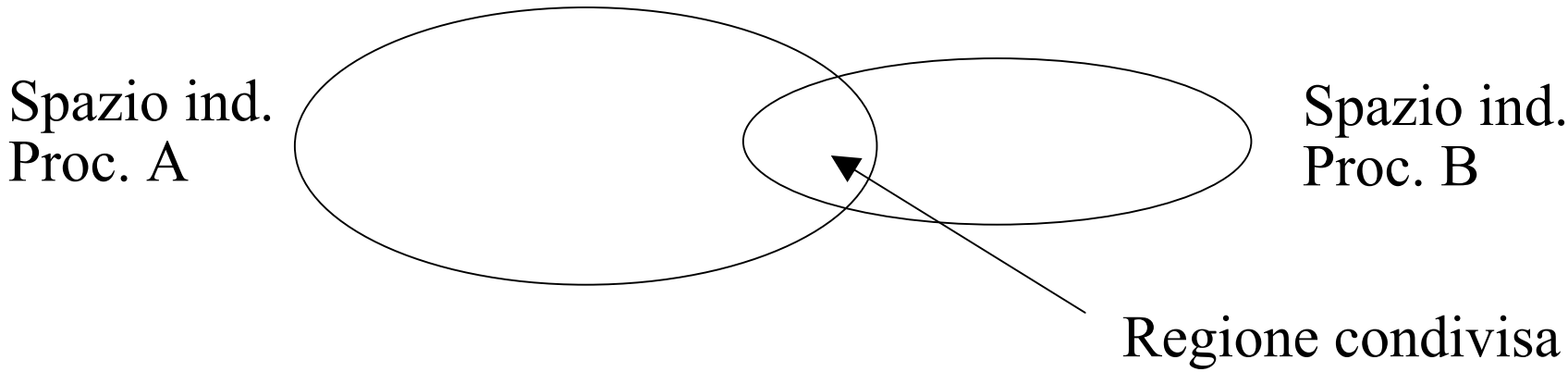
Sistemi Operativi I
Corso di Laurea in Ingegneria Informatica
Facolta' di Ingegneria, Universita' "La Sapienza"
Docente: Francesco Quaglia

Memoria condivisa e semafori:

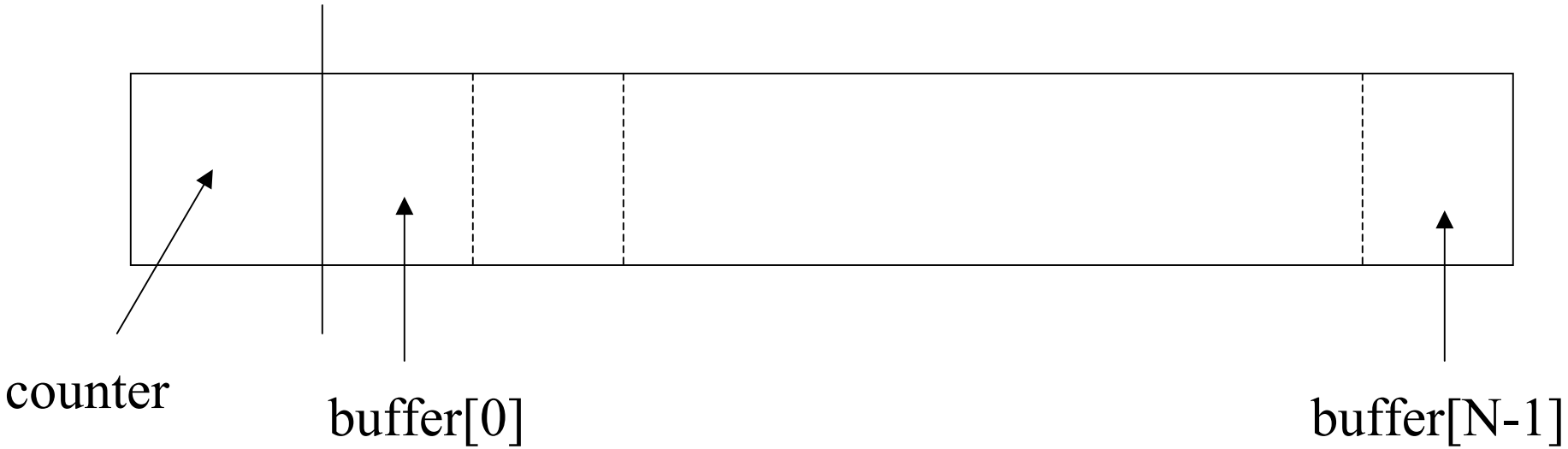
1. Memoria condivisa
2. Sezioni critiche
3. Approcci software ed hardware
4. Semafori
5. Memoria condivisa e semafori in sistemi operativi attuali (NT/UNIX)

