Sincronizzazione e comunicazione fra processi/thread

Per poter realizzare applicazioni costituite da più processi o thread cooperanti occorrono meccanismi per:

- Attivazione e terminazione di processi/thread
- Sincronizzazione: P1 per andare avanti attende che P2 sia arrivato ad un dato punto – utile *fra l'altro (ma non solo)* per disciplinare l'accesso a dati condivisi
- Comunicazione: P2 passa dati a P1 – che tipicamente attende

Anche nella sincronizzazione "semplice" c'è almeno il passaggio dell'informazione (implicita) che P2 è arrivato al punto desiderato

I meccanismi si possono trovare in:

- linguaggi concorrenti (cioè dotati di istruzioni apposite per attivare e far sincronizzare/comunicare più processi/threads)
- *system calls* messe a disposizione dal sistema operativo
- librerie di funzioni per lo sviluppo di applicazioni concorrenti

Processi concorrenti e cooperanti: modelli di interazione

- Modello a memoria condivisa: i processi (o thread) condividono almeno alcune variabili in memoria. Se un processo/thread modifica una di queste variabili, tutti gli altri processi/thread vedranno il nuovo valore. La condivisione è naturale per threads, per i processi è stata introdotta con meccanismi appositi.
- Modello a scambio di messaggi: i processi non condividono aree di memoria, ma possono inviarsi messaggi utilizzando istruzioni *send* e *receive*. Lo scambio di informazioni è sempre esplicito.

Ci occuperemo prevalentemente del MODELLO A MEMORIA CONDIVISA

Processi concorrenti e cooperanti

I meccanismi di interazione devono permettere

- lo scambio di informazioni
- il corretto ordinamento di azioni compiute da diverse entità
- l'accesso controllato a risorse condivise

Si possono dunque avere interazioni di tipo *cooperativo* o *competitivo* che utilizzano le stesse operazioni messe a disposizione

Affinché l'interazione avvenga in modo corretto i processi devono rispettare alcune regole (per esempio fare richiesta esplicita di una risorsa condivisa ed eventualmente attendere il proprio turno per l'uso): se le regole non vengono rispettate si possono verificare *interferenze* (malfunzionamenti)

Sincronizzazione e memoria condivisa: le corse critiche

```
VersaSulConto(int numconto,int versamento)
{ Saldo = CC[numconto];

  Saldo = Saldo + versamento;

  CC[numconto] = Saldo; }
```

Supponiamo che più persone possano effettuare versamenti sullo stesso conto *contemporaneamente*: ovvero possono esserci più processi che eseguono *contemporaneamente* la procedura VersaSulConto.

I processi/threads *condividono* il vettore CC[ ] che contiene il saldo di tutti i conti correnti.
Invece Saldo è locale alla procedura, potrebbe essere una variabile introdotta dal compilatore per tradurre l'istruzione

CC[numconto] = CC[numconto] + versamento

Le corse critiche (race conditions)

P1 | CC[1200] | P2

CC[1200]: 2.000

1. Saldo = CC[1200];

P1: Saldo = 2.000

2. Saldo = CC[1200];

P2: Saldo = 2.000

4. Saldo = Saldo + 200;

P1: Saldo = 2.200

3. Saldo = Saldo + 350;

P2: Saldo = 2.350

5. CC[1200] = Saldo;

P1: CC[1200] = 2.200

CC[1200]: 2.200

6. CC[1200] = Saldo;

P2: CC[1200] = 2.350

CC[1200]: 2.350

ERRORE !!!!!

NOTA: essendo Saldo LOCALE, è diversa nei due thread (ognuno ha il suo stack dove si trovano le variabili locali). CC[ ] è condiviso.

Le corse critiche (race conditions)

P1 CC[1200] P2

2.000

1. Saldo = CC[1200];

P1: Saldo = 2.000

4. Saldo = CC[1200];

P2: Saldo = 2.200

2. Saldo = Saldo + 200;

P1: Saldo = 2.200

5. Saldo = Saldo + 350;

P2: Saldo = 2.550

3. CC[1200] = Saldo;

P1: CC[1200] = 2.200

6. CC[1200] = Saldo;

2.200

P2: CC[1200] = 2.550

2.550

CORRETTO !!

come garantire che avvengano solo esecuzioni come questa, in cui la sequenza di un processo/thread non viene interrotta dall'altra?

