

Sistemi Operativi

Laurea in Ingegneria Informatica

Università di Roma Tor Vergata

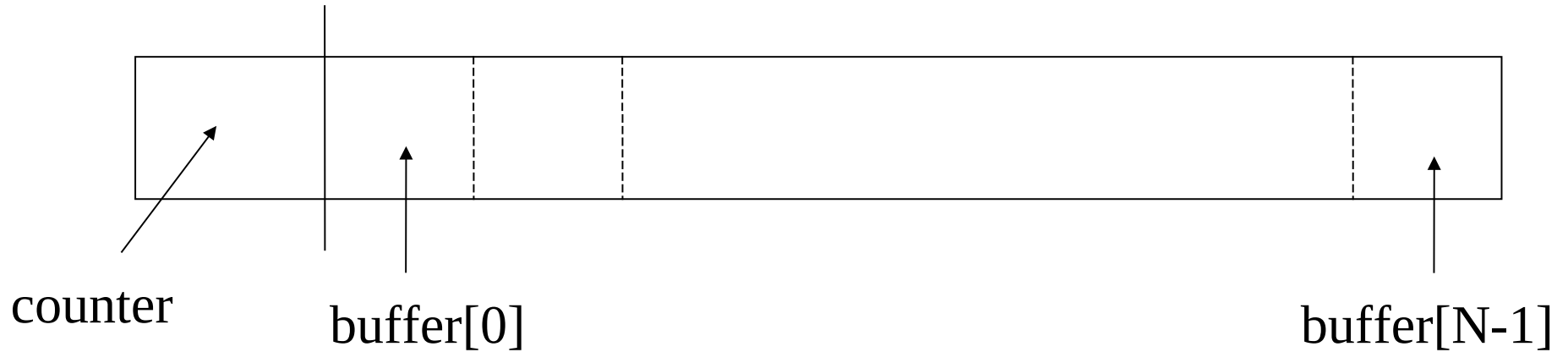
Docente: Francesco Quaglia



Sincronizzazione

1. Sezioni critiche
2. Approcci per la sincronizzazione
3. Spinlock, mutex e semafori
4. Supporti per la sincronizzazione in sistemi UNIX/Windows

L'archetipo del produttore/consumatore



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];

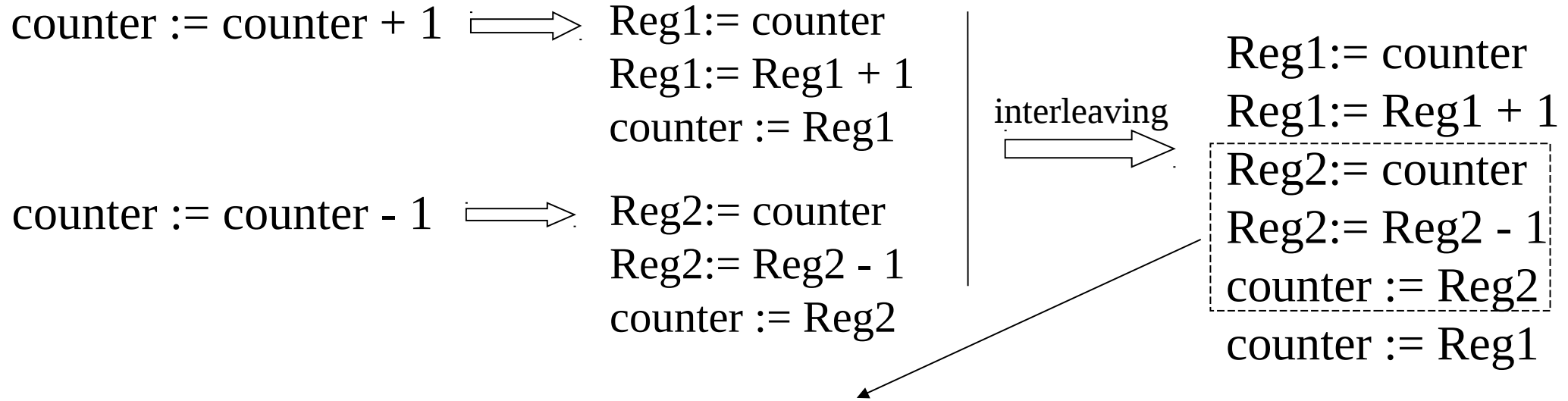
out := (out+1)mod(N)

counter := counter - 1;