Memoria condivisa



- la regione condivisa e' costituita da una sequenza di bytes che possono essere letti e/o scritti da entrambi i processi
- dal punto di vista del sistema operativo la regione condivisa e' equivalente ad una qualsiasi regione non condivisa dello spazio di indirizzamento
- il contenuto della memoria condivisa dipende dalla sequenza di accessi in scrittura da parte dei processi
- la massima porzione della memoria condivisa che puo'essere letta/scritta da un processo in modo atomico dipende dalle istruzioni di linguaggio macchina (relazioni con la taglia del BUS dati)

Produttore/consumatore tramite memoria condivisa



PRODUTTORE

Repeat

<produce X>
while counter = N do no-op;
buffer[in] := X;
in := (in+1)mod(N)
counter := counter + 1;

until false

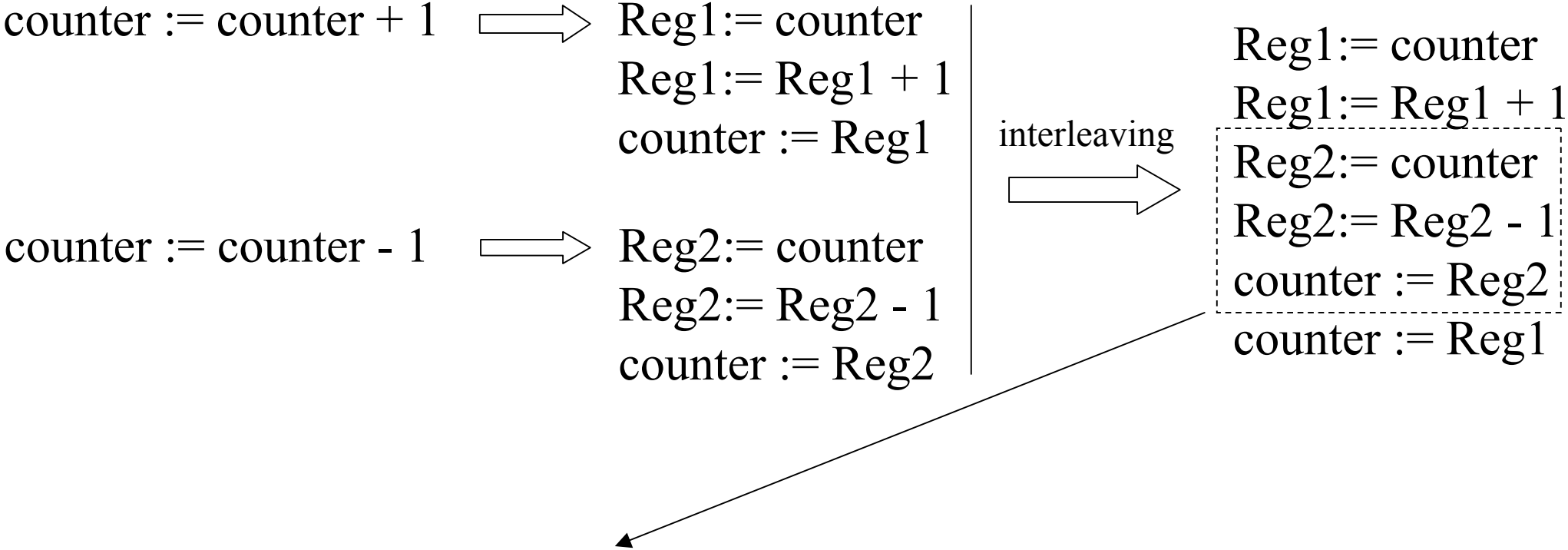
CONSUMATORE

Repeat

while counter = 0 do no-op;
Y := buffer[out];
out := (out+1)mod(N)
counter := counter - 1;
<consume Y>

until false

Sezioni critiche



Una sezione critica e' una porzione di traccia ove

- un processo puo' leggere/scrivere dati condivisi con altri processi
- la correttezza del risultato dipende dall'interleaving delle tracce dei processi

---

Risoluzione del problema della sezione critica

- permettere l'esecuzione della porzione di traccia relativa alla sezione critica come se fosse un'azione atomica

Vincoli per il problema della sezione critica

Mutua esclusione

• quando un processo entra nella sezione critica nessun altro processo puo' eseguire nella sezione critica

Progresso

• un processo che lo chiede deve essere ammesso alla sezione critica senza ritardi in caso in cui nessun altro processo si trovi nella sezione critica

