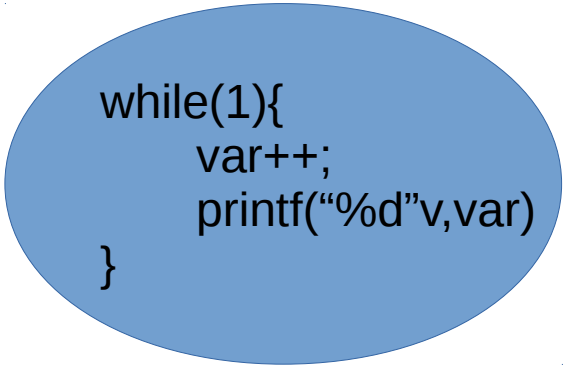
Introduzione alla Sincronizzazione tra processi in Linux con il linguaggio C

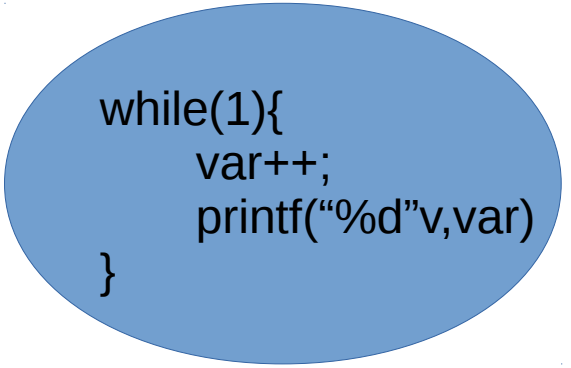
E. Mumolo

Mutua esclusione

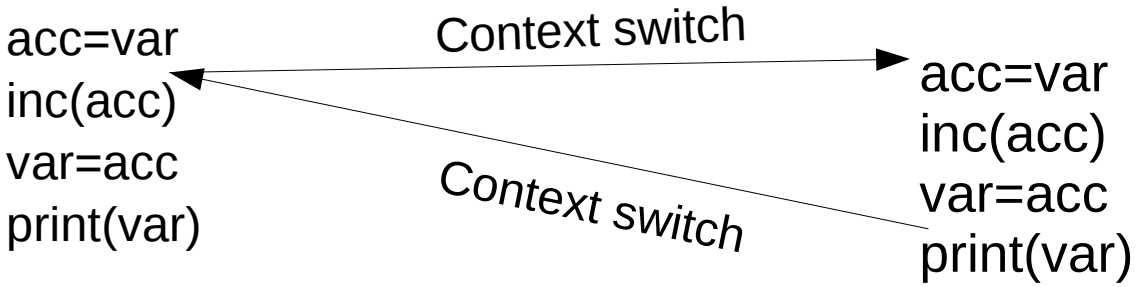
- Il più semplice problema di mutua esclusione: variabile condivisa

Processi concorrenti





- var++ viene tradotta come: (acc=accumulatore, inc=incrementa)

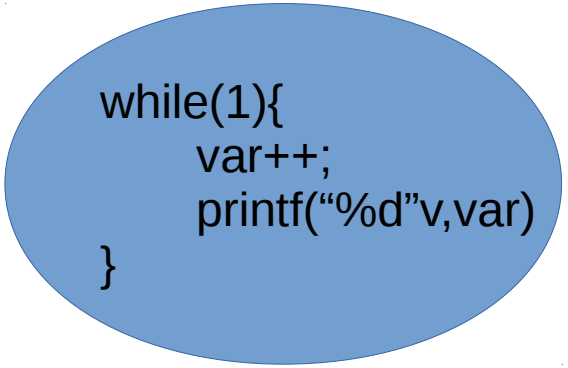


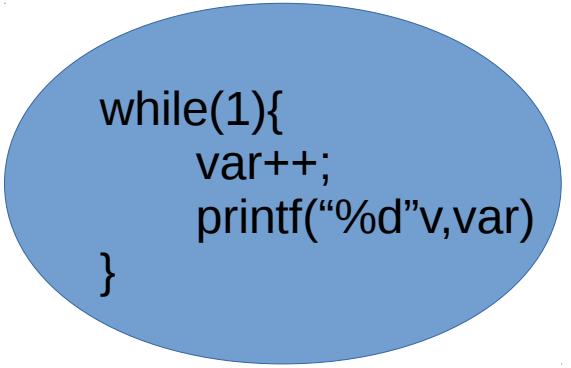
- cosa può succedere?
- Interleaving tra processi

Mutua esclusione

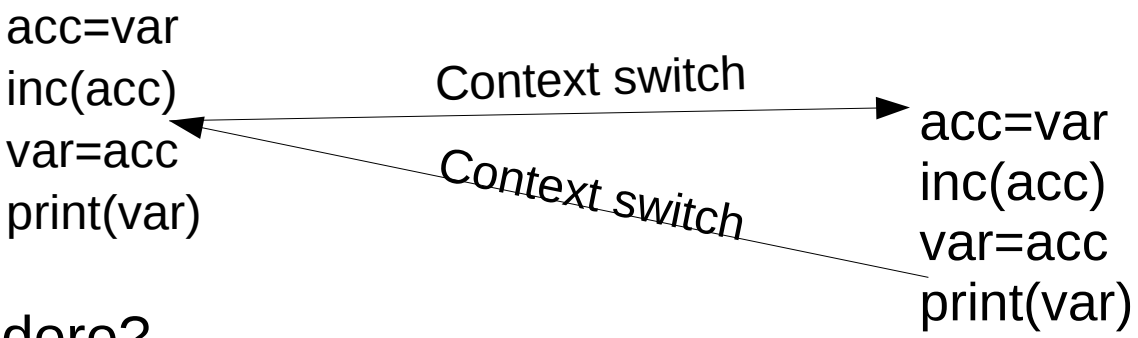
- Il più semplice problema di mutua esclusione: variabile condivisa