<consume Y>

until false

Sezioni critiche



Una sezione critica è una porzione di traccia ove

- un processo/thread può leggere/scrivere dati condivisi con altri processi/thread
- la correttezza del risultato dipende dall'interleaving delle tracce di esecuzione

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 thread accede alla sezione critica nessun altro thread può eseguire nella stessa sezione critica

Progresso

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

Attesa limitata

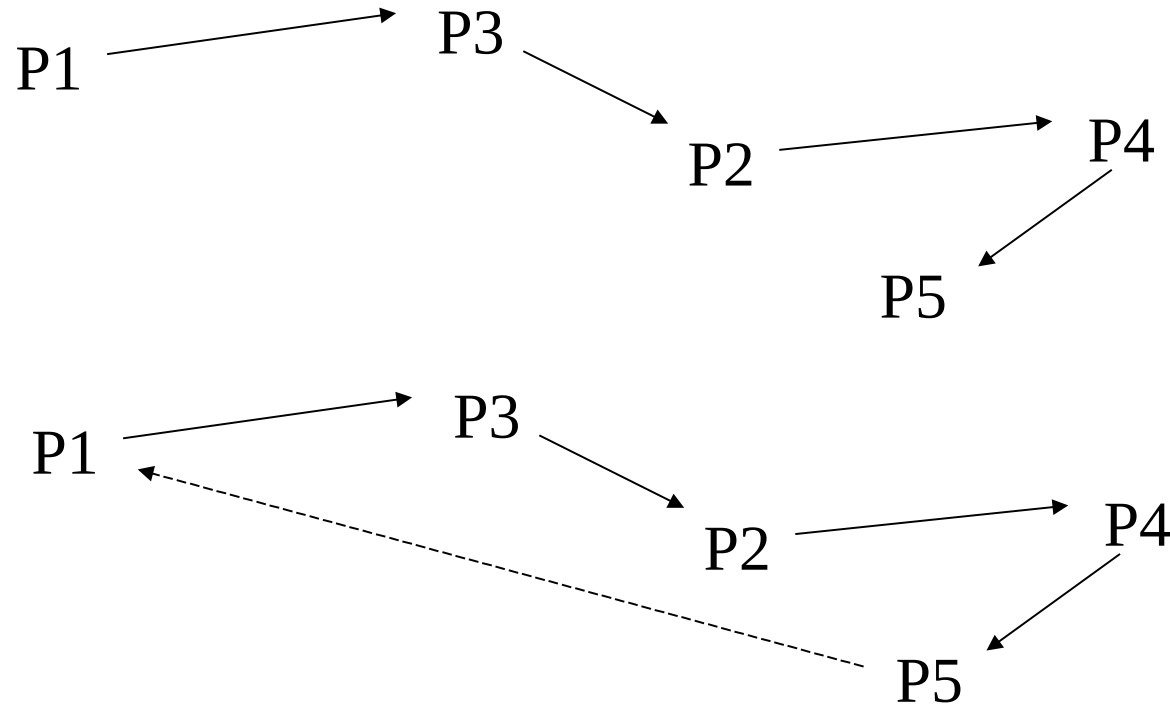
- un thread che lo chiede deve essere ammesso ad eseguire la sezione critica in un tempo limitato (non ci devono essere **stalli** o **starvation**)
-

Approcci risolutivi

- algoritmi di mutua esclusione
- approcci hardware e istruzioni Read-Modify-Write (RMW)
- mutex/semaphori

