

# Cooperazione tra Processi

- Principi
- Il problema della sezione critica: le *race condition*
- Supporto hardware
- Semafori
- Monitor
- Scambio di messaggi
- Barriere
- Problemi classici di sincronizzazione

## **Cos'è la concorrenza?**

- È un tema centrale nella progettazione dei S.O. e riguarda la gestione di processi multipli
  - Multiprogramming: più processi su un solo processore, parallelismo apparente
  - Multiprocessing: più processi su una macchina con processori multipli, parallelismo reale
  - Distributed processing: più processi su un'insieme di computer indipendenti, parallelismo reale

# Teoria della concorrenza

- Due processi si dicono in **esecuzione concorrente** se vengono eseguiti in parallelo (con parallelismo reale o apparente)
- La **teoria della concorrenza** studia
  - l'insieme di notazioni (linguaggi) per descrivere l'esecuzione concorrente di programmi
  - l'insieme di tecniche per risolvere i problemi associati all'esecuzione concorrente, quali comunicazione e sincronizzazione

## Applicazioni della concorrenza

- Multiprogrammazione: utilizzo efficiente di un processore
- Estensione della modularità in programmazione basate sullo sviluppo di applicazioni formate da insiemi di processi concorrenti
- Sviluppo di parti del sistema operativo (ad es. demoni)

# Processi (e Thread) Cooperanti

- L'esecuzione di processi *indipendenti* non può modificare o essere modificata dall'esecuzione di un altro processo.
- L'esecuzione di processi *cooperanti* può modificare o essere modificata dall'esecuzione di altri processi.
- Vantaggi della cooperazione tra processi:
  - Condivisione delle informazioni
  - Aumento della computazione (parallelismo)
  - Modularità
  - Praticità implementativa/di utilizzo

# IPC: InterProcess Communication

Meccanismi di comunicazione e interazione tra processi (e thread)

Questioni da considerare:

- Come può un processo passare informazioni ad un altro?
- Come evitare accessi inconsistenti a risorse condivise?
- Come sequenzializzare gli accessi alle risorse secondo la causalità?

Mantenere la consistenza dei dati richiede dei meccanismi per assicurare l'esecuzione ordinata dei processi cooperanti.

## Tipologia di comunicazione

- Scambio di messaggi
  - comunicazione diretta/indiretta (tramite una mailbox)
  - code di messaggi con capacità zero/limitata/illimitata
  - comunicazione sincrona/asincrona
- Memoria condivisa (=canale)

# Notazione per programmi concorrenti

- Notazione implicita

```
cobegin
...statement S1...
||
...statement S2...
||
...
||
...statement SK...
coend
```

- Ogni statement  $S_1, \dots, S_K$  viene eseguito in concorrenza
- Le istruzioni che seguono il `coend` verranno eseguite solo quando tutto gli statement sono terminati

## Esercizio

- Supponiamo di voler definire un programma con cobegin-coend che implementa Mergesort
- Abbiamo a disposizione le funzioni
  - $\text{sort}(v,i,j)$ : ordina gli elementi dell'array  $v$  dall'indice  $i$  all'indice  $j$
  - $\text{merge}(v,i,j,k)$ : fa il merge dei due segmenti (che supponiamo già ordinati) di  $v$  che vanno rispettivamente da  $i$  a  $j$  e da  $j+1$  a  $k$

## Possibile soluzione

```
mergesort(v,l)=  
{  
  m= l/2  
  cobegin  
    sort(v,1,m);  
    ||  
    sort(v,m+1,l);  
    ||  
    merge(v,1,m,l);  
  coend  
}
```

Le tre procedure operano in parallelo sullo stesso array (e quindi le chiamate a sort e merge non sono indipendenti)

## Soluzione corretta

```
mergesort(v,l)=  
{  
  m= l/2  
  cobegin  
    sort(v,1,m);  
    ||  
    sort(v,m+1,l);  
  coend;  
  merge(v,1,m,l);  
}
```

