

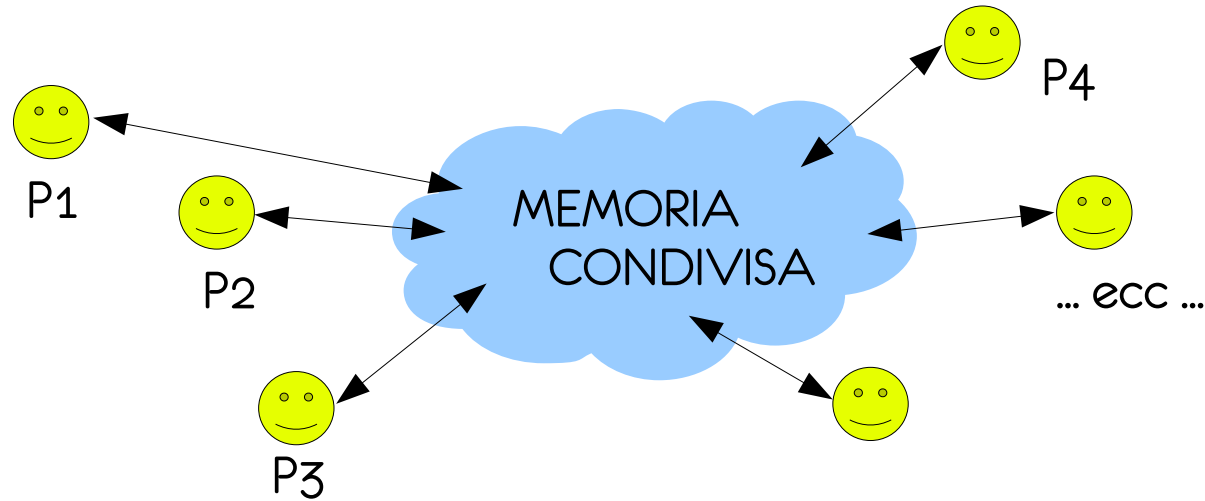
# esecuzione concorrente asincrona

capitolo 6 del libro (VII ed.)  
e capitolo 5 del libro di Deitel, Deitel e  
Choffnes

approfondimento: Dijkstra, E. W. (1971, June).  
[Hierarchical ordering of sequential processes.](#)  
Acta Informatica 1(2): 115-138.

Chandy, K. M., e Misra, J.  
[The drinking philosophers problem](#) (1984), ACM.  
Link su moodle