Stallo (deadlock)

Un insieme di processi P_1, \dots, P_n è coinvolto in uno stallo se ognuno dei processi è in attesa (attiva o passiva) di un evento che può essere causato solo da un altro dei processi dell'insieme



Genesi di un deadlock

Algoritmi di mutua esclusione - Dekker

var turno: **int**;

Processo X

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

Processo Y

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

I processi vanno in alternanza stretta nella sezione critica

- non c'è garanzia di progresso
- la velocità di esecuzione è limitata dal processo più 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'è garanzia di progresso
- non c'è garanzia di mutua esclusione (problema che diviene evidente nel caso di numero elevato di processi)

Terzo tentativo

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

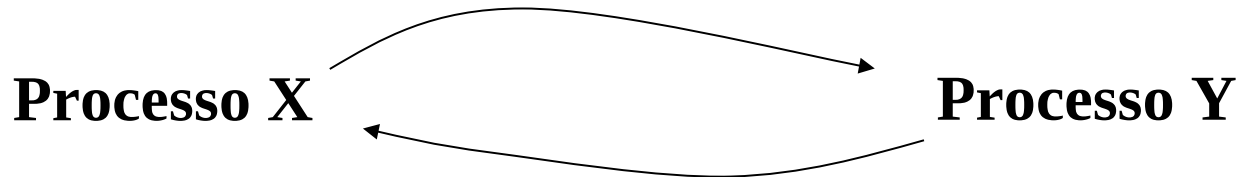
Processo X

```
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;
```

Possibilità di deadlock, non c'è 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;
```

Possibilità di starvation, non c'è garanzia di attesa limitata

Algoritmo del fornaio (Lamport - 1974)

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**;

}

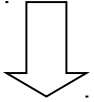
<sezione critica>;

number[i] := 0;

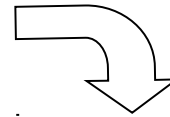
}until FALSE

Approcci hardware ed istruzioni RMW (Read-Modify-Write)

- 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;
```