# Notazione per programmi concorrenti

- Notazione esplicita

Risorse condivise

...dichiarazioni di variabili...

```
Processo P1 {  
... istruzioni ...  
}
```

....

```
Processo PK {  
... istruzioni ...  
}
```

- In questo caso tutti i frammenti di codice P1, P2, ... vanno intesi come programmi (sequenziali) eseguiti in parallelo condividendo le variabili dichiarate come risorse condivise

- Il codice nei sottoprogrammi  $P_1, \dots, P_K$  può contenere istruzioni quali assegnamenti, if-then-else, while-loop
- NOTA: l'esecuzione in parallelo non implica l'esecuzione ripetuta dei sottoprogrammi: se  $P_1$  non ha cicli allora verrà eseguito una sola volta in parallelo con gli altri programmi

# Esempio 1

```
Var x=0
```

```
Processo P1 {
```

```
  x:=500;
```

```
}
```

```
Processo P2 {
```

```
  x:=0;
```

```
}
```

```
Processo P3 {
```

```
  write(x);
```

```
}
```

```
...
```

## **Soluzione esempio 1**

Nell'esempio 2 vi sono due possibili output (valori scritti sul monitor dall'istruzione write): 0 o 500.

Nota:

L'esecuzione di tale programma da origine ad un solo valore

Esecuzioni diverse possono dare origine a risultati diversi

A differenza dei programmi sequenziali la funzione definita dai programmi paralleli associa all'input un'insieme di possibili output e non un singolo valore

Si parla in questo caso di programmi con comportamento non deterministico

## Esempio 2

```
Var x=0
```

```
Processo P1 {
```

```
    while (true) x:=500;
```

```
}
```

```
Processo P2 {
```

```
    while (true) x:=0;
```

```
}
```

```
Processo P3 {
```

```
    while (true) write(x);
```

```
}
```

```
...
```

## Soluzione esempio 2

Nell'esempio 2 vi sono un numero infinito di possibili output

Ognuno di essi è a sua volta una sequenza infinita  $v_1v_2v_3\dots$  con  $v_i \in \{0, 500\}$  per  $i \geq 1$

Ad esempio un possibile output è 0 500 0 500 0 500 ...

Nota:

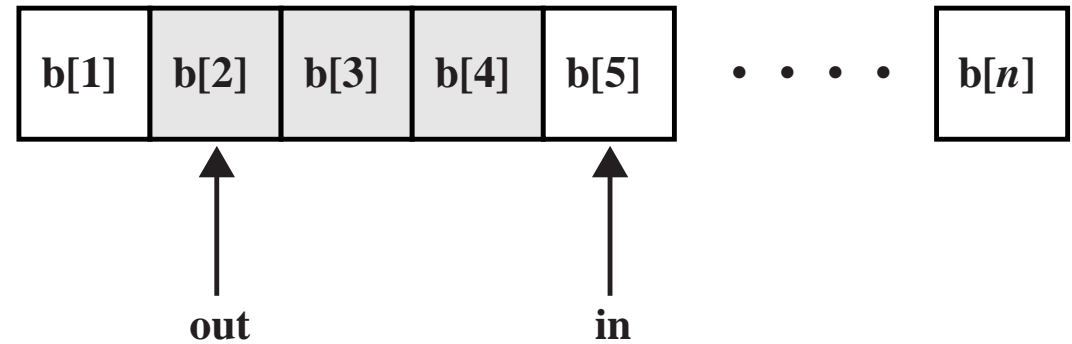
L'esecuzione di tale programma da origine ad una sola sequenza infinita

Tuttavia esecuzioni diverse possono dare origine a risultati diversi (sequenze infinite diverse)

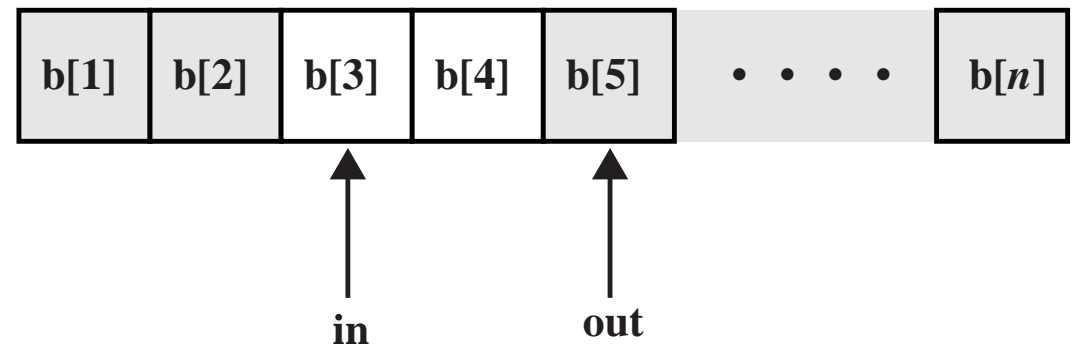
# **Problemi e primitive di Sincronizzazione**