Attesa limitata

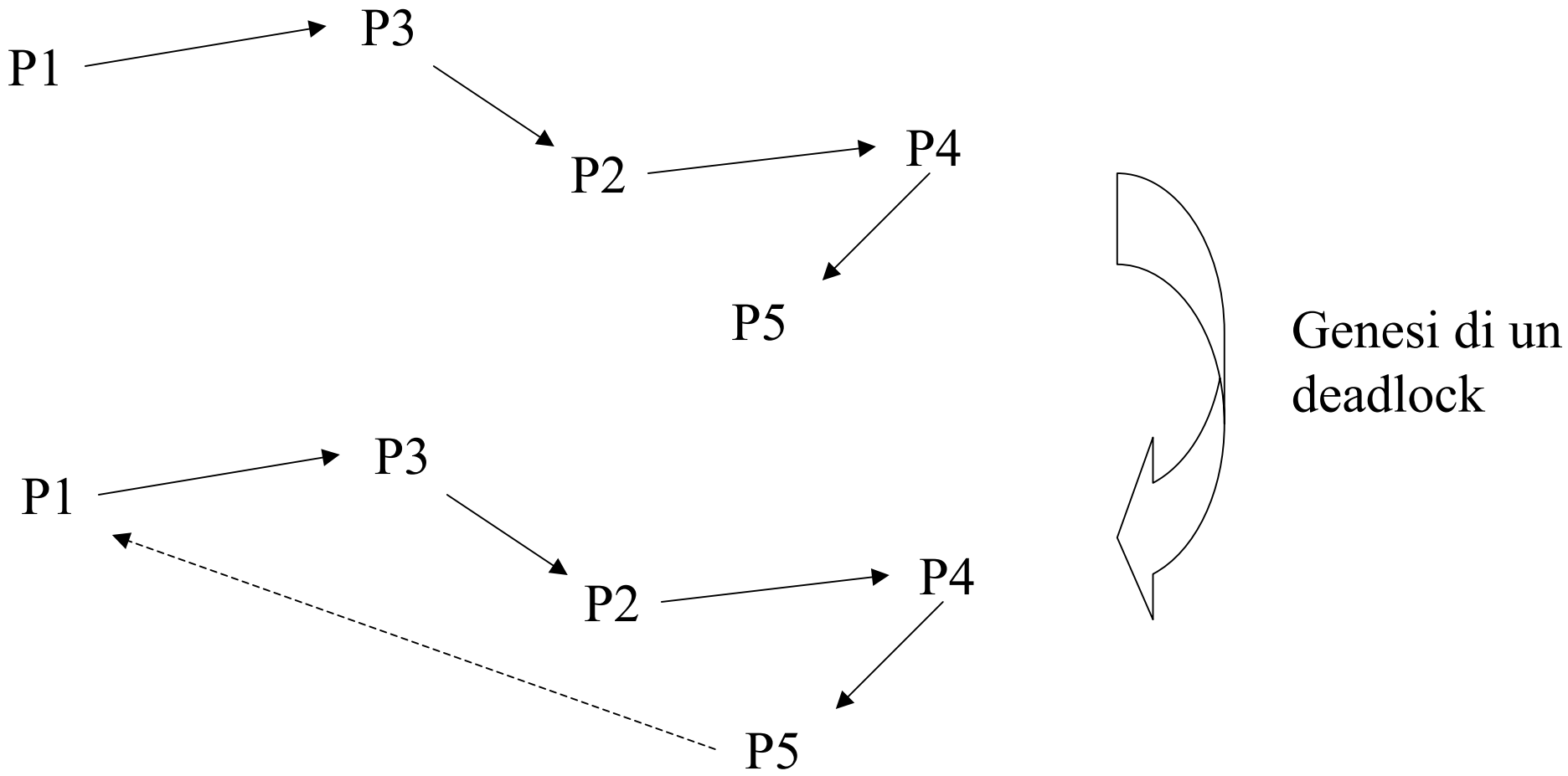
• un processo che lo chiede deve essere ammesso ad eseguire la sezione critica in un tempo limitato (non ci devono essere **stalli** o **starvation**)

Approcci risolutivi

• algoritmi di mutua esclusione
• harware specializzato
• semafori

Stallo (deadlock)

Un insieme di processi P1, ..., Pn e' coinvoltoin uno stallo se ognuno dei processi e' in attesa (attiva o passiva) di un evento che puo' essere causato solo da un altro dei processi dell'insieme



Algoritmi di mutua esclusione: Dekker

var turno: **int**;

Processo X	**Processo Y**
While turno ≠ X **do no-op;**	**While** turno ≠ Y **do no-op;**
<sezione critica>;	<sezione critica>;
turno := Y;	turno := X;

I processi vanno in alternanza stretta nella sezione critica

- non c'e' garanzia di progresso
- la velocita' di esecuzione e' limitata dal processo piu' lento

---

I processi lavorano come coroutine (ovvero routine che si passano mutuamente il controllo), classiche della strutturazione di un singolo processo, ma inadeguate a processi concorrenti

Secondo tentativo

var flag: **array**[1,n] **of boolean**;

Processo X

While flag[Y] **do no-op;**
flag[X] := TRUE;
<sezione critica>;
flag[X] := FALSE;

Processo Y

While flag[X] **do no-op;**
flag[Y] := TRUE;
<sezione critica>;
flag[Y] := FALSE;

I processi non vanno in alternanza stretta nella sezione critica

- c'e' garanzia di progresso
- non c'e' garanzia di mutua esclusione (problema che diviene evidente nel caso di numero elevato di processi)

Terzo tentativo

var flag: **array**[1,n] **of boolean**;