# Introduzione 1/2

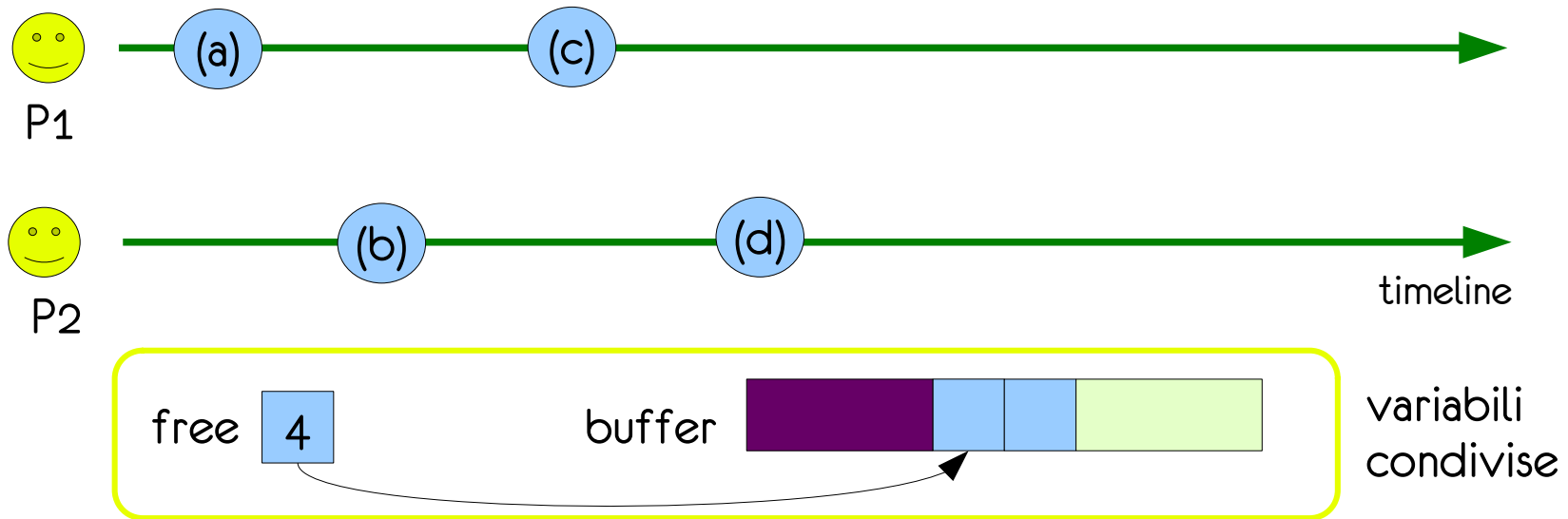


- Consideriamo **due** processi **produttori** che inseriscono dati nella stessa area di memoria condivisa
- L'indice della prima posizione libera è indicato da `free`
- Inserzione:

1) `buffer[free] = dato`  
2) `free = (free+1) % D`

questo codice può creare problemi in un contesto di concorrenza?

# Introduzione 2/2



- Non ci sono controlli! L'interleaving potrebbe produrre a una evoluzione dell'esecuzione dove le due istruzioni non sono eseguite in modo atomico, es:

|                |  |               |                         |
|----------------|--|---------------|-------------------------|
| (a) P1 esegue: | $\text{buffer}[\text{free}] = \text{dato}$ | $\Rightarrow$ | $\text{buffer}[4] = 2$  |
| (b) P2 esegue: | $\text{buffer}[\text{free}] = \text{dato}$ | $\Rightarrow$ | $\text{buffer}[4] = 18$ |
| (c) P1 esegue: | $\text{free} = (\text{free} + 1) \% D$     | $\Rightarrow$ | $\text{free} = 5$       |
| (d) P2 esegue: | $\text{free} = (\text{free} + 1) \% D$     | $\Rightarrow$ | $\text{free} = 6$       |



i dati risultano inconsistenti!!!!

## Ancora peggio . . .

- Nell'esempio precedente il buffer condiviso era legato a un applicativo utente
- Molte strutture dati condivise (tabella dei file aperti, tabella delle pagine, ecc.) sono strutture usate dal SO!!!
- Un SO con multitasking, in cui possono essere presenti allo stesso tempo diversi processi eseguiti in modalità kernel, può creare inconsistenze nelle tabelle di sistema

# Sezione critica 1/4

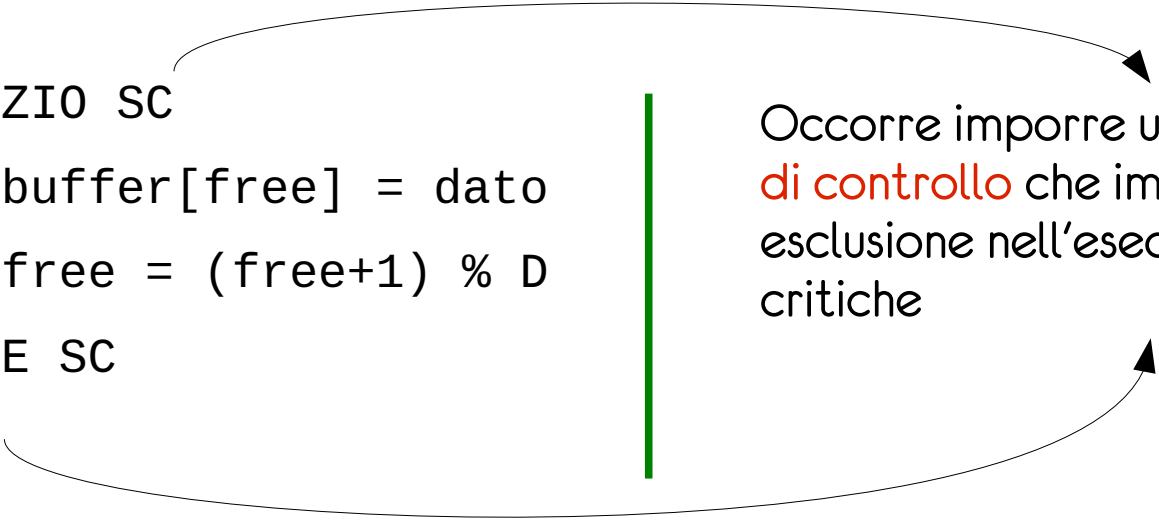
- Per **sezione critica** si intende una porzione di codice in cui un processo modifica variabili condivise (in generale dati condivisi)
- Requisito: se più processi che condividono una variabile, solo uno di essi per volta può essere nella sua sezione critica (**mutua esclusione**)
- NB: due processi possono essere simultaneamente in sezione critica se tali sezioni si riferiscono a variabili differenti