# Esempio: Problema del produttore-consumatore

- Tipico paradigma dei processi cooperanti: il processo *produttore* produce informazione che viene consumata da un processo *consumatore*
- Soluzione a memoria condivisa: tra i due processi si pone un buffer di comunicazione di dimensione fissata.



(a)



(b)

## Produttore-consumatore con buffer limitato

- Dati condivisi tra i processi

```
type item = ... ;  
var buffer: array [0..n-1] of item;  
in, out: 0..n-1;  
counter: 0..n;  
in, out, counter := 0;
```

Processo produttore

**repeat**

...

produce un item in *nextp*

...

**while** *counter = n* **do** *no-op*;

*buffer[in] := nextp*;

*in := in + 1 mod n*;

*counter := counter + 1*;

**until** *false*;

Processo consumatore

**repeat**

**while** *counter = 0* **do** *no-op*;

*nextc := buffer[out]*;

*out := out + 1 mod n*;

*counter := counter - 1*;

...

consumo l'item in *nextc*

...

**until** *false*;

## Attenzione!

- Le istruzioni

- $counter := counter + 1;$

- $counter := counter - 1;$

devono essere eseguite *atomicamente*: se eseguite in parallelo non atomicamente, possono portare ad inconsistenze.

## Inconsistenze

- Notate infatti che l'istruzione  $counter := counter + 1$  è solitamente compilata in codice assembly che esegue due fasi distinte:
  - Prima viene calcolato il valore di  $counter + 1$ , cioè si sposta il valore di  $counter$  in un registro e si esegue l'incremento
  - Poi si sposta il risultato di nuovo in  $counter$
- Nota: In generale per gestire l'istruzione  $X := Exp$  occorre prima calcolare il valore di  $Exp$  in un registro e poi spostarlo nella cella associata ad  $X$

## Inconsistenze

- Supponiamo che *counter* abbia valore = 10
- L'esecuzione concorrente delle istruzioni

$$I1 = counter := counter + 1$$

$$I2 = counter := counter - 1$$

dà origine alle seguenti tracce (interleaving di istruzioni di due processi)

## Traccia 1

- Eseguo completamente  $I1$  e poi  $I2$
- il valore finale di *counter* è ancora 10

## Traccia 2

- Eseguo completamente  $I2$  e poi  $I1$
- il valore finale di *counter* è ancora 10

### Traccia 3

- Eseguo  $I1$  fino alla valutazione di  $counter + 1 (= 11)$  in un registro  $R$ ,
- lo scheduler interrompe  $I1$  e passa a  $I2$
- eseguo completamente  $I2$ ,
- ripristino il vecchio valore di  $R$  e termino l'esecuzione di  $I1$  (sposto il valore di  $R$  in  $counter$ )
- il valore finale è 1  $counter$  è ora 11

## Traccia 4

- Eseguo  $I2$  fino alla valutazione di  $counter - 1 (= 9)$  in un registro  $R$ ,
- lo scheduler interrompe  $I2$  e passa a  $I1$ ,
- eseguo completamente  $I1$ ,
- ripristino il vecchio valore di  $R$  e termino l'esecuzione di  $I2$
- il valore finale di  $counter$  è ora 9

- Quindi ho **tre** possibili valori per *counter* dopo l'esecuzione concorrente di *I1* e *I2*: 9, 10, 11.
- Questo tipo di inconsistenze sono dovute a *race condition* sulle variabili condivise (i.e. *counter*)