Le corse critiche (race conditions)

Il problema si ha anche semplicemente quando si aggiorna una variabile condivisa: se l'aggiornamento non viene eseguito con una unica istruzione di linguaggio macchina (che non viene intercalata con altre operazioni... *almeno in un sistema uniprocessore*), ma richiede un passaggio intermedio attraverso un registro, allora si può presentare esattamente lo stesso problema

Es. Se a++ in C viene tradotto in linguaggio macchina sotto forma di:

```
MOVE R1 a  // copia la variabile a nel registro R1
INC  R1 1  // incrementa il registro
MOVE a R1  // copia il nuovo valore dal registro in a
```

se due processi/thread che condividono a eseguono a++ può accadere che solo uno dei due aggiornamenti venga effettivamente portato a termine e salvato in *a*, mentre l'altro viene perso.

Le corse critiche (race conditions)

Altro esempio, si riempie un array a condiviso utilizzando una variabile i condivisa come indice:

(i == 5)

```
a[i] = 10;          a[i] = 20;
i++;                i++;
```

Vorremmo presumibilmente ottenere: i==7 e a[5]==10,a[6]==20 oppure viceversa

Ma se anche le istruzioni C fossero «atomiche» (indivisibili), le operazioni potrebbero avvenire ad esempio nell'ordine:

```
a[i] = 10;
a[i] = 20;
i++;
i++;
```

Il valore 20 viene scritto sopra 10 in a[5], mentre in a[6] rimane il valore che c'era prima

Le corse critiche (race conditions)

La velocità relativa con cui avanza l'esecuzione dei processi/thread non è prevedibile (*conseguenza:* una esecuzione non è facilmente riproducibile)

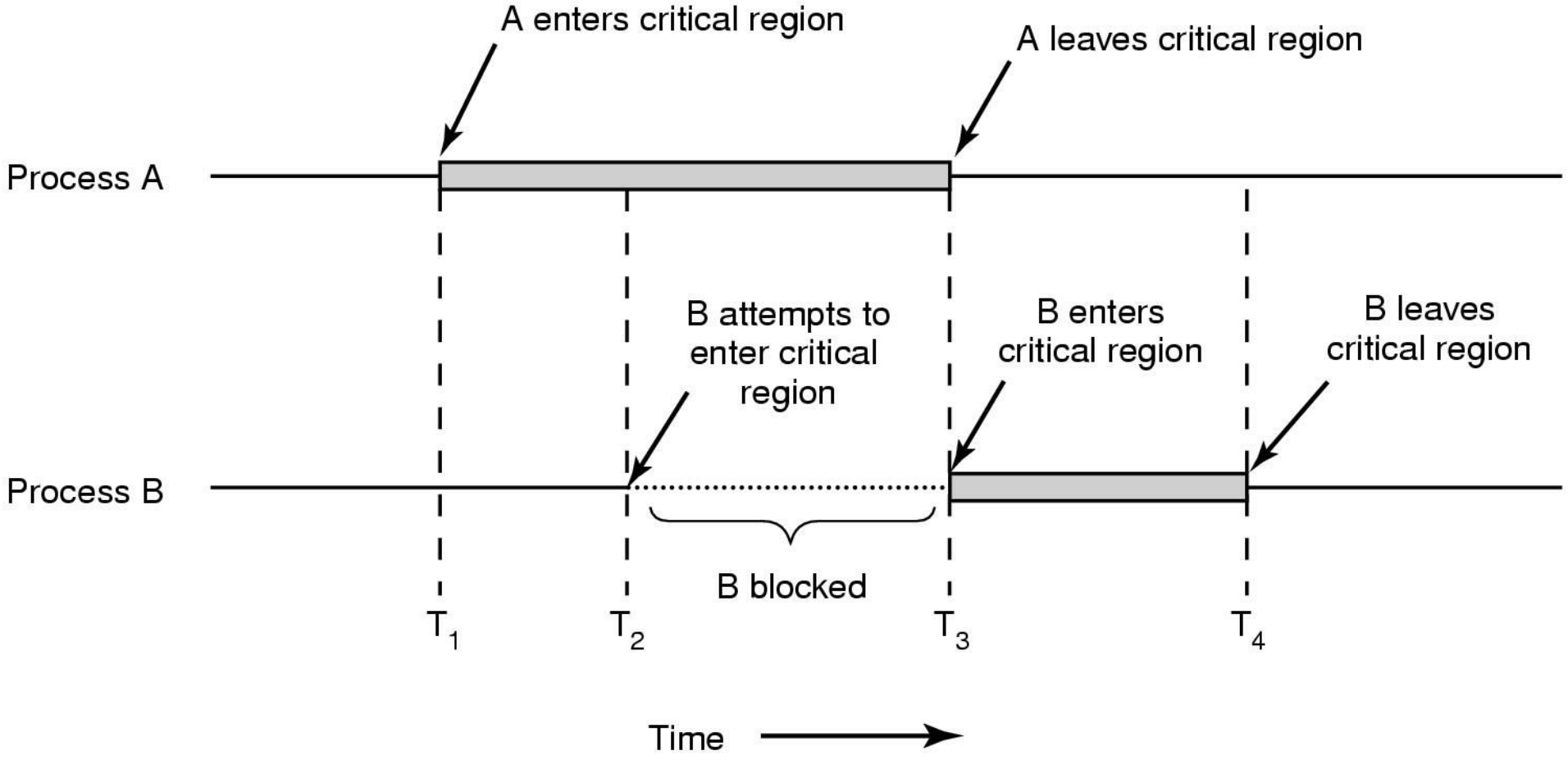
SITUAZIONE DI CORSA CRITICA
(Race condition)