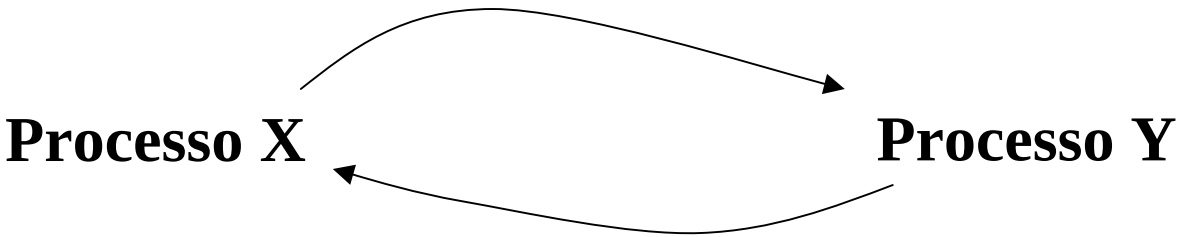
Processo X

flag[X] := TRUE;
While flag[Y] **do no-op;**
<sezione critica>;
flag[X] := FALSE;

Processo Y

flag[Y] := TRUE;
While flag[X] **do no-op;**
<sezione critica>;
flag[Y] := FALSE;

Possibilita' di deadlock, non c'e garanzia di attesa limitata



Quarto tentativo

var flag: **array**[1,n] **of boolean**;

Processo X

```
flag[X] := TRUE;
While flag[Y] do {
        flag[X] := FALSE;
       <pausa>;
        flag[X] := TRUE;
}
<sezione critica>;
flag[X] := FALSE;
```

Processo Y

```
flag[Y] := TRUE;
While flag[X] do {
        flag[Y] := FALSE;
       <pausa>;
        flag[Y] := TRUE;
}
<sezione critica>;
flag[Y] := FALSE;
```

Possibilita' di starvation, non c'e garanzia di attesa limitata

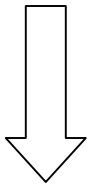
Algoritmo del fornaio

Basato su assegnazione di numeri per prenotare un turno di accesso alla sezione critica

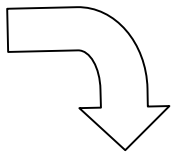
```
var choosing: array[1,n] of boolean;
    number: array[1,n] of int;
repeat {
    choosing[i] := TRUE;
    number [i] := <max in array number[] + 1>;
    choosing[i] := FALSE;
    for j = 1 to n do {
        while choosing[j] do no-op;
        while number[j] ≠0 and (number [j],j)< (number [i],i) do no-op;
    }
    <sezione critica>;
    number[i] := 0;

}until FALSE
```

Approcci hardware

- disabilitare le interruzioni (valido nel caso di uniprocessori)
- istruzioni di basso livello per il test e la manipolazione di informazioni in modo atomico (valido anche per i multiprocessori)



```
function test_and_set(var z: int) : boolean;
{
 if (!z) {
        z := 1;
        test_and_set := TRUE;
 }
 else test_and_set := FALSE;
}
```



```
var serratura: int;

Processo X

While !test_and_set(serratura) do no-op;
<sezione critica>;
serratura := 0
```

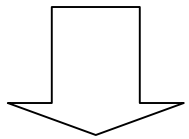
Semafori: modello di riferimento

Un semaforo e' una variabile intera con 3 operazioni associate

- inizializzazione con un valore non negativo

- una operazione **wait** che ne decrementa il valore di una unita' ed induce una attesa sul processo che la esegue in caso il valore del semaforo diventa negativo

- una operazione **signal** che ne incrementa il valore di una unita'; se il valore ottenuto non e' positivo allora uno dei processi in attesa a causa di una operazione di wait precedentemente eseguita viene liberato dall'attesa



Le operazioni devono essere eseguite in <u>modo atomico</u>

Semafori binari

- il valore assunto puo' essere solo 1 oppure 0

Implementazioni

```
procedure Wait(var s: int);
        ATOMIC s := s-1;
        while s<0 do no-op;
```

```
procedure Signal(var s: int);
        ATOMIC s := s+1;
```

Spin-lock semaphore (livello applicazione)

Livello kernel

```
type semaphore = record
                        value: int;
                        L: list of processes;
                 end;

procedue Wait(var s: semaphore);
        s.value := s.value - 1;
        if s.value < 0 {add process to s.L; block process;}

procedure Signal(var s: semaphore);
        s.value = s.value + 1;
        if s.value <= 0 {delete a process P from s.L; unblock P;}
```

Memoria condivisa in sistemi UNIX

int shmget(key_t key, int size, int flag)	
Descrizione	invoca la creazione di un'area di memoria condivisibile
Parametri	1) key: chiave per identificare la memoria condivisibile in maniera univoca nel sistema 2) size: taglia in byte della memoria condivisibile 3) flag: specifica della modalita' di creazione (IPC_CREAT, IPC_EXCL, definiti negli header file sys/ipc.h e sys/shm.h) e dei permessi di accesso
Descrizione	identificatore numerico per la memoria in caso di successo (descrittore), -1 in caso di fallimento

