ok_prod_cv, &pc->mutex);
91
92 }
93
94 void FineProduzione (struct ProdCons * pc){
95
96         pc->ok_consumo = 1;
97         pc->ok_produzione = 0;
98
99         pthread_cond_signal(&pc->ok_cons_cv);
100
101        pthread_mutex_unlock(&pc->mutex);
102 }
```

Non dimenticare il casting inverso

```
void *Produttore(void* p) {

                struct ProdCons * pc = (struct ProdCons *)p;
```

LIBRERIE UTILI

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/msg.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
```