Processi concorrenti





- var++ viene tradotta come: (acc=accumulatore, inc=incrementa)

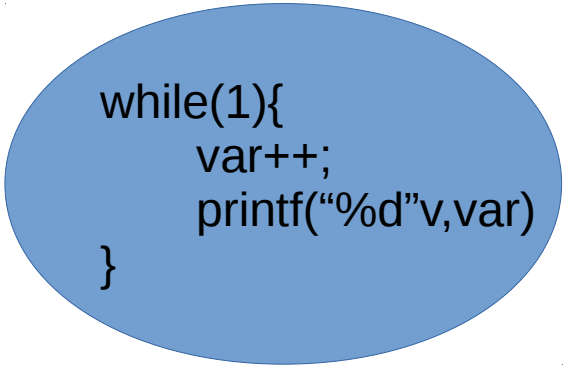


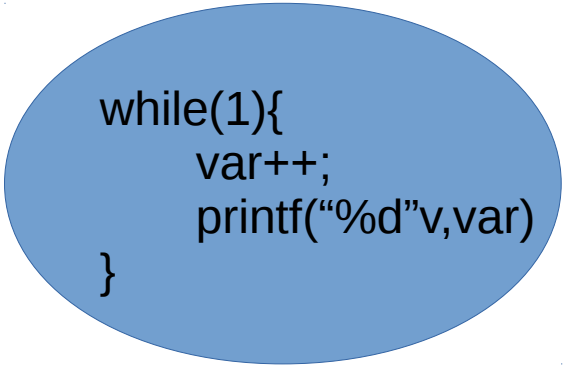
- cosa può succedere?
- Interleaving tra processi

Mutua esclusione

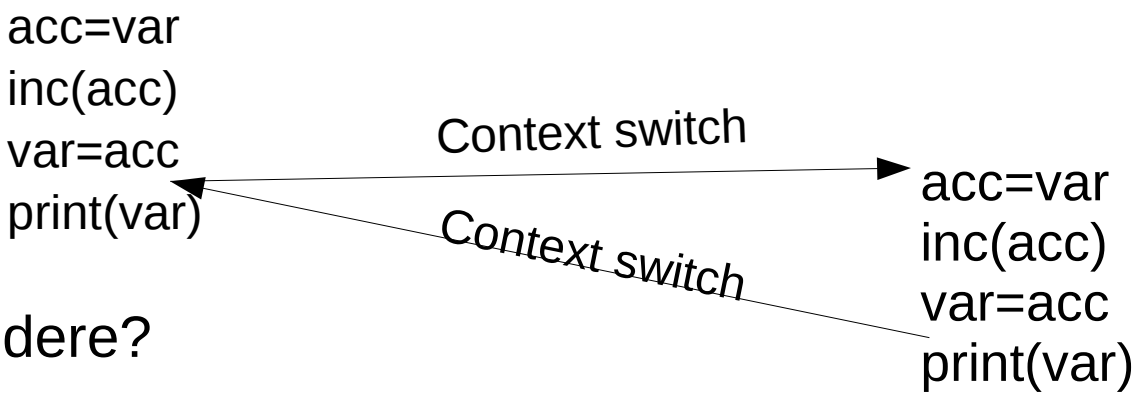
- Il più semplice problema di mutua esclusione: variabile condivisa

Processi concorrenti





- var++ viene tradotta come: (acc=accumulatore, inc=incrementa)



- cosa può succedere?
- Interleaving tra processi

Mutua esclusione

```
#include <semaphore.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#define SIZE 10
//syncro1.c
int main(int argc, char **argv)
{
  int fd, i, val, pid,*ptr;
  sem_t mutex;
  ptr = mmap(0, sizeof(int)*SIZE+sizeof(mutex),
             PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_SHARED, -1, 0);
  *(ptr+10*sizeof(int))=mutex;
  pid=fork();
  if (pid == 0) {
    for (i = 0; i < SIZE; i++) { val=(*ptr)++;printf("figlio: %d\n", val); }
    exit(0);
  }
  else{
     for (i = 0; i < SIZE; i++) {  val=(*ptr)++; printf("padre:  %d\n", val);   }
     exit(0);
   }
}
```