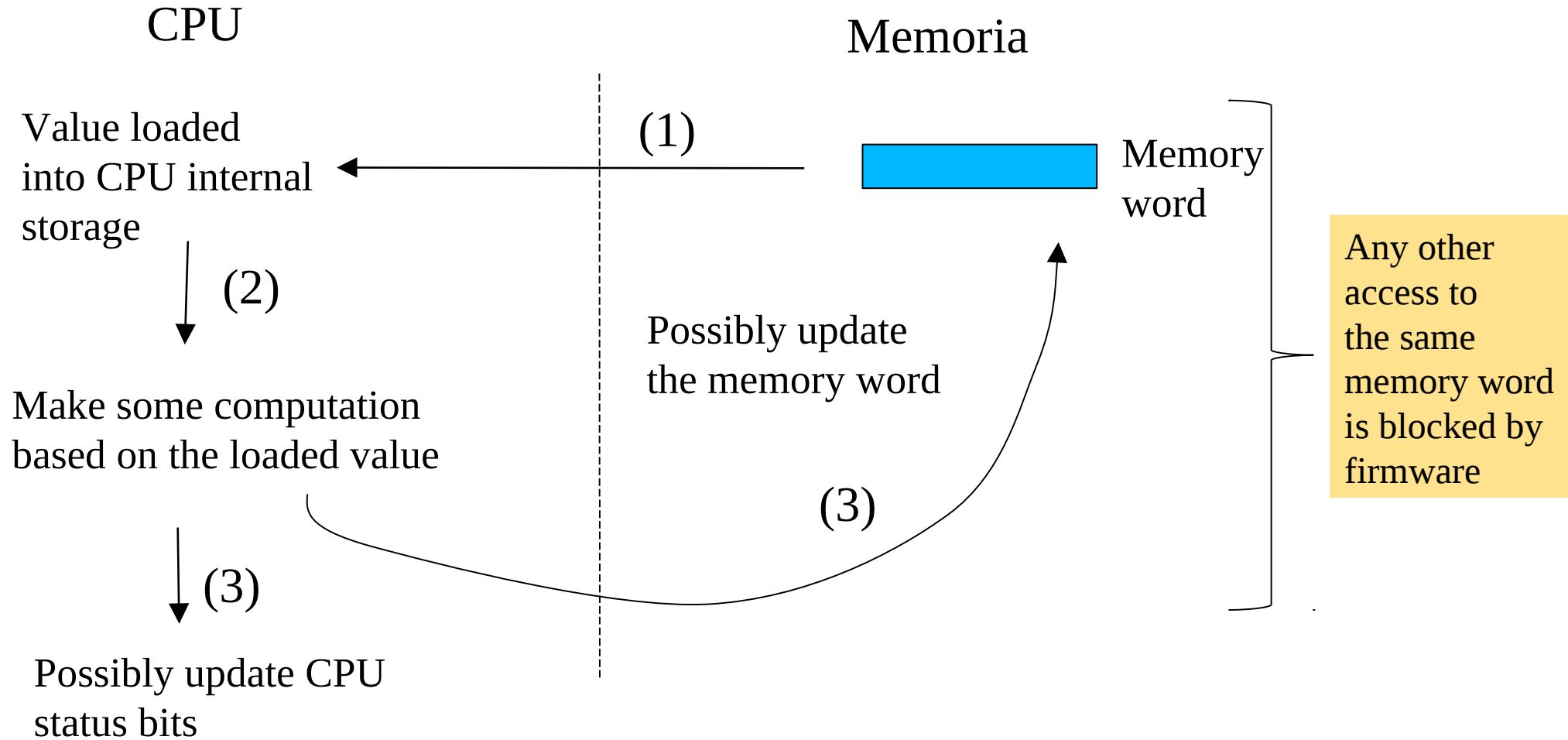
Processo X

```
While !test_and_set(serratura) do no-op;
```

```
<sezione critica>;
```

```
serratura := 0
```

Timeline di una istruzione RMW



Compare-and-Swap

- Test and set atomico è correntemente supportato tramite specifiche istruzioni macchina RMW
- L'istruzione macchina più comunemente supportata è la CAS (Compare And Swap)
- Su architetture x86 tale istruzione corrisponde a CMPXCHG (Compare And Exchange)
- CMPXCHG compara il valore di una locazione di memoria con quello del registro RAX, se uguali la locazione è aggiornata col valore di un altro registro esplicitamente indicato come operando, e.g. RBX

Un esempio di utilizzo

```
int try_lock(void * uadr){
    unsigned long r = 0 ;
    asm volatile(
        "xor %%rax,%%rax\n"
        "mov $1,%%rbx\n"
        "lock cmpxchg %%ebx, (%1)\n"
        "sete (%0)\n"
        :↑ : "r"(&r), "r" (uadr)
        : "%rax", "%rbx"
    );
    return (r) ? 1 : 0;
}
```

Target memory word

Set equal

If they were equal return 1

Pthread spinlock

Tipi e API per la programmazione

✓ `pthread_spinlock_t lock;`

✓ `int pthread_spin_init(pthread_spinlock_t *lock, int pshared);` ←

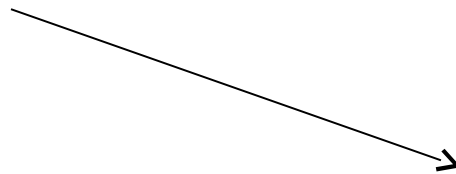
✓ `pthread_spin_lock(&lock);`

✓ `pthread_spin_unlock(&lock);`

PTHREAD_PROCESS_SHARED
PTHREAD_PROCESS_PRIVATE

WinAPI

- L'utilizzo delle istruzioni RMW è tipicamente incapsulato (come del resto per Posix) all'interno di funzioni di semplice uso per il programmatore
- Queste si dicono “interlocked”, locuzione che appare anche nel nome stesso delle funzioni
- Classici esempi sono:
 - ✓ InterlockedCompareExchange
 - ✓ InterlockedBitTestAndSet