# Race condition

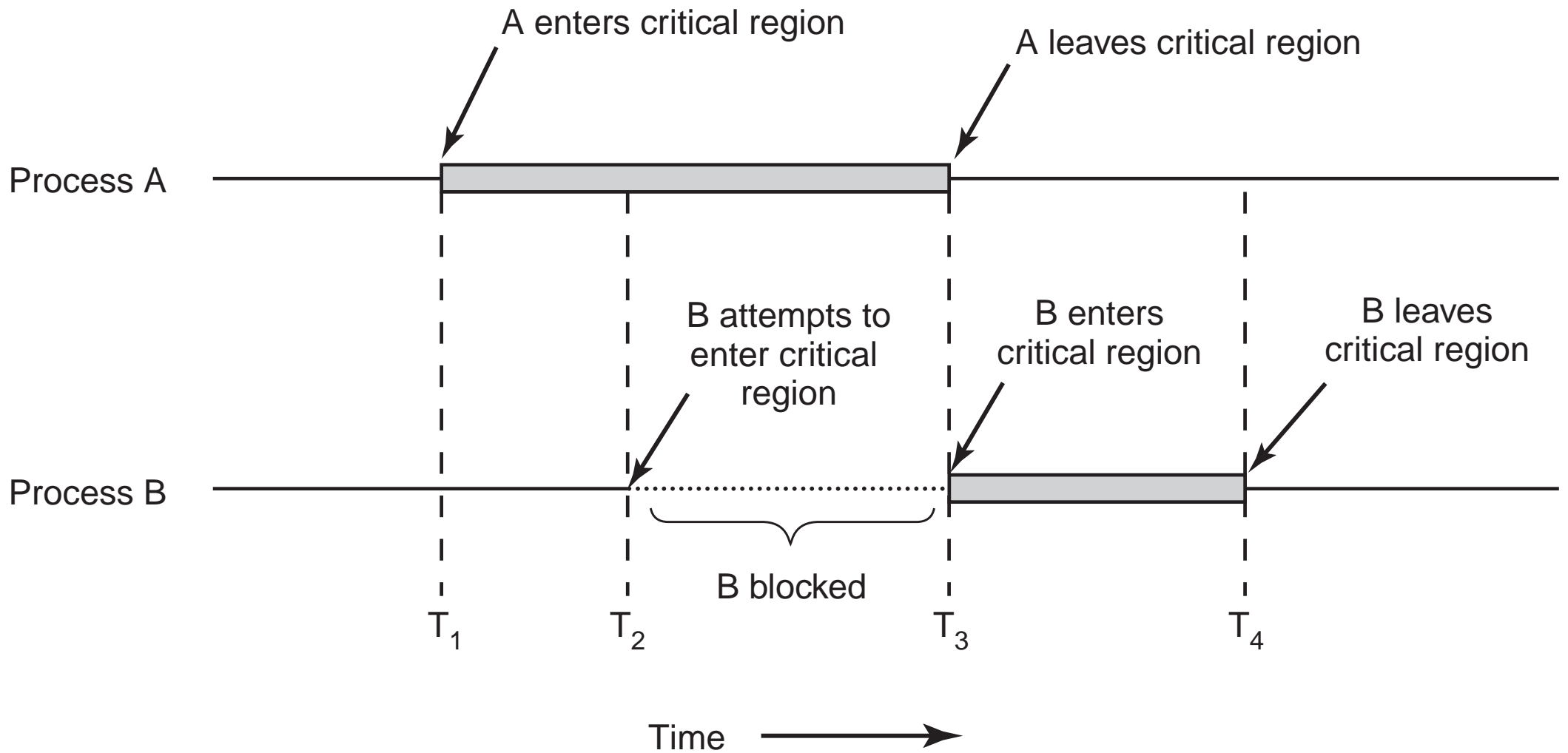
**Race condition:** più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di interleaving dei processi.