Il problema può manifestarsi o meno a seconda della velocità relativa dei processi (*conseguenza: debugging* difficile)

Durante l'esecuzione delle sezioni critiche si possono avere *strutture dati che si trovano in uno stato non consistente* perché non ancora completamente modificate: la modifica non è «atomica» (indivisibile)

Le corse critiche (race conditions)

La procedura VersaSulConto è una *Sezione Critica*, bisogna evitare che tale porzione di codice venga intercalata con istruzioni appartenenti ad un'altra sezione critica che opera sugli stessi dati (nell'esempio, la stessa procedura, eseguita simultaneamente da un altro processo o thread).



Le corse critiche (race conditions)

Per controllare l'accesso alle sezioni critiche nel codice dei processi devono essere introdotte istruzioni di sincronizzazione all'inizio ed alla fine di ogni sezione critica:

P_i:

… // Sezione non critica

…

Istruzioni per la sincronizzazione in ingresso alla sezione critica

… //

… // Sezione critica

… //

Istruzioni per la sincronizzazione in uscita dalla sezione critica

…

… // Sezione non critica

Sezioni critiche: requisiti per una buona soluzione

Una soluzione soddisfacente al problema della mutua esclusione di sezioni critiche deve rispettare i seguenti requisiti:

[Mutua esclusione] se il processo Pi è in esecuzione nella propria sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica (sugli stessi dati)

[Progresso] se nessun processo è in esecuzione nella propria sezione critica, e vi sono dei processi che intendono entrare nelle rispettive sezioni critiche, la scelta su chi può procedere dipende solo da quali sono questi ultimi processi, e questa scelta non può essere rimandata indefinitamente

[Attesa limitata] quando un processo Pi ha richiesto di entrare in sezione critica, esiste un limite massimo al numero di volte per cui viene consentito ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi l'accesso a Pi

NB: Si assume che ogni processo rimanga in sezione critica per un tempo finito, ma non si può fare alcuna assunzione sulla velocità relativa dei processi

Sezioni critiche: requisiti per una buona soluzione

Una formulazione alternativa dei requisiti di una buona soluzione (dal libro di testo) è la seguente:

- deve essere garantita la mutua esclusione nell'esecuzione delle sezioni critiche da parte di tutti i processi coinvolti;
- la soluzione non può basarsi su ipotesi riguardo alle velocità relative di esecuzione dei processi coinvolti e al numero di CPU;
- un processo che sta eseguendo sezioni non critiche non deve impedire ad altri processi di accedere alla propria sezione critica;
- non può accadere che un processo debba attendere indefinitamente il proprio turno di entrare in sezione critica.