Syntax

C++

```
UCHAR WINAPI InterlockedBitTestAndSet(  
    _In_ LONG volatile *Base,  
    _In_ LONG          Bit  
);
```

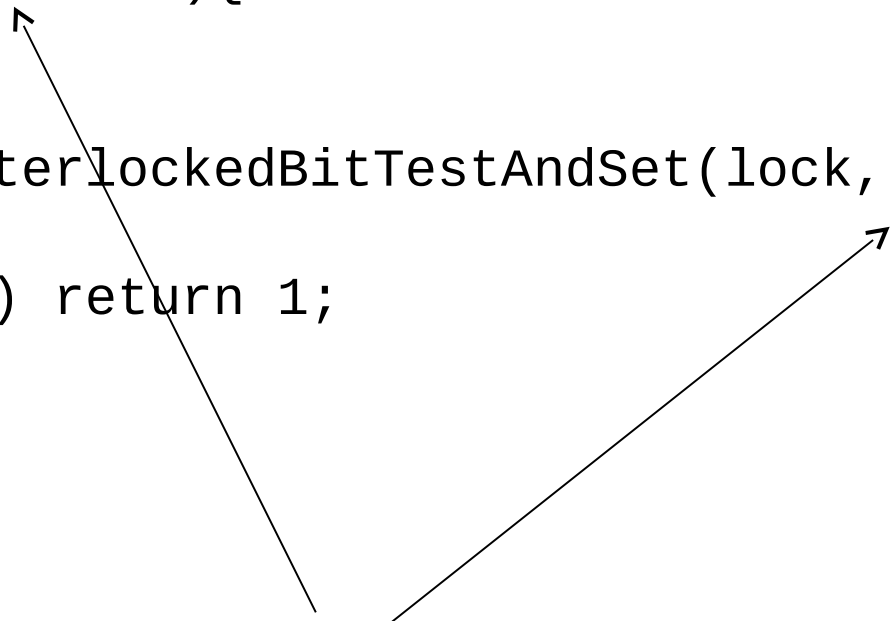

Un semplice try-lock basato su API interlocked

```
int try_lock(LONG * lock){
    int ret;

    ret =(int)InterlockedBitTestAndSet(lock, 0);

    if (ret == 0) return 1;

    return 0;
}
```

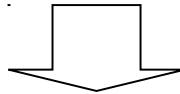


Il 0-esimo bit della locazione di memoria puntata dal parametro 'lock' rappresenta il lock

Semafori - modello di riferimento

Un semaforo è una struttura dati che include un valore intero non negativo con 3 operazioni associate

- inizializzazione
- una operazione **wait** che tenta il decremento di una unità ed induce una attesa sul processo che la esegue in caso il valore del semaforo dovesse diventare negativo per effetto del decremento
- una operazione **signal** che ne incrementa il valore di una unità e libera dall'attesa in attesa un processo che ha eseguito una operazione di wait bloccante



Le operazioni sono eseguite in modo atomico dal kernel del sistema operativo

Semafori binari

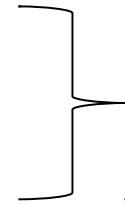
- il valore assunto può essere solo 1 oppure 0

Implementazioni

```
type semaphore = record //struct like
    value: int;
    L: list of processes;// or threads
end;
```

```
procedure Wait(var s: semaphore):
    if (s.value - 1) < 0 {
        add current process to s.L;
        block current process;
    }else{s.value--;}

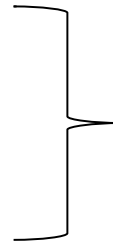
```