- Frequenti nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel.
- Estremamente pericolose: portano al malfunzionamento dei processi co-operanti, o anche (nel caso delle strutture in kernel space) dell'intero sistema
- difficili da individuare e riprodurre: dipendono da informazioni astratte dai processi (decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, . . . )

## Problema della Sezione Critica

- $n$  processi che competono per usare dati condivisi
- Ogni processo ha un segmento di codice, detto *sezione critica* in cui si accede ai dati condivisi.
- Problema: assicurare che quando un processo esegue la sua sezione critica, nessun altro processo possa entrare nella propria sezione critica.
- Bisogna proteggere la sezione critica con apposito *codice di controllo*

```
while (TRUE) {  
    entry section  
    sezione critica  
    exit section  
    sezione non critica  
};
```



## Criteri per una Soluzione del Problema della Sezione Critica

1. **Mutua esclusione:** se il processo  $P_i$  sta eseguendo la sua sezione critica, allora nessun altro processo può eseguire la propria sezione critica.
2. **Progresso:** se nessun processo è nella sezione critica e esiste un processo che desidera entrare nella propria sezione critica, allora l'esecuzione di tale processo non può essere posposta indefinitamente.
3. **Attesa limitata:** se un processo  $P$  ha richiesto di entrare nella propria sezione critica, allora il numero di volte che si concede agli altri processi di accedere alla propria sezione critica prima del processo  $P$  deve essere limitato.

# Assunzioni

- Occorre specificare quali istruzioni sono eseguite in modo *atomico* (cioè non sono interrompibili)
- L'esecuzione all'interno della sez. crit. termina sempre in un tempo finito
- Pur non facendo assunzioni sulla velocità relativa dei vari processi, si suppone che ogni processo venga eseguito ad una velocità non nulla ed in particolare si parla di esecuzione in *weak fairness*:
  - se la prossima istruzione di un processo è un'istruzione non bloccante prima o poi (entro un tempo finito) il processo la potrà eseguire;
  - se la prossima istruzione di un processo è bloccante ma dopo un certo tempo non lo è più, allora il processo potrà prima o poi eseguire l'istruzione (sbloccata).
- Lavoriamo quindi con uno scheduler *ideale* che assicura tale proprietà per ogni processo.

## Progresso: Definizioni alternative

- Progresso: varie definizioni

1. Se nessun processo e' nella sezione critica ed esiste un processo che desidera entrare nella propria sezione critica, allora l'esecuzione di tale processo non puo' essere posposta indefinitamente.
2. Se nessun processo e' in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori dalle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che puo' entrare per primo nella propria sezione critica; questa scelta non puo' essere rimandata indefinitamente
3. Un processo al di fuori della sua sezione critica non puo' prevenire altri processi dall'entrare la propria; i processi che cercano simultaneamente di accedere alla sezione critica devono decidere quale processo entra.

- Nota: il *deadlock* (tutti i processi sono bloccati) rappresenta violazione di progresso

## Bounded waiting

- Bounded waiting
  1. Se un processo ha gi richiesto l'ingresso nella sua sezione critica, esiste un limite superiore al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche priam che si accordi la richiesta al primo processo
- Nota: *starvation* (un processo non entra mai) rappresenta una violazione di bounded waiting

## Soluzioni software

- Supponiamo che ci siano solo 2 processi,  $P_0$  e  $P_1$
- Struttura del processo  $P_i$  (l'altro sia  $P_j$ )

```
while (TRUE) {  
    entry section  
    sezione critica  
    exit section  
    sezione non critica  
}
```

- Supponiamo che i processi possano condividere alcune variabili (dette *di lock*) per sincronizzare le proprie azioni

# Flag condiviso

- Variabili condivise

- **var** *occupato*: (0..1);

- inizialmente *occupato* = 0

- *occupato* = 0  $\Rightarrow$  un processo può entrare nella propria sezione critica

- Processo  $P_i$

```
while (TRUE) {  
    while (occupato  $\neq$  0) no-op;  
    occupato := 1;  
    sezione critica  
    occupato := 0;  
    sezione non critica  
};
```

- Problema: Non funziona

Lo scheduler può interrompere il processo subito dopo il ciclo interno e prima dell'istruzione `occupato:=1`.