NOTA

Il descrittore indicizza questa volta una struttura unica valida per qualsiasi processo

Controllo su una memoria condivisa

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

Descrizione invoca l'esecuzione di un comando su una coda di messaggi

---

Parametri

1) ds_shm: descrittore della memoria condivisa su cui si vuole operare
2) cmd: specifica del comando da eseguire (IPC_RMID, IPC_STAT, IPC_SET)
3) buff: puntatore al buffer con eventuali parametri per il comando

---

Descrizione -1 in caso di fallimento

IPC_RMID invoca la rimozione della memoria condivisa dal sistema

```
struct shmid_ds {                                          ↑
        struct    ipc_perm shm_perm;  /* operation perms */
        int  shm_segsz;          /* size of segment (bytes) */
        time_t    shm_atime;          /* last attach time */
        time_t    shm_dtime;          /* last detach time */
        time_t    shm_ctime;          /* last change time */
        unsigned short shm_cpid; /* pid of creator */
        unsigned short shm_lpid; /* pid of last operator */
        short     shm_nattch;         /* no. of current attaches */
      };

      struct ipc_perm
      {
       key_t  key;
       ushort uid;   /* owner euid and egid */
       ushort gid;
       ushort cuid;  /* creator euid and egid */
       ushort cgid;
       ushort mode;  /* lower 9 bits of shmflg */
       ushort seq;   /* sequence number */
      };
```

Attacco/distacco dallo spazio di indirizzamento

void *shmat(int ds_shm, void *addr, int flag)

Descrizione invoca il collegamento di una memoria condivisa allo spazio di indirizzamento del processo

Parametri
1) ds_shm: descrittore della memoria condivisa da collegare
2) *addr: indirizzo preferenziale per il collegamento (NULL come default)
3) flag: modalita' di accesso (`SHM_R`, `SHM_W`, `SHM_RW`)

Descrizione indirizzo valido per l'accesso alla memoria in caso di successo, -1 in caso di fallimento

int shmdet(const void *addr)

Descrizione invoca lo scollegamento di una memoria condivisa dallo spazio di indirizzamento del processo

Parametri *addr: indirizzo della memoria da scollegare

Descrizione -1 in caso di fallimento

Esempio di applicazione: trasferimento stringhe tra processi

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define DISP_ 20
#define Errore_(x) { puts(x); exit(1); }

char messaggio[256];

void scrittore(int ds) {
    char *p;

   p = shmat(ds, 0 , SHM_W);
    if ( p == (char*) -1 ) Errore_("Errore nella chiamata shmat");

    puts("Digitare le parole da trasferire in memoria condivisa ('quit‘ per terminare):");
    do {
        scanf("%s", messaggio);
        strncpy(p, messaggio, DISP_);
        p += DISP_;
    } while( (strcmp(messaggio,"quit") != 0));
    exit(0);
}
```

```
void lettore(int ds) {
      char *p;

      p = shmat(ds, 0 , SHM_R);
      if ( p == (char*) -1 ) Errore_("Errore nella chiamata shmat");
      printf("Contenuto memoria condivisa: \n");
      while( (strcmp(p,"quit") != 0)) {
         printf("%s\n", p);
         p += DISP_;
      }
    exit(0);
}

int main(int argc, char *argv[]) {
      int  ds_shm, rit, status;  long chiave=30;

      ds_shm = shmget(chiave, 1024, IPC_CREAT|0666);
      if ( ds_shm == -1 ) Errore_("Errore nella chiamata shmget");

      if (fork()) wait(&status);
      else scrittore(ds_shm);
      if (fork()) wait(&status);
      else lettore(ds_shm);

      ris = shmctl(id_shm, IPC_RMID, NULL);
      if ( ris == -1 ) Errore_("Errore nella chiamata shmctl");
}
```

File mapping NT/2000

```
HANDLE CreateFileMapping(HANDLE hFile,
                  LPSECURITY_ATTRIBUTES lpAttributes,
                  DWORD flProtect,
                  DWORD dwMaximumSizeHigh,
                  DWORD dwMaximumSizeLow,
                  LPCTSTR lpName)
```

Descrizione

- invoca il mapping di un file in memoria

Restituzione

- handle al mapping del file

Parametri

- hFile: handle ad un file aperto
- lpAttributes: puntatore a una struttura SECURITY_ATTRIBUTES
- flProtect: modalità di accesso al mapping del file (es. PAGE_READWRITE)
- dwMaximumSizeHigh: dimensione massima del mapping del file (32 bit piu' significativi)
- dwMaximumSizeLow: dimensione massima del mapping del file (32 bit meno significativi)
- lpName: nome del mapping

Apertura di un file mapping

```
HANDLE OpenFileMapping(DWORD dwDesiredAccess,
                       BOOL bInheritHandle,
                       LPCTSTR lpName)
```

Descrizione

• accede a un mapping di file esistente

Restituzione

• handle al mapping del file

Parametri

• dwDesiredAccess: modalità di accesso richiesta al mapping di file

• bInheritHandle: specifica se l'handle restituito deve essere ereditato da processi figli

•lpName: nome del mapping gia' esistente

Collegamento di file mapping