1) INIZIO SC

2) `buffer[free] = dato`

3) `free = (free+1) % D`

4) FINE SC



Occorre imporre un **meccanismo di controllo** che implementi la mutua esclusione nell'esecuzione di sezioni critiche

# Sezione critica 2/4

- Struttura del codice

1) sezione non critica

2) sezione di ingresso

3) `buffer[free] = dato`

4) `free = (free+1) % D`

5) sezione di uscita

6) sezione non critica

corrisponde ad una richiesta, fatta al meccanismo di gestione, di accedere alla sezione critica

**sezione critica**

avvisa il meccanismo di gestione che è possibile consentire ad un altro processo l'accesso in sezione critica

# Sezione critica 3/4

- Una sezione critica:
  - è determinata dalle variabili condivise utilizzate
  - è un segmento di codice che non deve essere eseguito con interleaving di istruzioni di altre sezioni critiche appartenenti alla stessa famiglia (che usano le stesse variabili condivise)
- **Problema:** definire un meccanismo che ne consenta l'utilizzo corretto

# Sezione critica 4/4

- Criteri soddisfatti da una soluzione al problema della sezione critica:
  - 1.Mutua esclusione:** Un solo processo per volta può eseguire la propria sezione critica
  - 2.Progresso:** Nessun processo che **non** desideri utilizzare una variabile condivisa può impedirne l'accesso a processi che desiderano utilizzarla. Solo i processi che intendono entrare in sezione critica concorrono a determinare chi entrerà
  - 3.Attesa limitata:** esiste un limite superiore all'attesa di ingresso in sezione critica



# Soluzione 1

- Vediamo una prima soluzione al problema, nel caso in cui si abbiano solo **due** processi. La soluzione si appoggia a una variabile **turno**: se  $turno == i$  allora  $P_i$  può accedere alla sezione critica.
- Codice di  $P_i$ :

<sezione non critica>

while (turno != i) do no\_op;

▼ sezione di ingresso

<sezione critica>

turno = j;

▼ sezione di uscita

<sezione non critica>

# Critica

- **Pro:**

- la soluzione garantisce la mutua esclusione

- **Contro:**

- la soluzione impone una stretta alternanza fra i processi: e se il turno fosse uguale ad  $i$  ma il processo  $P_i$  non avesse alcuna intenzione di entrare in sezione critica? Allora neppure  $P_j$  potrebbe entrarvi, pur volendolo
- si viola quindi la condizione di progresso:  $P_i$  è in sezione non critica, non dovrebbe impedire a  $P_j$  di entrare in sezione critica
- inoltre il `while (...) no_op` realizza un'attesa attiva: il processo occupa la CPU per non fare nulla, per aspettare (spreco!!)

# Idea!

- Per evitare la stretta alternanza:
  - usare **due** variabili anziché una sola
  - ciascuna variabile indica **l'intenzione** di un processo di entrare in sezione critica
  - **problema**: entrambi i processi potrebbero desiderare di entrare in sezione critica in uno stesso momento ...
  - per accedere alla sezione critica devo verificare se la variabile dell'altro processo è impostata

# Soluzione 2

- La variabile turno è sostituita da un array, *flag*, di due booleani inizializzati a *false*
- Codice di  $P_i$ :

```
flag[i] = true;  
while (flag[j]) do no_op;
```

sezione di ingresso

<sezione critica>

```
flag[i] = false;
```

sezione di uscita

<sezione non critica>

# Critica

- **Pro:**

- la soluzione garantisce la mutua esclusione
- la soluzione evita la stretta alternanza fra i due processi

- **Contro:**

- la sezione di ingresso **non è eseguita in maniera atomica**: cosa succede se  $P_i$  e  $P_j$  desiderano entrambi entrare in sezione critica? `flag[i] == true` e `flag[j] == true`!! I due rimangono bloccati nel while successivo!!
- inoltre il while (...) no\_op realizza **un'attesa attiva**: il processo occupa la CPU per non fare nulla, per aspettare (spreco!!)

