Condizioni per la *mutua esclusione*

regione critica: le righe di codice che accedono alla risorsa condivisa

Quattro condizioni

1. non più di un thread all'interno della regione
2. nessuna assunzione sulle velocità dei processi
3. nessun processo fuori della sezione critica può impedire ad un altro di entrare
4. accesso alla sezione critica consentito entro un tempo finito

Soluzione: disabilitare gli interrupt

può essere fatto solo in kernel mode

solo per brevissimi periodi!

e se ho più di una CPU?

$\Rightarrow$ solo nel kernel, solo per poche istruzioni

Non-soluzione con variabile di lock

```
int g_lock = 0; /* variabile globale */
...
/* codice eseguito da tutti i thread: */
while(1) {
  DoSomeWork();
  while (g_lock != 0)
    ;   /* busy wait */
  g_lock = 1;
  CriticalSection();
  g_lock = 0;
}
```

Purtroppo, NON funziona!

Quasi soluzione: accesso alternato

```
/* var globale */
int g_turn = 0;

/* codice del thread n. 0 */
  while(1) {
    DoSomeWork();
    while (g_turn != 0)
      ;  // busy wait
    EnterCriticalSection();
    g_turn = 1;
  }
```

```
/* codice del thread n. 1 */
  while(1) {
    DoSomeWork();
    while (g_turn != 1)
      ;  // busy wait
    EnterCriticalSection();
    g_turn = 0;
  }
```

Viola la condizione 3, quindi *non* è una buona soluzione!

Una soluzione hardware

Istruzione TSL reg,addr (Test and Set Lock)

- copia il contenuto della cella addr nel registro reg
- mette nella cella addr il valore 1
- tutto ciò in maniera atomica

```
loop: TSL rx, g_lock
      CMP rx, #1
      JEQ loop
... critical region ...
      MOV g_lock, #0
```

```
loop: rx := g_lock; g_lock := 1;
      if rx == 1
      then goto loop;
... critical region ...
      g_lock := 0;
```

Algoritmo di Peterson (1981)

Combina l'idea dell'accesso alternato e della variabile di lock

Completamente in software

Si basa su due var globali

```
int g_turn;             // di chi è il turno di aspettare?
int g_interested[2];    // chi è interessato ad entrare?
```

Algoritmo di Peterson, cont.

```
/* codice eseguito da tutti i thread: */
void thread(int n) {
  while(1) {
    DoSomeWork();
    PetersonEnterCS(n);
    // ... critical section ...
    PetersonLeaveCS(n);
  }
}
```

Algoritmo di Peterson, cont.

```
int g_turn = 0;                     // di chi è il turno?
int g_interested[2] = {0, 0};       // chi è interessato?

void PetersonEnterCS(int nThis) {   // thread che vuole entrare
  int nOther = 1 - nThread;         // numero dell'altro thread
  g_interested[nThis] = TRUE;       // mostra che sei interessato
  g_turn = nThis;                   // cambia il flag del turno

  while (g_turn == nThis && g_interested[nOther] == TRUE)
    ;                               // busy wait
}

void PetersonLeaveCS(int nThis) {   // thread che vuole uscire
  g_interested[nThis] = FALSE;      // stiamo uscendo
}
```

Priority inversion

Thread A, ad alta priorità, esegue (poniamo) Peterson

Thread B, a bassa priorità, è nella sezione critica

A esegue il busy wait e non cede mai la CPU a B

B non riesce ad uscire dalla sezione critica

$\Rightarrow$ A è bloccato!

Il busy wait è cattivo

Peterson e TSL usano busy wait

- Il busy wait sciupa CPU
- Priority inversion
- Preferire l'uso di primitive fornite dal kernel
- Per es.: *mutex*
- Per es.: *sleep* e *wakeup*

Possibile miglioramento: chiamare thread_yield(3) o sleep(2) all'interno del loop

Il *Mutex*

garantisce la mutua esclusione

implementato in user space (usando TSL o Peterson)

oppure implementato nel kernel

operazioni:

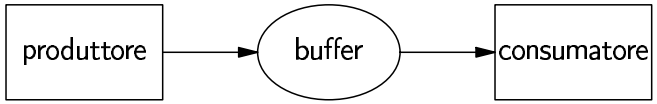
- create_mutex
- lock
- unlock
- destroy_mutex

Mutua esclusione con mutex

```
mutex m;

main() {
  m = create_mutex();
  create_thread(f);
  create_thread(f);
  ...
}
```

```
void f() {
  while(1) {
    DoSomeWork();
    lock(m);
    // ... critical section ...
    unlock(m);
  }
}
```

Problema dei *Produttori e Consumatori*



buffer di dimensione limitata

il produttore deve attendere se il buffer è pieno

il consumatore deve attendere se il buffer è vuoto

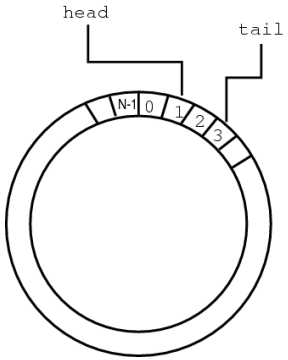
Produttori e Consumatori, cont.

```
void producer() {
  while (1) {
    item = produce_item();
    insert_item(item);

  }
}
```

```
void consumer() {
  while (1) {

    item = remove_item();
    consume_item(item);
  }
}
```