```
LPVOID MapViewofFile(HANDLE hFileMappingObject,
                     DWORD dwDesiredAccess,
                     DWORD dwFileOffsetHigh,
                     DWORD dwFileOffsetLow,
                     SIZE_T dwNumberOfBytesToMap)
```

Descrizione

• innesta un mapping di file aperto nello spazio di indirizzamento del processo

Restituzione

• un puntatore all'area di memoria contenente il mapping del file, NULL in caso di fallimento

Parametri

• hFileMappingObject: handle ad un file mapping
• dwDesiredAccess: modalita' di accesso desiderata (es. PAGE_READWRITE)
• dwFileOffsetHigh:punto del file mapping da cui iniziare l'innesto (32 bit piu' significativi)
• dwFileOffsetLow: punto del file mapping da cui iniziare l'innesto (32 bit meno significativi)
• dwNumberOfBytesToMap: numero di bytes da innestare nello spazio del processo

Scollegamento di file mapping

```
BOOL UnmapViewOfFile(LPCVOID lpBaseAddress)
```

Descrizione

- distacca un mapping di file aperto dallo spazio di indirizzamento del processo

Restituzione

- 0 in caso di fallimento

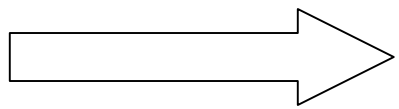
Parametri

- lpBaseAddress: puntatore all'inizio di un'area di memoria innestata tramite MapViewofFile()

Esempio di applicazione: trasferimento stringhe tra processi

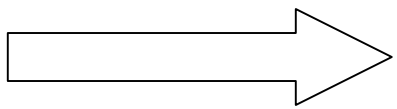
```
#include <windows.h>
 #include <stdio.h>

  #define DISP_ 20
  #define Errore_(x) { puts(x); ExitProcess(1); }

  char messaggio[256];

   void scrittore(HANDLE mapping) {
       char *p;

       p = (char *) MapViewOfFile(mapping, FILE_MAP_WRITE,  0, 0, 0);
       if ( p == NULL ) Errore_("Errore nella chiamata MapViewOfFile");

       puts("Digitare le parole da trasferire in memoria condivisa ('quit' per terminare):");
       do {
          scanf("%s", messaggio);
          strncpy(p, messaggio, DISP_);
          p += DISP_;
       } while( (strcmp(messaggio,"quit") != 0));

       ExitProcess(0);
    }
```

continua



```
void lettore(HANDLE mapping) {
      char *p;

      p = (char *)MapViewOfFile(mapping, FILE_MAP_WRITE, 0, 0, 0);
      if ( p == NULL ) Errore_("Errore nella chiamata MapViewOfFile");

      printf("Contenuto memoria condivisa: \n");
      while( (strcmp(p,"quit") != 0)) {
         printf("%s\n", p);
         p += DISP_;
      }
      ExitProcess(0);
 }

  int main(int argc, char *argv[]) {
      HANDLE mapping;  BOOL newprocess;  STARTUPINFO si; PROCESS_INFORMATION pi;

     if (argc==1) {
   /* Creazione/accesso segmento memoria condivisa */
       mapping = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE,
                              0, 50000, "my_mapping");
       if ( mapping ==  INVALID_HANDLE_VALUE) Errore_("Errore nella CreateFileMapping");

       /* genero il processo scrittore */
       memset(&si, 0, sizeof(si));
       memset(&pi, 0, sizeof(pi));
       si.cb = sizeof(si);
```

continua



```
newprocess = CreateProcess(".\\mapper.exe", ".\\mapper.exe scrittore",  NULL,  NULL, FALSE,
                                    NORMAL_PRIORITY_CLASS, NULL,  NULL, &si, &pi);
       if (newprocess == 0) { printf("Errore nella generazione dello scrittore!\n"); ExitProcess(-1); }
  WaitForSingleObject(pi.hProcess, INFINITE);
  memset(&si, 0, sizeof(si)); /* genero il processo lettore */
  memset(&pi, 0, sizeof(pi));
  si.cb = sizeof(si);
  newprocess = CreateProcess(".\\mapper.exe", ".\\mapper.exe lettore", NULL, NULL, FALSE,
                              NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi);
       if (newprocess == 0) {printf("Errore nella generazione dello scrittore!\n");
                                    ExitProcess(-1);
       }
   WaitForSingleObject(pi.hProcess, INFINITE);
}
      else if (argc == 2 && strcmp(argv[1], "scrittore") == 0){