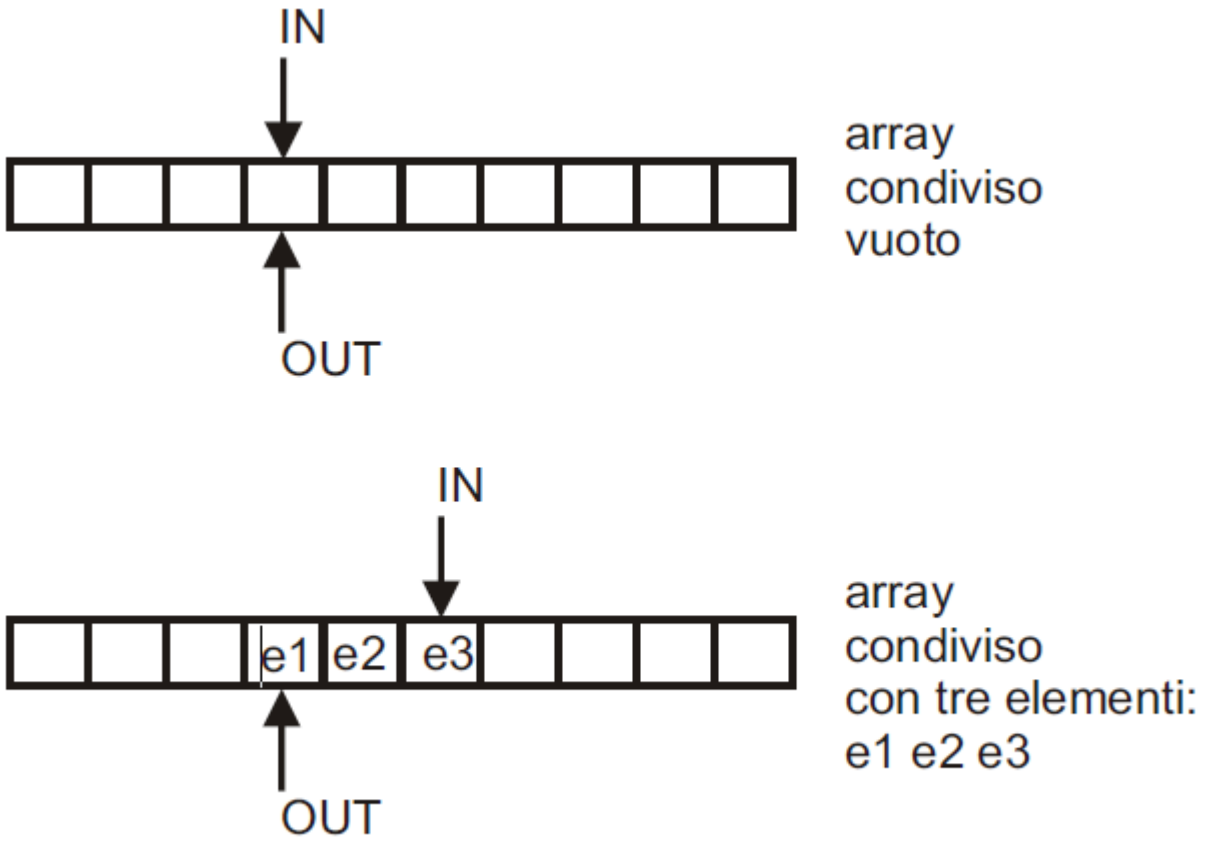
Questo programma contiene
Corse Critiche

Mutua esclusione

- **Condizione di corsa critica (Race conditions)**: quando il risultato finale dell'esecuzione di più processi concorrenti dipende dall'ordine in cui essi vengono eseguiti
- Il codice che genera corse critiche è chiamato **sezione critica**
- Operazioni con risorse condivise devono essere realizzate in modo atomico
- Risorse logiche e fisiche
- Cioè in **Mutua Esclusione:** solo un processo alla volta accede alle risorse condivise
- Approcci per garantire la mutua esclusione:
  - Software
  - Architetturali
  - Linguistici

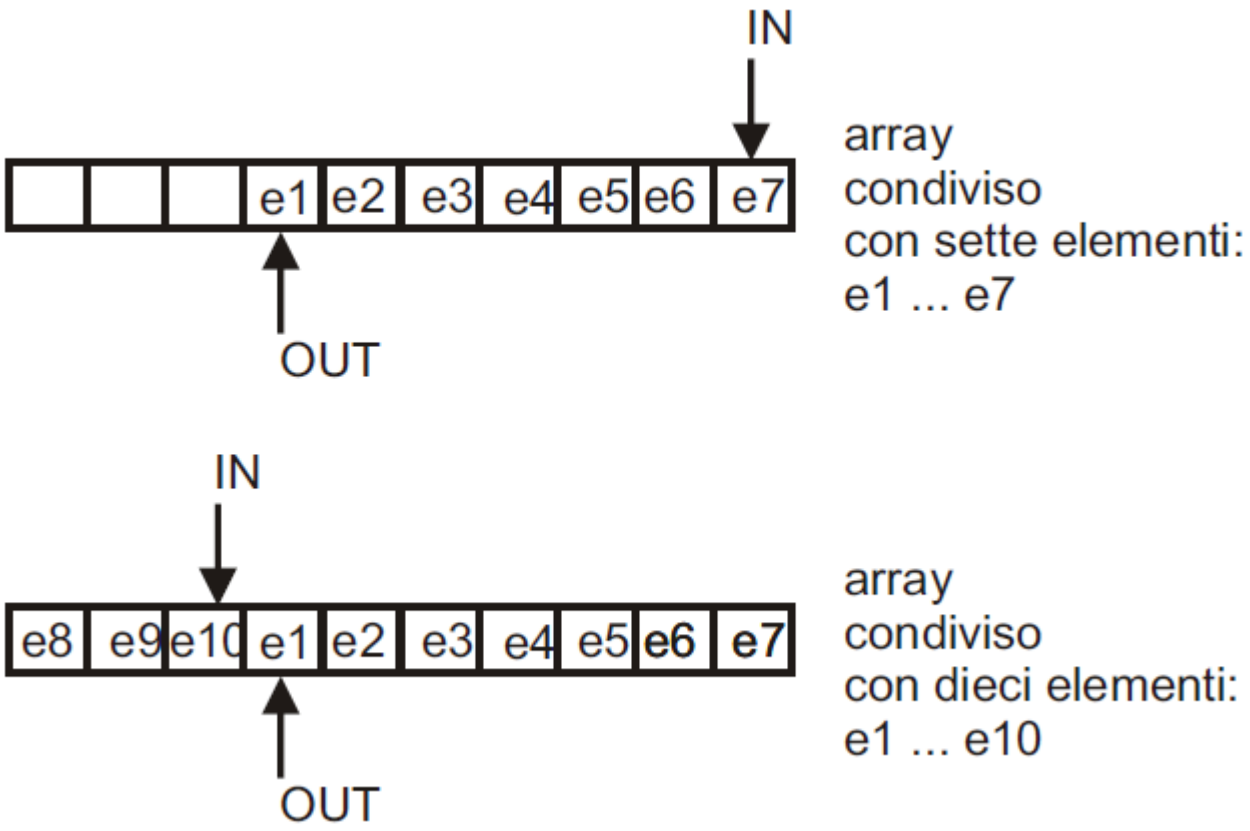
Una non soluzione software alla mutua esclusione