- P1 passa il test `occupato=0` (esce dal loop)

S.O. interrupt

P2 passa il test `occupato=0` (esce dal loop)

P2 `occupato:=1`

P2 sezione critica

S.O. interrupt

P1 `occupato:=1`

P1 sezione critica

# Alternanza stretta

- Variabili condivise

- **var** *turn*: (0..1);

- inizialmente *turn* = 0

- *turn* = *i*  $\Rightarrow$   $P_i$  può entrare nella propria sezione critica

- Processo  $P_i$

```
other := 1 - i;
```

```
while (TRUE) {
```

```
    while (turn  $\neq$  i) no-op;
```

```
        sezione critica
```

```
    turn := other;
```

```
        sezione non critica
```

```
};
```

- Problemi

Soddisfa il requisito di mutua esclusione, ma non di progresso per processi con differenze di velocità

- P1 entra nella sezione critica

P1 esce dalla sezione critica

P1 cerca di entrare nella sezione critica

P2 e' molto lento; fino a quando P2 non entra e poi esce dalla CS, P1 non puo' entrare

## Busy Wait

- Il precedente algoritmo utilizza la nozione *dbusy wait*: attesa *attiva* di un evento da parte di un processo (es: testare il valore di una variabile).
  - Semplice da implementare
  - Porta a consumi inaccettabili di CPU
  - In genere, da evitare, ma a volte è preferibile (es. in caso di attese molto brevi)
- Un processo che attende attivamente su una variabile esegue uno *spin lock*.

## **Bounded waiting vs Progresso**

Utilizzando alcune idee viste nei precedenti tentativi cerchiamo di dimostrare che le proprietà di progresso e bounded waiting non sono equivalenti

## Controesempio 1

- Entry nella sez. critica controllata con una variabile condivisa *turn*: i processi entrano a turno stretto ( $P_1, P_2, \dots, P_k, P_1, \dots, P_k, P_1, \dots, P_k, \dots$ ).  
Cioè quando  $P_i$  esce setta  $turn = i + 1$  (modulo  $k$ )
- Se  $P_i$  e' nella sezione NON critica ed esegue un loop infinito allora non vale progresso (tutti gli altri processi sono bloccati per sempre) .
- Tuttavia vale bounded waiting. Infatti i processi aspettano al piu'  $k-1$  turni.

## Controesempio 2

- Scegliamo in modo casuale il processo che entra nella sezione critica dall'insieme dei processi in attesa di entrare.
- Vale progresso, Infatti nessun processo al di fuori della sezione critica può influenzare la scelta di quale processo può entrare.
- Non vale bounded waiting (nel caso peggiore  $P_i$  non entra mai: starvation).

## Algoritmo di Peterson (cont)

- Basato su una combinazione di *richiesta* e *accesso*
- Soddisfa tutti i requisiti; risolve il problema della sezione critica per 2 processi
- Si può generalizzare a  $N$  processi
- È ancora basato su spinlock

# Algoritmo di Peterson

```
#define FALSE 0
#define TRUE 1
#define N      2          /* number of processes */

int turn;                /* whose turn is it? */
int interested[N];      /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;           /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;       /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

## Algoritmo di Peterson (cont)

- Le procedure *enter\_region* ed *exit\_region* si utilizzano nel seguente modo per programmare una soluzione al problema della sezione critica per **2 processi**

```
var interested: array[0..1] of boolean = (false,false);  
var turn : [0..1] = 0;
```

```
process  $P_0$  {  
    enter_region(0);  
    sezione critica  
    exit_region(0);  
}  
  