       mapping = OpenFileMapping(FILE_MAP_WRITE,  FALSE, "my_mapping"); /* Apro il file mapping */
       if (mapping == INVALID_HANDLE_VALUE) {printf("Errore nell'aperura del mapping!\n");  ExitProcess(-1)  }
        scrittore(mapping); /* chiamo lo scrittore*/
      }
      else if (argc == 2 && strcmp(argv[1], "lettore") == 0){
       mapping = OpenFileMapping(FILE_MAP_WRITE, FALSE, "my_mapping"); /* Apro il file mapping */
        if (mapping == INVALID_HANDLE_VALUE) {printf("Errore nell'aperura del mapping!\n");
                                                                ExitProcess(-1);
        }
       lettore(mapping); /* chiamo il lettore */
      }
   return(0);
 }
```

Semafori in sistemi UNIX

int semget(key_t key, int size, int flag)

Descrizione invoca la creazione di unsemaforo

Parametri
1) key: chiave per identificare il semaforo in maniera univoca nel sistema
2) size: numero di elementi del semaforo
3) flag: specifica della modalita' di creazione (IPC_CREAT, IPC_EXCL, definiti negli header file sys/ipc.h e sys/shm.h) e dei permessi di accesso

Descrizione identificatore numerico per il semaforo in caso di successo (descrittore), -1 in caso di fallimento

NOTA
Il descrittore indicizza questa volta una struttura unica valida per qualsiasi processo

Controllo su un semaforo

int semctl(int ds_sem, int sem_num, int cmd, union semun arg)

Descrizione invoca l'esecuzione di un comando su una coda di messaggi

Parametri
1) ds_sem: descrittore del semaforo su cui si vuole operare
2) sem_num: indice dell'elemento del semaforo su cui si vuole operare
3) cmd: specifica del comando da eseguire (IPC_RMID, IPC_STAT, IPC_SET, GETALL, SETALL, GETVAL, SETVAL)
4) arg: puntatore al buffer con eventuali parametri per il comando

Descrizione -1 in caso di fallimento

```
union semuu {
        int     val;           /* usato se cmd == SETVAL  */
        struct semid_ds *buf   /* usato per IPC_STAT e IPC_SET */
        ushort *array;         /* usato se cmd == GETALL o SETALL */
  };
```

Operazioni su un semaforo

int semop(int ds_sem, struct sembuf oper[], int number)

Descrizione invoca l'esecuzione di un comando su una coda di messaggi

---

Parametri
1) ds_sem: descrittore del semaforo su cui si vuole operare
2) oper: indirizzo dell'array contenente la specifica delle operazioni da eseguire
3) number: numero di argomenti validi nell'array puntato da oper

---

Descrizione -1 in caso di fallimento

```
struct sembuf {
        ushort sem_num;
        short  sem_op;  /* 0=sincronzzazione sullo 0 – n=incremento
                           di n - -n=decremento di n */
        short  sem_flg; /* IPC_NOWAIT – SEM_UNDO */
    };
```

Un decremento di n su un semaforo dal valore minore di n provoca blocco del processo chiamante a meno della specifica di IPC_NOWAIT

SEM_UNDO revoca l'operazione in caso di exit del processo

Per poter annullare operazioni semaforiche, il sistema operativo mantiene una struttura **sem_undo** in cui sono registrate tali operazioni per ogni processo

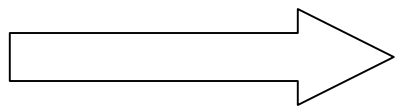
Il valore di tale struttura non viene ereditato da in processo figlio generato tramite fork()

Il valore della struttura viene mantenuto in caso di sostituzione di codice tramite exec()

Trasferimento di stringhe con sincronizzazione esplicita

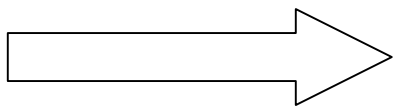
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#define DISP_ 20
#define Errore_(x) { puts(x); exit(1); }

char messaggio[256];

void produttore(int id, int  sem_id) {
        char *p; int ret; struct sembuf oper;

        p = shmat(id, 0 , SHM_W);
        if ( p == (char*) -1 ) Errore_("Errore nella chiamata shmat");
        puts("Digitare le parole da trasferire (quit per terminare):");
        do { scanf("%s", messaggio);
             strncpy(p, messaggio, DISP_);
             p += DISP_;
        } while( (strcmp(messaggio,"quit") != 0));
        oper.sem_num =  0;
        oper.sem_op  = -1;
        oper.sem_flg =  0;
        ret = semop(sem_id, &oper, 1);
        if ( ret == -1 ) Errore_("Errore nella chiamata semop");
        exit(0);
}
```

continua



```
void consumatore(int id, int  sem_id) {
       char *p;
       int ret;
       struct sembuf oper;

       p = shmat(id, 0 , SHM_R);
       if ( p == (char*) -1 ) Errore_("Errore nella chiamata shmat");

       oper.sem_num = 0;
       oper.sem_op  = 0;
       oper.sem_flg = 0;

       ret = semop(sem_id, &oper, 1);
       if ( ret == -1 ) Errore_("Errore nella chiamata semop");

       puts("Contenuto memoria condivisa:");
       while( (strcmp(p,"quit") != 0)){
           printf("%s\n", p);
           p += DISP_;
       }

       exit(0);
   }
```