- Produttore consumatore attraverso un array condiviso di 10 celle



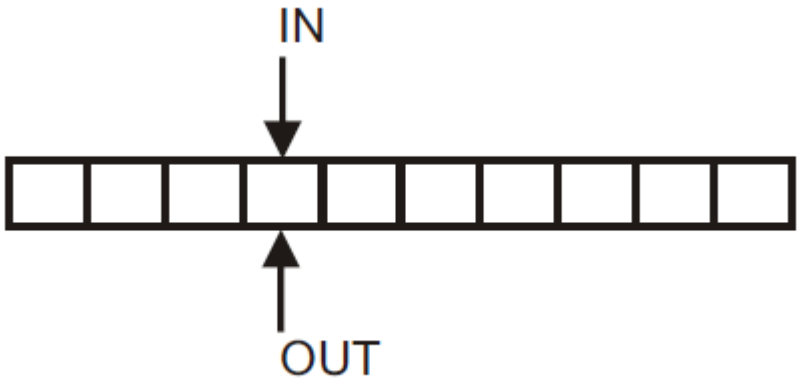
Una non soluzione software alla mutua esclusione

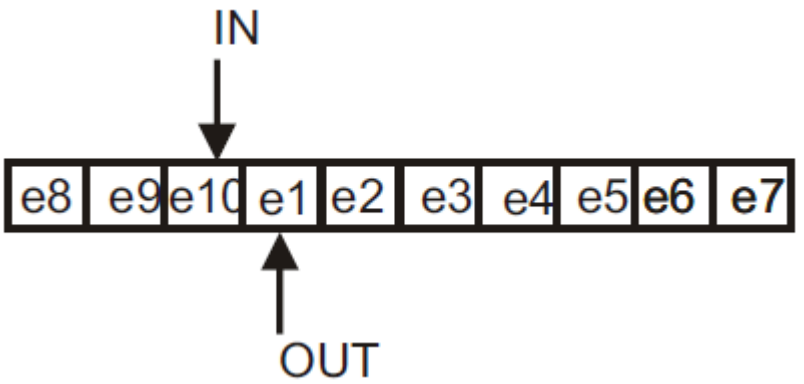
- Produttore consumatore attraverso un array condiviso



Una non soluzione software alla mutua esclusione

- Condizioni di buffer pieno/buffer vuoto





Condizione array vuoto:

IN == OUT

Condizione array pieno:

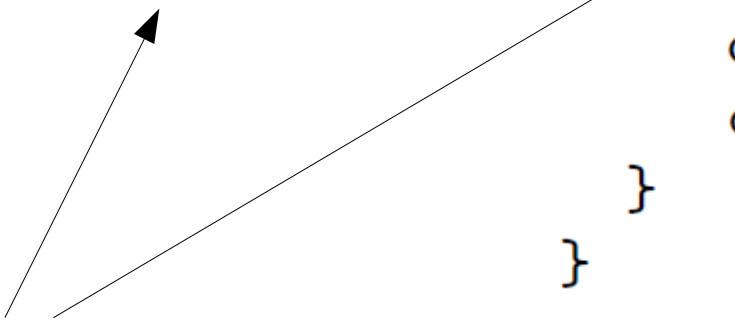
(IN +1)%N == OUT

Una non soluzione software alla mutua esclusione

- Un algoritmo di produttore/consumatore
- E' una prima non soluzione software
- Deve funzionare con N consumatori e N produttori concorrenti

```
prod(){
 while(true)
 {
    el=produci();
    while((IN+1)%N==out)){};
    buf[in]=el;
    in=(in+1)%N;
  }
}
```