Il buffer circolare

```
ITEM buf[N+1];
int head = 0;
int tail = 0;
```



- gli elementi "pieni" vanno da head a tail-1
- il buffer è vuoto se tail==head
- il buffer è pieno se tail+1==head
- può contenere N elementi

Il buffer circolare, cont.

```
ITEM buf[N+1];
int head = 0;
int tail = 0;
ITEM remove_item() {
  ITEM it;
  if (tail == head) error(buffer vuoto);
  it = buf[head];
  head = (head+1) % (N+1);
  return it;
}
void enter_item(ITEM it) {
  if ((tail+1) % (N+1) == head) error(buffer pieno);
  buf[tail] = it;
  tail = (tail+1) % (N+1);
}
```

Primo passo: aggiungere la mutua esclusione

```
void producer() {
  while (1) {
    item = produce_item();
    lock(m);
    insert_item(item);
    unlock(m);

  }
}
```

```
void consumer() {
  while (1) {

    lock(m);
    item = remove_item();
    unlock(m);
    consume_item(item);
  }
}
```

Secondo passo: bloccare i processi

```
void producer() {
 while (1) {
  while (N == count) /* wait */;
  item = produce_item();
  lock(m);
  insert_item(item);
  count++;
  unlock(m);

 }
}
```

```
void consumer() {
 while (1) {
  while (0 == count) /* wait */

   lock(m);
   item = remove_item();
   count--;
   unlock(m);
   consume_item(item);
 }
}
```

E se provassimo senza busy wait?

```
void producer() {
  while (1) {
    if (N == count) sleep();
    item = produce_item();
    lock(m);
    insert_item(item);
    count++;
    if (1 == count) wakeup(consumer);
    unlock(m);

  }
}
```

```
void consumer() {
  while (1) {
    if (0 == count) sleep();

    lock(m);
    item = remove_item();
    count--;
    if (N-1 == count) wakeup(produce
    unlock(m);
    consume_item(item);
  }
}
```

Problema: *lost wakeup*

Se il wakeup viene eseguito *prima* che l'altro processo esegua *sleep*?

Soluzione di Java, e di Posix Threads

- sleep si può eseguire solo se ho il lock di un mutex
- eseguendo sleep perdo il lock

Portiamo la sleep dentro la CS (Java, Posix Threads)

```
void producer() {
  while (1) {
    item = produce_item();
    lock(m);
    if (N == count) sleep();
    insert_item(item);
    count++;
    if (1 == count) wakeup(consumer);
    unlock(m);

  }
}
```

```
void consumer() {
  while (1) {

    lock(m);
    if (0 == count) sleep();
    item = remove_item();
    count--;
    if (N-1 == count) wakeup(prod
    unlock(m);
    consume_item(item);
  }
}
```

E se ci sono più di due thread?

```
void producer() {
  while (1) {
    item = produce_item();
    lock(m);
    while (N == count) sleep();
    insert_item(item);
    count++;
    if (1 == count) wakeup(consumer);
    unlock(m);

  }
}
```

```
void consumer() {
  while (1) {

    lock(m);
    while (0 == count) sleep();
    item = remove_item();
    count--;
    if (N-1 == count) wakeup(produce
    unlock(m);
    consume_item(item);
  }
}
```

Edsger Wybe Dijkstra, (1930–2002)

Semafori

- Dijkstra, 1965
- Due operazioni: down e up *atomiche per definizione*
- Down:
  - se sem $>$ 0 allora
  - decrementa sem
  - altrimenti
  - sospendi il thread corrente
- Up:
  - se ci sono thread in attesa sul semaforo
  - svegliane uno
  - altrimenti
  - incrementa sem

Mutua esclusione con semafori

```
/* il semaforo è una variabile globale */
semaphore sem = 1;

/* codice eseguito da ciascun thread: */
  while (1) {
    DoSomeWork();
    DOWN(sem);
    EnterCriticalSection();
    UP(sem);
  }
```

Produttori-consumatori con semafori

```
semaphore mutex = 1;\semaphore free = N;\semaphore busy =
0;\
void producer() {
  while (1) {
    item = produce_item();
    down(free);
    down(mutex);
    insert_item(item);
    up(mutex);
    up(busy);

  }
}
```

```
void consumer() {
  while (1) {

    down(busy);
    down(mutex);
    item = remove_item();
    up(mutex);
    up(free);
    consume_item(item);
  }
}
```

Strutture dati thread-safe

Una struttura dati è thread safe solo se:

- l'accesso è consentito solo tramite apposite procedure
- l'accesso di più thread concorrenti non crea problemi

Meglio nascondere il codice di sincronizzazione nelle procedure di accesso ai dati, piuttosto che nel codice dei thread come fa Tanenbaum

```
ITEM buf[N+1];
int head = 0;
int tail = 0;
semaphore mutex = 1;
semaphore free = N;
semaphore busy = 0;

void enter_item(ITEM it) {
  down(free);
  down(mutex);
  buf[tail] = it;
  tail = (tail+1) % (N+1);
  up(mutex);
  up(busy);
}
```

```
ITEM remove_item() {
  ITEM it;
  down(busy);
  down(mutex);
  it = buf[head];
  head = (head+1) % (N+1);
  up(mutex);
  up(free);
  return it;
}
```