continua



```
int main(int argc, char *argv[]) {
        long id_shm, id_sem;
        int  ret, status;
        long chiave_shm=30, chiave_sem = 50;

        id_shm = shmget(chiave_shm, 1024, IPC_CREAT|0666);
        if ( id_shm == -1 ) Errore_("Errore nella chiamata shmget");

        id_sem = semget(chiave_sem, 1, IPC_CREAT|IPC_EXCL|0666);
        if ( id_sem == -1 ) Errore_("Errore nella chiamata semget");

        ret = semctl(id_sem, 0, SETVAL, 1);
        if ( ret == -1 ) Errore_("Errore nella chiamata semctl");

        if ( fork()!=0 ){
            if ( fork()!=0 ){
                wait(&STATUS);
                wait(&STATUS);
            }
            else consumatore(id_shm, id_sem);
        }
        else produttore(id_shm, id_sem);

        ret = shmctl(id_shm, IPC_RMID, NULL);
        if ( ret == -1 ) Errore_("Errore nella chiamata shmctl");

        ret = semctl(id_sem, 0, IPC_RMID , 1);
        if ( ret == -1 ) Errore_("Errore nella chiamata semctl");
  }
```

Mutex NT/2000

Sono in pratica dei semafori binari, ovvero dei semplici meccanismi per la mutua esclusione

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttributes,
                   BOOL bInitialOwner,
                   LPCTSTR lpName)
```

Descrizione

• invoca la creazione di un mutex

Restituzione

• handle al mutex in caso di successo, NULL in caso di fallimento

Parametri

• lpMutexAttributes: puntatore a una struttura SECURITY_ATTRIBUTES
• bInitialOwner: indicazione del processo chiamante come possessore del mutex
• lpName: nome del mutex

Apertura di un mutex

```
HANDLE OpenMutex(DWORD dwDesiredAccess,
                 BOOL bInheritHandle,
                 LPCTSTR lpName)
```

Descrizione

• invoca l'apertura di un mutex

Restituzione

• handle al mutex in caso di successo, NULL in caso di fallimento

Parametri

• dwDesiredAccess: accessi richiesti al mutex
• bnheritHandle: specifica se l'handle e' ereditabile
• lpName: nome del mutex

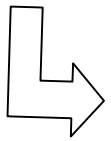
Operazioni su un mutex

Accesso al mutex

```
DWORD WaitForSingleObject(HANDLE hHandle,
                          DWORD dwMilliseconds)
```

Rilascio del mutex

```
BOOL ReleaseMutex(HANDLE hMutex)
```



0 in caso di fallimento

Un solo processo (o thread) viene eventualmente risvegliato per effetto del rilascio del mutex

Semafori NT/2000

```
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES
                         lpSemaphoreAttributes,
                       LONG lInitialCount,
                       LONG lMaximumCount,
                       LPCTSTR lpNamE)
```

Descrizione

• invoca la creazione di un semaforo

Restituzione

• handle al semaforo in caso di successo, NULL in caso di fallimento

Parametri

• lpSemaphoreAttributes: puntatore a una struttura SECURITY_ATTRIBUTES
• lInitialCount: valore iniziale del semaforo
• lMaximumCount: massimo valore che il semaforo puo' assumere
• lpName: nome del semaforo

Apertura ed operazioni su un semaforo

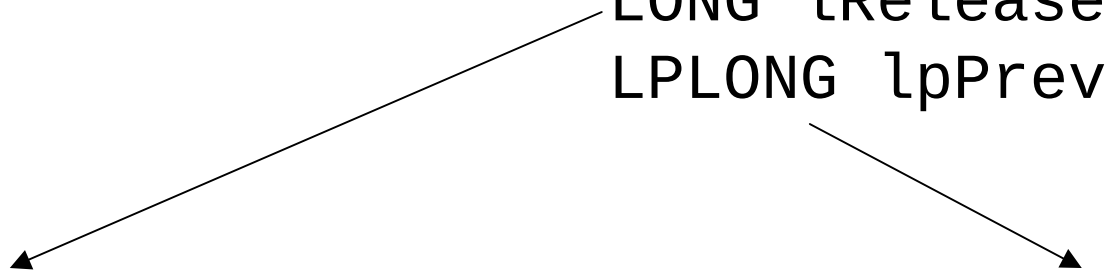
```
HANDLE OpenSemaphore(LDWORD dwDesiredAccess,
                     BOOL bInheritHandle,
                     LPCTSTR lpName)
```

Accesso al semaforo

```
DWORD WaitForSingleObject(HANDLE hHandle,
                          DWORD dwMilliseconds)
```

Rilascio del semaforo

```
BOOL ReleaseSemaphore(HANDLE hSemaphore,
                      LONG lReleaseCount,
                      LPLONG lpPreviousCount)
```



Unita' di rilascio

Puntatore all'area di memoria dove scrivere il vecchio valore del semaforo