process  $P_1$  {  
    enter_region(1);  
    sezione critica  
    exit_region(1);  
}
```

$P_0$  e  $P_1$  vengono eseguiti in parallelo

## Correttezza dell'Algoritmo di Peterson

### Mutua Esclusione:

- Supponiamo che  $P_0$  entri nella sezione critica, allora  $interested[1] = FALSE$  oppure  $turn = 1$ .
- Se  $interested[1] = FALSE$  allora  $P_1$  non può essere in sezione critica (prima di entrarvi deve settare il flag a  $TRUE$ ).
- Se  $interested[1] = TRUE$  e  $turn = 1$ , allora  $P_1$  è in attesa che la condizione del suo spin-lock diventi falsa e quindi non può essere in sezione critica.
- Lo stesso ragionamento si applica invertendo  $P_0$  con  $P_1$

## Progresso

- Supponiamo che  $P_0$  sia in attesa di entrare in sezione critica, i.e., esegue il proprio spin-lock.
- Se  $P_1$  rilascia oppure non è interessato alla sezione critica (i.e.  $interested[1] = FALSE$ )  $P_0$  può entrare dopo un tempo finito in S.C.
- Se  $P_1$  è interessato alla sezione critica ( $interested[1] = TRUE$ ) e  $P_1$  ha settato per ultimo la variabile  $turn$ ,  $P_0$  può accedere alla sezione critica ( $turn = 0$  è falsa)  
Se  $P_0$  ha settato la variabile  $turn$  per ultimo, allora  $P_1$  entra ( $turn = 1$  è falsa)  
Quando  $P_1$  rilascia la sezione critica (per ipotesi dopo un tempo finito) torniamo al caso del punto precedente (II punto).

## Bounded Waiting

- Segue dall'analisi fatta per il progresso. Se  $P_0$  aspetta di entrare in sezione critica (cioè esegue lo spin-lock),  $P_1$  potrà entrare al più una volta in S.C.

## Osservazione su Algoritmo di Peterson

- Il processo corrente può anche assegnare l'identificatore del processo oppo-  
nente alla variabile *turn*.
- In questo caso lo spin lock deve contenere la condizione

$turn == other \text{ and } interester[other] == TRUE$

## Altra versione dell'algoritmo di Peterson

- Si ottiene quindi il seguente codice:

```
void enter_region(int process);
{
    int other;
    other:=1-process;
    interested[process]:=TRUE;
    turn:=other;
    while (turn==other and interested[other]==TRUE) do no-op;
}
```

```
void exit_region(int process);
{
    interested[process]:=FALSE;
}
```

## Soluzione per $n$ processi: Algoritmo del Fornaio

Risolve la sezione critica per  $n$  processi, generalizzando l'idea vista precedentemente.

- Prima di entrare nella sezione critica, ogni processo riceve un numero.
- Chi ha il numero più basso entra nella sezione critica.

## Algoritmo del Fornaio (di Lamport): Prima versione

*Variabili globali*

**var** number: **array**[1,...,N] of integer;

Process  $P_i$

{

**while** TRUE **do**

*number*[i] = **max**{*number*[1],...,*number*[N]} + 1;

**for**  $k : 1$  **to**  $N$  **do**

**while** (*number*[k]  $\neq$  0 **and** *number*[k] < *number*[i]) **do no-op**;

    - *critical section* -

*number*[i] = 0;

}

## Algoritmo del Fornaio (di Lamport): Completa

La selezione del numero (ticket) in generale non può essere fatta in modo atomico: richiede la scansione dei numeri di tutti i processi

L'algoritmo completo funziona nel seguente modo:

- Prima di entrare nella sezione critica, ogni processo riceve un numero. Chi ha il numero più basso entra nella sezione critica.
- Se i processi  $P_i$  and  $P_j$  ricevono lo stesso numero: se  $i < j$ , allora  $P_i$  è servito per primo; altrimenti  $P_j$  è servito per primo.
- Lo schema di numerazione genera numeri in ordine crescente

# Algoritmo del Fornaio