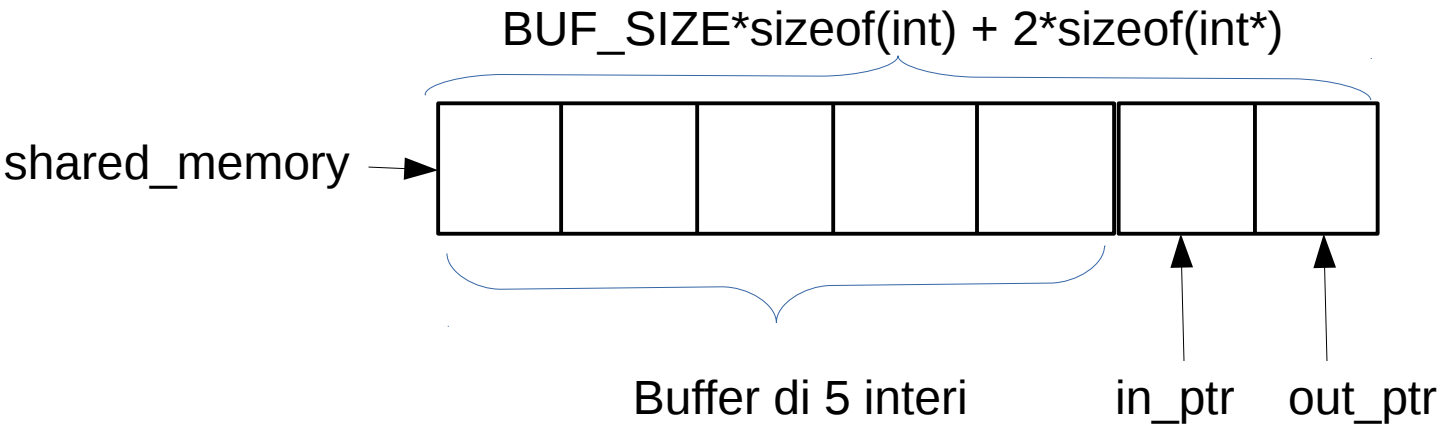
```
cons() {
 while(true)
 {
    while(in==out){};
    el=buf[out];
    out=(out+1)%N;
    consuma(el);
 }
}
```



- Spinlock
- Busy waiting
- Attesa attiva

Una non soluzione software alla mutua esclusione

- Implementazione con shared memory
  - Variabili:
    - Buffer
    - in, out
  - Struttura dati:



  - Puntatori:

```
Puntatore a Buffer = shared_memory
Ptr ad in = (int*)(shared_memory + BUF_SIZE*sizeof(int));
Ptr ad out = (int*)(shared_memory+BUF_SIZE*sizeof(int)+ sizeof(int*));
```

In pratica...

```
#define BUF_SIZE 5
#define SHARED_MEM_SIZE (BUF_SIZE)*sizeof(int) + 2*sizeof(int*)
//syncro2.c
int main (void){
    int pid, i, j, value;
    int* shared_memory;
    int *in;
    int *out;
    int *buffer;

    shared_memory=mmap(0,SHARED_MEM_SIZE,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_SHARED,-1,0);
    buffer = (int*) shared_memory;
    in = (int*) shared_memory + BUF_SIZE*sizeof(int);
    out = (int*) (shared_memory + BUF_SIZE*sizeof(int) + sizeof(int*));  *in = *out = 0;

    if ((pid = fork()) == 0)  { /* consumatore*/
        for (i = 0; i < 20; i++)  {
            while (*in == *out) ;
            value = buffer[*out]; /* leggo il buffer */
            *out = (*out + 1) % BUF_SIZE;
            printf ("\tleggo buf[%d] == %d\n", *out, value);
           }
    }
    else{    /* produttore */
        for (j = 0; j < 20; j++)   {
            while ((*in + 1) % BUF_SIZE == *out);
            buffer[*in] = j;  /* scrivo il buffer */
            *in = (*in + 1) % BUF_SIZE;
            printf ("ho scritto  %d  in buf[%d]\n", j, *in);
        }
        wait (pid);
    }
    exit (0);
}
```

Include e define da aggiungere al programma precedente

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#define BUF_SIZE 5
#define SHARED_MEM_SIZE (BUF_SIZE+2)*sizeof(int)
```

Perchè non-soluzione?

- Supponiamo ci siano 2 consumatori:

```
Cons(){
    while(true){
        While(in==out){};
        el=buf[out];
        out=(out+1)%N;
        consuma(el);
    }
}
```

```
Cons(){
    while(true){
        While(in==out){};
        el=buf[out];
        out=(out+1)%N;
        consuma(el);
    }
}
```

- Supponiamo ci sia questo Context Switch:

```
Cons(){
    while(true){
        While(in==out){};
        el=buf[out];
        out=(out+1)%N;
        consuma(el);
    }
}
```

```
Cons(){
    while(true){
        While(in==out){};
        el=buf[out];
        out=(out+1)%N;
        consuma(el);
    }
}
```

Due consumatori