Disabilitazione interrupt

Il motivo per cui le istruzioni che compongono le sezioni critiche di due processi ***eseguiti in pseudo-parallelismo*** possono essere intercalate in modo arbitrario è che un processo può essere interrotto durante l'esecuzione (della sezione critica) a causa dell'arrivo di un timer interrupt (o altro interrupt nella gestione del quale venisse chiamato lo scheduler): se ciò avviene, il S.O. può assegnare la CPU ad un altro processo. Se si disabilitano gli interrupt tale eventualità non può più verificarsi.

PROBLEMI DI QUESTA SOLUZIONE:

- per motivi di protezione non si vuole permettere ad un processo che gira in modalità utente di disabilitare gli interrupt (potrebbe approfittarne per monopolizzare la CPU chiamando la procedura di "ingresso sezione critica")
- la soluzione funziona correttamente solo se il sistema ha un'unica CPU: nei sistemi multiprocessore la disabilitazione degli interrupt è locale a ciascuna CPU, e i processi possono eseguire davvero in parallelo (anzichè in pseudo-parallelismo).

Non-Soluzione dei problemi di sincronizzazione

Inizializzazione:

```
lock = 0;     /* lock è una variabile condivisa dai processi
              vale 0 se nessuno sta eseguendo all'interno
                  della sezione critica, 1 altrimenti */
```

Per entrare in sezione critica:

```
while (lock == 1) /* attendi */ ;
lock = 1;
```

attesa attiva (busy waiting): si attende usando CPU

In uscita dalla sezione critica:

```
lock = 0;
```

ATTENZIONE: la procedura per entrare in sezione critica è essa stessa una sezione critica!!! Può succedere che:

- **due processi valutano in (pseudo)parallelo (lock==1)**
- **entrambi la trovano falsa e ed eseguono lock = 1**
- **entrambi entrano nella sezione critica**

L'istruzione Test and Set Lock (TSL)

Un'altra strada è sfruttare l'aiuto dell'hardware, torniamo all'idea di utilizzare una variabile "lock" condivisa

Per avere come ATOMICA (indivisibile) la sequenza di azioni che effettua il test sul suo valore e poi la imposta a 1, ci si fa fornire dall'hw una istruzione DI LINGUAGGIO MACCHINA (non interrompibile, anche in presenza di sistemi multicore o multiprocessore):

TSL *registro*, *variabile*

che ATOMICAMENTE opera sul *registro* e sulla *variabile* in memoria:

1) copia *variabile* in *registro* (che può poi essere *test*ato)
2) imposta *variabile* a 1 (*set*)

Nei sistemi multiprocessore, la TSL deve *riservare* il bus e mantenerlo finché non sono concluse sia la lettura che la scrittura della variabile

Soluzione dei problemi di sincronizzazione con busy waiting

L'istruzione Test and Set Lock (TSL)

La TSL (istruzione di livello ISA) può essere usata nel modo seguente per garantire la mutua esclusione:

```
enter_region:
    TSL REGISTER,LOCK      | copy lock to register and set lock to 1
    CMP REGISTER,#0        | was lock zero?
    JNE enter_region       | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0           | store a 0 in lock
    RET | return to caller
```

Busy Waiting

Soluzione dei problemi di sincronizzazione con busy waiting

Vista in un linguaggio ad alto livello, può essere utilizzata come segue per garantire la mutua esclusione delle sezioni critiche di n processi:

Inizializzazione:

lock = 0;

Per entrare in sezione critica:

realizzata con TSL, assegna 1 a lock e restituisce il valore precedente

while (TestAndSet(&lock)) ; // busy waiting

In uscita dalla sezione critica:

lock = 0;

Questa soluzione soddisfa *tutti* i requisiti per una buona soluzione al problema della mutua esclusione?