```
var choosing: array[1,...,N] of boolean;  
var number: array[1,...,N] of integer;
```

Process  $P_i$

{

**while** TRUE **do**

*choosing*[**i**] = *TRUE*;

*number*[**i**] = **max**{*number*[1], ..., *number*[*N*]} + 1;

*choosing*[**i**] = *FALSE*;

**for**  $k : 1$  **to**  $N$  **do**

**while** *choosing*[ $k$ ] **do no-op**;

**while** (*number*[ $k$ ]  $\neq 0$  **and** ( $\langle \textit{number}[k], k \rangle \ll \langle \textit{number}[i], i \rangle$ )) **do no-op**;

    - *critical section* -

*number*[**i**] = 0;

}

Dove  $\langle a, b \rangle \ll \langle c, d \rangle$  sse  $a < c$  oppure  $a = c$  e  $b < d$

## **Soluzioni hardware**

- Istruzioni per disabilitare interrupt
- Istruzioni speciali per rendere atomica l'esecuzione di un test e di un assegnamento

# Disabilitazione degli interrupt

- Il processo può disabilitare TUTTI gli interrupt hw all'ingresso della sezione critica, e riabilitarli all'uscita
  - Soluzione semplice; garantisce la mutua esclusione
  - ma pericolosa: il processo può non riabilitare più gli interrupt, acquisendosi la macchina
  - può allungare di molto i tempi di latenza
  - non scala a macchine multiprocessore (a meno di non bloccare tutte le altre CPU)
- Inadatto come meccanismo di mutua esclusione tra processi utente
- Adatto per brevi(ssimi) segmenti di codice affidabile (es: in kernel, quando si accede a strutture condivise)

## Istruzioni speciali: Test-and-Set-Lock

- Le istruzioni di Test-and-Set-Lock: testano e modificano atomicamente il contenuto di una variabile/cella di memoria

$TS(x,y) := \text{atomico}( y = x ; x = 1 )$

$TS(x,y)$  ritorna in  $y$  il valore precedente di  $x$

- assegna 1 ad  $x$

item Questi due passi devono essere atomici Cioe' abbiamo bisogno di una ipotetica istruzione

```
TSL RX,LOCK
```

che copia il contenuto della cella LOCK nel registro RX, e poi imposta la cella LOCK ad un valore  $\neq 0$ .

Il tutto atomicamente (viene bloccato il bus di memoria).

## Utilizzo di TS (cont.)

*Variabili condivise:*

**var** *occupato*: boolean;

Process  $P_i$

**var** *aux*: boolean;

**while** TRUE **do**

**repeat**

*TS(occupato, aux)*;

**until** *aux = 0*;

  - *critical section* -

*occupato = 0*;

**end.**

- Assicura mutua esclusione e progresso
- Tuttavia e' basato su spinlock e quindi genera busy wait

## Evitare il busy wait

- Le soluzioni basate su spinlock portano a
  - busy wait: alto consumo di CPU
  - inversione di priorità: un processo a bassa priorità che blocca una risorsa viene ostacolato nella sua esecuzione da un processo ad alta priorità in busy wait sulla stessa risorsa.
- Idea migliore: quando un processo deve attendere un evento, che venga posto in *wait*; quando l'evento avviene, che venga posto in *ready*
- Servono specifiche syscall o funzioni di kernel. Esempio:
  - *sleep()*: il processo si autosospende (si mette in *wait*)
  - *wakeup(pid)*: il processo *pid* viene posto in *ready*, se era in *wait*.

Ci sono molte varianti. Molto comune: con *evento* esplicito.

# Produttore-consumatore con sleep e wakeup

```
#define N 100                                     /* number of slots in the buffer */
int count = 0;                                   /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                               /* repeat forever */
        item = produce_item();                  /* generate next item */
        if (count == N) sleep();               /* if buffer is full, go to sleep */
        insert_item(item);                     /* put item in buffer */
        count = count + 1;                      /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);     /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                               /* repeat forever */
        if (count == 0) sleep();               /* if buffer is empty, got to sleep */
        item = remove_item();                  /* take item out of buffer */
        count = count - 1;                     /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                   /* print item */
    }
}
```

## Produttore-consumatore con sleep e wakeup (cont.)

- Risolve il problema del busy wait
- Non risolve la corsa critica sulla variabile *count*
- I segnali possono andare *perduti*, con conseguenti *deadlock*

## Problemi con sleep e wakeup

- Esempio di perdita di un segnale:
  - Supponiamo che il buffer sia vuoto.
  - Il consumatore esegue  $count = 0$  e prima di eseguire *sleep* viene interrotto dallo scheduler.
  - Il produttore va in esecuzione, incrementa *count* e cerca di svegliare un processo sospeso
  - Il segnale va perso perchè il consumatore non è ancora sospeso (è ready e sta aspettando di eseguire *sleep*)
  - Lo scheduler seleziona nuovamente il consumatore il quale esegue *sleep* e si sospende.
  - Il produttore continua a riempire il buffer fino a che è pieno.

- Il produttore esegue la *sleep* e il sistema va in deadlock (i due processi sono bloccati)
- Soluzione: salvare i segnali “in attesa” in un contatore