```
#define SHARED_MEM_SIZE (BUF_SIZE)*sizeof(int) + 2*sizeof(int*)
//syncro5.c
int main (void){
    int pid, i, j, value;
    int* shared_memory;
    int *in;
    int *out;
    int *buffer;

    shared_memory=mmap(0,SHARED_MEM_SIZE,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_SHARED,-1,0);
    buffer = (int*) shared_memory;
    in = (int*) shared_memory + BUF_SIZE*sizeof(int);
    out = (int*) (shared_memory + BUF_SIZE*sizeof(int) + sizeof(int*));  *in = *out = 0;
    if ((pid = fork()) == 0) /* consumatore*/
        for (i = 0; i < 10; i++)  {
            while (*in == *out) ;
            value = buffer[*out]; /* leggo il buffer */
            *out = (*out + 1) % BUF_SIZE;
            printf ("\tleggo1 buf[%d] == %d\n", *out, value);
           }
    if ((pid = fork()) == 0) /* consumatore*/
           for (i = 0; i < 10; i++)  {
            while (*in == *out) ;
            value = buffer[*out]; /* leggo il buffer */
            *out = (*out + 1) % BUF_SIZE;
            printf ("\t\tleggo2 buf[%d] == %d\n", *out, value);
           }
    /* produttore */
    for (j = 0; j < 10; j++)   {
        while ((*in + 1) % BUF_SIZE == *out);
        buffer[*in] = j;  /* scrivo il buffer */
        *in = (*in + 1) % BUF_SIZE;
        printf ("ho scritto  %d  in buf[%d]\n", j, *in);
    }
    return(0);
}
```

Una soluzione di mutua esclusione

Soluzione: eseguire in Mutua Esclusione le sezioni che usano OUT

```
cons() {
  while(true)
  {

        while(in==out){};
    |---------------------|
    |   el=buf[out];      |
    |   out=(out+1)%N;    |
    |---------------------|
     consuma(el);
  }
}
```

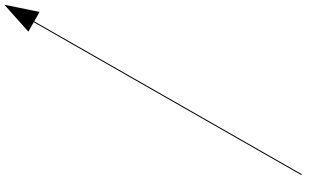
```
cons1() {
   while(true)
   {

         while(in==out)();
     |----------------------|
     |    el=buf[out];      |
     |    out=(out+1)%N;    |
     |----------------------|
      consuma(el);
   }
}
```

⇒ Uso un arbitro che controlla l'esecuzione

Semafori

Dijkstra introduce due operazioni,**down() e up()**, che operano su una variabile intera *s*.
In principio:

```
down(s)
{   while (s<=0); /* loop di attesa: BUSY WAITING */
    s--;
}
up(s)
{ s++; }
```



Semafori spinlock

Ma: le operazioni DEVONO ESEGUIRE ATOMICAMENTE!

Per questo gli operatori down() e up() sono realizzate dal kernel!

Semafori binari (mutex) e semafori contatori

Semafori

- Protezione semaforica di una sezione critica

```
Processo1(){
   while(1){
      down(s);
      Sezione_Critica1();
      up(s);
      Sezione_NON_Critica1();
   }
}
```

```
Processo2(){
   while(1){
      down(s) ;
      Sezione_Critica2();
      up(s);
      Sezione_NON_Critica1();
   }
}
```

```
Processo2(){
   while(1){
      down(s) ;
      Sezione_Critica2();
      up(s);
      Sezione_NON_Critica1();
   }
}
```

- I semafori spinlock sono adatti a brevi sezioni critiche
- Per sezioni critiche lunghe, semafori waitlock

Semafori waitlock

- Anche I semafori waitlock eseguono atomicamente
- Inoltre, i semafori waitlock devono accedere alla coda di processi dello scheduler:

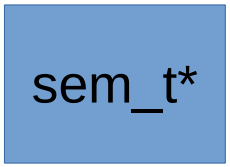
```
down(s)
{
   if(s<=0)
       aggiungi il descrittore del processo in coda su s;
   else s--;
 }


 up(s)
 {
   if(c'e' un descrittore in attesa su s)
       esegui il processo;
   else
       s++;
  }
```

Semafori Posix in Linux

- Semafori senza nome.
- Chiamate di sistema:
  - Richiedono #include <semaphore.h> e #include <pthread.h>
  - int sem_init(sem_t *sem, int pshared, unsigned int value);
  - int sem_wait(sem_t *sem); /* decrementa sem. Se 0 aspetta */
  - int sem_post(sem_t *sem); /* **incrementa sem** */
  - int sem_getvalue(sem_t *sem, **int *sval**); /* legge il valore corrente di sem in sval */
  - int sem_close(sem_t *sem; /* chiude sem */
  - int sem_unlink(const char **name**); /* rimuove */

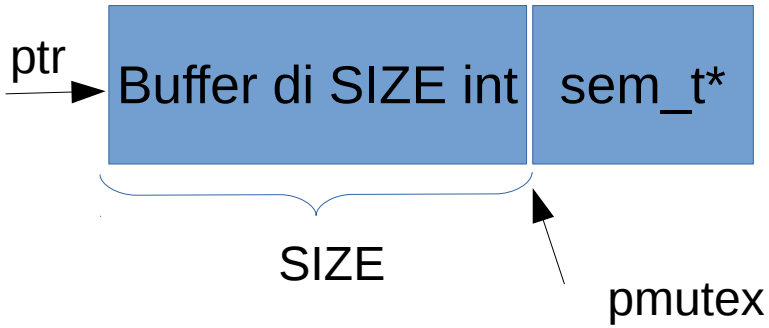
Esempio di condivisione semafori senza nome tra processi parenti in shared memory

1- Creazione di una memoria condivisa per il puntatore al mutex




```
sem_t* pmutex;       sem_t  mutex;

pmutex = mmap(0, sizeof(sem_t*), PROT_READ|PROT_WRITE,
                              MAP_ANONYMOUS|MAP_SHARED, -1, 0);
sem_init(pmutex,1,1);
```

2- Supponiamo che ci sia anche un buffer in N elementi. Per esempio