NON SODDISFA IL REQUISITO DI ATTESA LIMITATA (ma si usa lo stesso per sezioni di breve durata; non vediamo una soluzione più complessa che usa sempre TSL)

Problemi di sincronizzazione

L'esecuzione in mutua esclusione di sezioni critiche
non è l'unico tipo di problema di sincronizzazione tra processi

Altri esempi di problemi di sincronizzazione sono:

- gestire la condivisione di un pool di N risorse (equivalenti) fra due o più processi/threads: ogni p./t. deve chiedere di acquisire una risorsa prima di utilizzarla, rimanere in attesa se non ve ne sono disponibili, procedere ad utilizzarla solo dopo che l'acquisizione è andata a buon fine, rilasciarla (esplicitamente) nel momento in cui l'utilizzo è concluso

  (possiamo vedere le sezioni critiche come caso particolare con N=1, la «risorsa» sono le variabili condivise)

- garantire che la porzione di codice A di P1 sia eseguita prima della porzione di codice B di P2
- garantire che N threads completino TUTTI una prima fase di esecuzione prima di poter passare a una seconda fase

Problema del produttore consumatore

Il problema "classico" del Produttore e Consumatore incorpora diversi problemi di sincronizzazione: la necessità di *ordinare* correttamente le attività dei due processi e di *far attendere un processo quando non sono disponibili risorse* (per esempio questo problema si presenta quando due processi - in UNIX - comunicano tramite una *pipe*).

- Il *produttore* può inserire dati nel buffer solo se esistono posizioni vuote (inizialmente disponibili, o liberate dal consumatore)

  Nota: possiamo considerare le posizioni disponibili nel buffer come risorse

- Il *consumatore* può prelevare dati dal buffer solo se il buffer contiene dati non ancora prelevati

Nota: possiamo considerare le posizioni «piene» nel buffer come risorse

Astrazioni per la sincronizzazione

Soluzioni basate su variabili di lock e istruzioni come *test-and-set-lock* sono adeguate solo per alcuni problemi di sincronizzazione semplici, ma *sono eccessivamente di "basso livello"* per problemi più complessi

per sviluppare applicazioni concorrenti (con più thread o più processi che cooperano o che devono coordinarsi nell'uso di risorse condivise) *sono utili meccanismi più di alto livello*

È inoltre opportuno cercare di eliminare (o almeno ridurre al minimo) l'attesa attiva (*busy waiting*) introducendo operazioni sospensive

Vedremo:

- semafori
- mutex nei Pthreads (POSIX threads)

Problemi del busy waiting: l'inversione di priorità

Il ***busy waiting*** deve essere evitato nei limiti del possibile (o almeno utilizzato solo quando la probabilità di dover attendere è bassa e comunque la condizione che causa l'attesa dura poco) perchè:

- Spreca tempo di CPU (il processo/thread che attende è ready/running e se la CPU è una sola la condizione non può cambiare fino a che non viene data ad un altro processo/thread).
- Nel caso di processore singolo può verificarsi il problema dell'**inversione di priorità**:

  consideriamo due processi P1 e P2, dove P1 ha priorità su P2 (tra P1 e P2, lo scheduler sceglie sempre P1; NB lo scheduler sceglie tra i processi pronti, non si occupa di sincronizzazione e non va certo a vedere il codice per accorgersi che stanno facendo attesa attiva)

  Se P1 cicla, ad es., `while (lock)` e P2 è il processo il cui codice dovrebbe fare `lock=0`, non esiste via d'uscita dato che lo scheduler selezionerà sempre P1, (o almeno si perde molto tempo se lo scheduler tende a scegliere P1 rispetto a P2)

Un'idea per evitare busy waiting: sleep e wakeup

Per evitare il busy waiting si introducono due *system call* con la seguente funzione:

- quando un processo esegue una ***sleep*** viene sospeso (*waiting*)…
- …finché qualcuno non lo sveglia con una ***wakeup***
- ***wakeup*** non ha alcun effetto se eseguita su un processo non bloccato.