Atomically executed by the
kernel software

```
procedure Signal(var s: semaphore):
    s.value = s.value + 1;
    if s.L not empty{
        delete a process P from s.L;
        unblock P;
        s.value = s.value - 1;
    }

```



Atomically executed by the
kernel software

Produttore consumatore tramite semafori - pseudo-codice

SHARED: item buffer[N]; semaphore S = 1; int counter = 0;

PRODUTTORE

PRIVATE int in = 0; item X;

Repeat

```
    <produce X>
retry:
    wait(S);
    if counter = N {
        signal(S);
        goto retry;
    }
    else{
        buffer[in] := X;
        in := (in+1)mod(N)
        counter := counter + 1;
        signal(S);
    }
until false
```

CONSUMATORE

PRIVATE int out = 0; item Y;

Repeat

```
    wait(S);
    if counter = 0 {
        signal(S);
    }
    else{
        Y := buffer[out];
        out := (out+1)mod(N)
        counter := counter - 1;
        signal(S);
        <consume Y>
    }
until false
```

Semafori in sistemi UNIX (System V – Posix traditional)

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

Descrizione invoca la creazione di un semaforo

Parametri

- 1) key: chiave per identificare il semaforo in maniera univoca nel sistema (IPC_PRIVATE è un caso speciale)
- 2) size: numero di elementi del semaforo
- 3) flag: specifica della modalità 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 una struttura unica valida per qualsiasi processo

Comandi su un semaforo

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

Descrizione invoca l'esecuzione di un comando su un semaforo

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

Ritorno -1 in caso di fallimento

```
union semun {  
    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 semaforiche

```
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

Ritorno -1 in caso di fallimento

```
struct sembuf {  
    ushort sem_num;  
    short sem_op; /* 0=sincronizzazione 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

Creazioni di processi ed operazioni semaforiche

- 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 execXX()

Posix - librerie semaforiche

POSIX named/unnamed semaphores

- ✓ `sem_t sem_name;`
- ✓ `sem_t *sem_open(const char *name, int oflag)`
- ✓ `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- ✓ `sem_wait(&sem_name);`
- ✓ `sem_post(&sem_name);`
- ✓ `sem_getvalue(sem_t *sem, int *valp);`
- ✓ `sem_unlink(const char *name);`

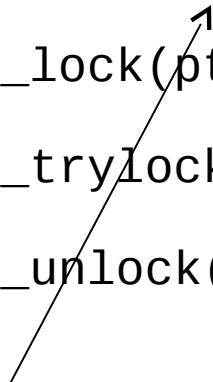
Può inizializzare anche
semafori unnamed

Implementazione basata su pseudo files

I semafori named sono di fatto oggetti in `/dev/shm`

Posix - librerie semaforiche

POSIX pthread mutexes

- ✓ `pthread_mutex_t mutex;`
 - ✓ `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`
 - ✓ `pthread_mutex_lock(pthread_mutex_t *mutex)`
 - ✓ `pthread_mutex_trylock(pthread_mutex_t *mutex)`
 - ✓ `pthread_mutex_unlock(pthread_mutex_t *mutex)`
- 

Idealmente può essere usato ricorsivamente ma non tutte le implementazioni sono conformi

Windows mutex

Sono in pratica simili a semafori binari, ovvero a 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
- bInheritHandle: specifica se l'handle è ereditabile
- lpName: nome del mutex

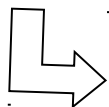
Operazioni su un mutex

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 Windows

```
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 e 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)
```

Unità di rilascio

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

Sincronizzazione su oggetti multipli

Array di handles verso gli oggetti target

Numero di elementi dell'array
di handles

Regola di attesa (tutti o almeno uno)

Syntax

C++

```
DWORD WINAPI WaitForMultipleObjects(  
    _In_   DWORD nCount,  
    _In_   const HANDLE *lpHandles,  
    _In_   BOOL bWaitAll,  
    _In_   DWORD dwMilliseconds  
);
```