```
sem_t* pmutex;
int* ptr = mmap(0, sizeof(int)*SIZE+sizeof(sem_t*),
        PROT_READ|PROT_WRITE,
        MAP_ANONYMOUS|MAP_SHARED, -1, 0);

pmutex=(sem_t*)(ptr+sizeof(int)*SIZE);

sem_init(pmutex,1,1);
```

Riprendiamo il programma syncro1.c

Corsa critica; definiamo un mutex

```
#include <semaphore.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#define SIZE 20
//syncro4.c
int main(int argc, char **argv){
  int fd, i, val, pid,*ptr;
  sem_t mutex;
  sem_t* pmutex;
  ptr = mmap(0, sizeof(int)*SIZE+sizeof(sem_t*), PROT_READ|PROT_WRITE, MAP_ANONYMOUS|
                                                    MAP_SHARED, -1, 0);
  pmutex=(sem_t*)ptr+sizeof(int)*SIZE;
  sem_init(pmutex,1,1);
  pid=fork();
  if (pid == 0) {
    for (i = 0; i < SIZE; i++) {
        val=(*ptr)++;
        printf("figlio: %d\n", val);
    }
    exit(0);
  }
  else{
     for (i = 0; i < SIZE; i++) {
        val=(*ptr)++;
        usleep(1e2);
        printf("padre:  %d\n", val);
     }
   }
   wait();   return(0);
}
```

Riprendiamo il programma syncro1.c

Inseriamo il mutex

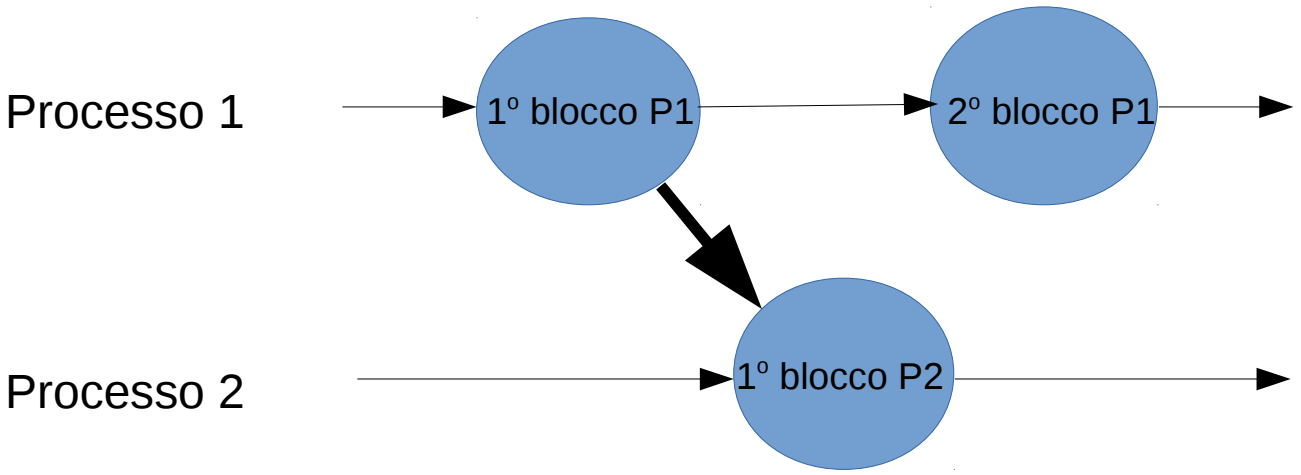
```
#include <semaphore.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#define SIZE 20
//syncro6.c
int main(int argc, char **argv){
  int fd, i, val, pid,*ptr;
  sem_t mutex;
  sem_t* pmutex;
  ptr = mmap(0, sizeof(int)*SIZE+sizeof(sem_t*),
             PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_SHARED, -1, 0);
  pmutex=(sem_t*)(ptr+sizeof(int)*SIZE);
  sem_init(pmutex,1,1);
  pid=fork();
  if (pid == 0) {
    for (i = 0; i < SIZE; i++) {
        sem_wait(pmutex);
        val=(*ptr)++;    printf("figlio: %d\n", val);
        sem_post(pmutex);
    }
  }
  else{
     for (i = 0; i < SIZE; i++) {
        sem_wait(pmutex);
        val=(*ptr)++;    sleep(1e2); printf("padre:  %d\n", val);
        sem_post(pmutex);
     }
   }
   wait();   return(0);
}
```

Semafori Posix in Linux

- Semafori con nome.
- Chiamate di sistema:
  - Richiedono #include <semaphore.h> e #include <pthread.h>
  - sem_t ***sem_open(const char name,int oflag,mode_t mode,int value);***
    / apre il semaforo, crea con O_CREATE, ritorna sem_t *sem; */
  - int sem_wait(sem_t *sem); /* decrementa sem. Se 0 aspetta */
  - int sem_post(sem_t *sem); /* **incrementa sem** */
  - int sem_getvalue(sem_t *sem, **int *sval**); /* legge il valore corrente di sem in sval */
  - int sem_close(sem_t *sem; /* chiude sem */
  - int sem_unlink(const char **name**); /* rimuove */

Sincronizzazione processi
Protezione (mutua esclusione)

- Sincronizzazione processi
  - Dividendo le esecuzioni dei processi in blocchi, introduzione di precedenze tra blocchi
  - Esempio:



- Protezione: uso di mutex per mutua esclusione

Sincronizzazione con semafori con nome tra processi indipendenti I/II

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>        /* mnemonici in mode */
#include <fcntl.h>           /* mnemonici in oflag */
#include <semaphore.h>
//syncro7.c
int main (void)
{    int fd,i, pid,size=sizeof(float);
     float* shared_memory;
     float* p;
     float* q;
     sem_t * sem1_id;
     sem_t * sem2_id;

     fd = shm_open("mymem",  O_CREAT | O_EXCL |O_RDWR, S_IRUSR | S_IWUSR);
     ftruncate(fd,size);
     shared_memory = mmap(NULL ,size,PROT_READ | PROT_WRITE, MAP_SHARED ,fd, 0);

     sem1_id=sem_open("/mysem1", O_CREAT, S_IRUSR | S_IWUSR, 1);
     sem2_id=sem_open("/mysem2", O_CREAT, S_IRUSR | S_IWUSR, 0);

     sem_wait(sem2_id);

     q=shared_memory;
     for(i=0;i<10;i++) {printf("sshh2. *q=%f\n", (*q)++);}
}
```