Si possono immaginare due varianti di *sleep* e *wakeup*:

- nella prima variante *sleep* non ha parametri e *wakeup* ha un unico parametro, l'id del processo da svegliare:

```
P1:                 P2:
.....               .....
 sleep()                wakeup(P1)
.....               .....
```

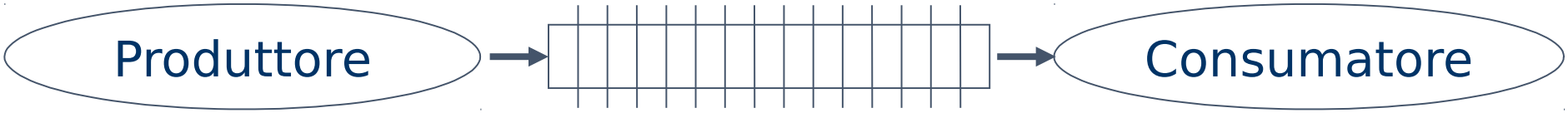
- nella seconda variante *sleep* deve specificare l'identificatore di una "condizione di attesa" *cond*, alla quale farà riferimento anche la *wakeup*; se piú processi possono essere in attesa su *cond* occorre definire se la *wakeup* deve svegliarne solo uno o svegliarli tutti.

```
 P1:                P2:
.....               .....
 sleep(cond)            wakeup(cond)
.....               .....
```

Problema del produttore-consumatore (P-C)

Proviamo a risolvere il problema del produttore /consumatore evitando il busy waiting:

- Il *produttore* può inserire dati nel buffer solo se esistono posizioni vuote
- Il *consumatore* può prelevare dati dal buffer solo se il buffer non è completamente vuoto



- utilizzo di sleep:
  Se non vi sono posizioni vuote il produttore *attende*
  Se non vi sono posizioni piene il consumatore *attende*
- utilizzo di wakeup:
  Il produttore sveglia il consumatore bloccato in attesa di un dato
  Il consumatore sveglia il produttore bloccato in attesa di uno spazio libero

Non-soluzione del problema P-C con sleep e wakeup

```
void producer(void)
{
 int item;

 while (TRUE){
      item = produce_item();
      if (count==N) sleep();
      insert_item(item);
      count++;
      if (count==1)
wakeup(consumer);
    }
}
```

```
void consumer(void)
{
 int item;

 while (TRUE){
      if (count==0) sleep();
      item = remove_item();
      count--;
      if (count==N -1) wakeup(producer);
      process_item(item);
    }
}
```

Le operazioni che inseriscono/prelevano effettivamente i dati nel/dal buffer condiviso sono insert_item e remove_item, il buffer può essere un vettore in cui due indici (per inserire ed estrarre) scorrono circolarmente (quando si arriva in fondo si torna all'inizio)

Errore nella soluzione del problema P-C con sleep e wakeup

```
void producer(void)
{
 int item;

 while (TRUE){
      item = produce_item();
      if (count==N) sleep();
      insert_item(item);
      count++;
      if (count==1) wakeup(consumer);
    }
}
```

```
void consumer(void)
{
 int item;

 while (TRUE){
      if (count==0) sleep();
      item = remove_item();
      count--;
      if (count==N -1) wakeup(producer);
      process_item(item);
    }
}
```

Se:

- il consumatore ha trovato count==0 e sta per eseguire sleep()
- appena prima di eseguirla viene interrotto (dal timer)
- viene schedulato il produttore che inserisce un item, incrementa count, ed essendo count==1 esegue wakeup(consumer)

quest'ultima **non ha effetto** dato che il consumatore non è bloccato (non ha ancora eseguito sleep()). A questo punto il consumatore esegue sleep(), e va a dormire ... *per sempre*

Errore nella soluzione del problema P-C con sleep e wakeup

```
void producer(void)
{
 int item;

 while (TRUE){
      item = produce_item();
      if (count==N) sleep();
      insert_item(item);
      count++;
      if (count==1) wakeup(consumer);
    }
}
```

```
void consumer(void)
{
 int item;

 while (TRUE){
      if (count==0) sleep();
      item = remove_item();
      count--;
      if (count==N -1) wakeup(producer);
      process_item(item);
    }
}
```

Anche se eseguissimo wakeup() ad ogni inserimento (eliminando if(count==1)) si possono raggiungere situazioni non desiderate

Ad esempio se il produttore produce un numero limitato di dati, l'ultimo dei quali indica al consumatore che il lavoro è concluso, e proprio quest'ultimo viene perso, il consumatore rimane sospeso in eterno

Occorre una soluzione che non faccia perdere le segnalazioni di *wakeup()*