Un processo che incrementa var

Semafori con nome

Aspetta il via dall'altro processo (sincronizzazione)

Sincronizzazione con semafori con nome tra processi indipendenti II/II

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>        /* mnemonici in mode */
#include <fcntl.h>           /* mnemonici in oflag */
#include <semaphore.h>
//syncro8.c    da usare con syncro7.c
int main (void){
    int fd,i, pid,size=5*sizeof(float);
    float* shared_memory;
    float* p;
    float* q;
    sem_t * sem1_id;
    sem_t * sem2_id;

    fd = shm_open("mymem",  O_EXCL |O_RDWR, S_IRUSR | S_IWUSR);
    shared_memory = mmap(NULL ,size,PROT_READ | PROT_WRITE, MAP_SHARED ,fd, 0);

    sem1_id = sem_open("/mysem1", O_CREAT, S_IRUSR | S_IWUSR, 1);
    sem2_id = sem_open("/mysem2", O_CREAT, S_IRUSR | S_IWUSR, 0);

    p=shared_memory;
    sem_post(sem2_id);

    for(i=0; i<10; i++) printf ("\tsshh1 *p = %f\n", (*p)++);

    shm_unlink("mymem");
}
```

Altro processo che incrementa var in concorrenza

Semafori con nome

Da' il via al processo syncro7 (sincronizzazione)

Protezione e sincronizzazione con semafori con nome tra processi indipendenti I/II

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>        /* mnemonici in mode */
#include <fcntl.h>           /* mnemonici in oflag */
#include <semaphore.h>
//syncro9.c
int main (void)
{    int fd,i, pid,size=sizeof(float);
     float* shared_memory;
     float* p;
     float* q;
     sem_t * sem1_id;
     sem_t * sem2_id;

     fd = shm_open("mymem",  O_CREAT | O_EXCL |O_RDWR, S_IRUSR | S_IWUSR);
     ftruncate(fd,size);
     shared_memory = mmap(NULL ,size,PROT_READ | PROT_WRITE, MAP_SHARED ,fd, 0);

     sem1_id=sem_open("/mysem1", O_CREAT, S_IRUSR | S_IWUSR, 1);
     sem2_id=sem_open("/mysem2", O_CREAT, S_IRUSR | S_IWUSR, 0);

     sem_wait(sem2_id);

     q=shared_memory;
     for(i=0;i<10;i++){sem_wait(sem1_id); printf("A. *q=%f\n", (*q)++); sem_post(sem1_id);}
}
```

Un processo

Crea la memoria condivisa con nome 'mymem'

Semafori con nome

Aspetta il via dall'altro processo (**sincronizzazione**)

protezione

Protezione e sincronizzazione con semafori con nome tra processi indipendenti II/II

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>        /* mnemonici in mode */
#include <fcntl.h>           /* mnemonici in oflag */
#include <semaphore.h>
//syncro10.c
int main (void)
{
    int fd,i, pid,size=5*sizeof(float);
    float* shared_memory;
    float* p;
    float* q;
    sem_t * sem1_id;
    sem_t * sem2_id;

    fd = shm_open("mymem",  O_EXCL |O_RDWR, S_IRUSR | S_IWUSR);
    shared_memory = mmap(NULL ,size,PROT_READ | PROT_WRITE, MAP_SHARED ,fd, 0);

    sem1_id = sem_open("/mysem1", O_CREAT, S_IRUSR | S_IWUSR, 1);
    sem2_id = sem_open("/mysem2", O_CREAT, S_IRUSR | S_IWUSR, 0);

    p=shared_memory;
    sem_post(sem2_id);
    for(i=0;i<10;i++){sem_wait(sem1_id); printf("\tB *p = %f\n", (*p)++);sem_post(sem1_id);}

    shm_unlink("mymem");
}
```

Altro processo

Si collega alla memoria condivisa con nome 'mymem'

Semafori con nome

Dà il via all'altro processo (sincronizzazione)

protezione

Note conclusive sui Semafori Posix in Linux

- Necessitano di <semaphore.h>
- Cancellazione semafori:
  - `int sem_unlink(const char *name);`
- Cancellazione shared memory:
  - `int shm_unlink(const char *name);`
- Devono essere compilati con la libreria pthread:
  - `gcc source.c -o source -lpthread`
- Se uso semafori e shared memory:
  - `gcc source.c -o source -lpthread -lrt`