# Idea!

- Per evitare lo stallo:
- posso **usare sia flag che turno**, la variabile turno viene presa in considerazione solo quando entrambi i processi in competizione desiderano entrare in sezione critica ed hanno flag a true (race condition)
- turno, quindi, dirime le dispute che portano allo stallo

# Soluzione 3

- La variabile *flag*, di due booleani inizializzati a *false*, è affiancata dalla variabile *turno*, che indica il processo da favorire in caso di competizione

- Codice di  $P_i$ :

```
flag[i] = true; turno = j;  
while (flag[j] && turno == j) do no_op;
```

sezione di ingresso

<sezione critica>

flag[i] = false;

sezione di uscita

<sezione non critica>

---

L'**algoritmo di Peterson** somiglia nell'uso di flag e turno ad un algoritmo precedente, l'algoritmo di Dekker a lungo considerato "la" soluzione al problema però è più semplice

# Commento

- Sezione di ingresso:

```
(1) flag[i] = true;  
(2) turno = j;  
(3) while (flag[j] && turno == j) do no_op;
```

- la variabile turno è unica e condivisa, se i due processi eseguono la loro sezione di ingresso in parallelo, uno dei due sarà l'ultimo ad accedere (e settare) il valore di turno. Alcuni possibili interleaving:

- P1-1, P1-2, P2-1, P2-2: turno vale 1
- P1-1, P1-2, P2-1, P1-3, P2-2, P2-3: ... segue ...

(provate a fare diverse simulazioni con diversi interleaving, anche spezzando l'esecuzione della condizione del while)



# Simulazione

• P1-1, P1-2, P2-1, P1-3,

```
P1
(1) flag[1] = true;
(2) turno = 2;
(3) while (flag[2] &&
          turno == 2) do no_op;
```

```
P2
(1) flag[2] = true;
(2) turno = 1;
(3) while (flag[1] &&
          turno == 1) do no_op;
```

|      | P1  |       | P2                             |
|------|---|-------|--------------------------------|
|      |   | tempo |                                |
| P1-1 | flag[1]=true                                  |       |                                |
| P1-2 | turno = 2                                     |       |                                |
| P2-1 |   |       | flag[2] = true                 |
| P1-3 | flag[2] && turno == 2<br>vera!                |       |                                |
| P2-2 |   |       | turno = 1                      |
| P2-3 |   |       | flag[1] && turno == 1<br>vera! |
| P1-3 | flag[2] && turno == 2<br>falsa! Entro in S.C. |       |                                |

NB: turno è una variabile  
condivisa dai due processi

il meccanismo funziona perché  
ogni processo cede (educata-  
mente) il turno all'altro

# Critica

- **Pro:**
  - la soluzione garantisce la **mutua esclusione**, evitando la stretta alternanza fra i due processi
  - **progresso**: chi esce dalla sezione critica mette a false il proprio flag, quindi l'altro processo avrà modo di uscire dal proprio while
  - **attesa limitata**: l'attesa di un processo nel proprio while dura quanto la sezione critica dell'altro processo
  - l'algoritmo è estendibile a N processi (**algoritmo del fornaio**)
- **Contro:**
  - il while (...) no\_op realizza un'attesa attiva: il processo occupa la CPU per non fare nulla, per aspettare (spreco!!)

# Soluzione per N processi

- La prima soluzione semplice e veloce per il caso più generale a N processi venne proposta da Lamport ed è nota come **Algoritmo del Fornaio** (o del panettiere)
- L'idea è di utilizzare un **ticket con numero crescente** per dirimere le race condition (la competizione)

# Soluzione per N processi

- Codice di  $P_i$ :

```
choosing[i] = true;
ticket[i] = max_ticket + 1;
choosing[i] = false;
for (j = 0; j < N; j++) {
    while (choosing[j]) no_op;
    while (ticket[j] != 0 &&
           ticket[j] < ticket[i] ||
           ticket[j] == ticket[i] && j < i) no_op;
}
```

sezione di ingresso



<sezione critica>

ticket[i] = 0;  sezione di uscita

<sezione non critica>

# Soluzione per N processi

## • Codice di $P_i$ :

```
choosing[i] = true;
ticket[i] = max_ticket + 1;
choosing[i] = false;
```

sto richiedendo un biglietto  
mi viene dato un biglietto  
non sto più richiedendo un biglietto

```
for (j = 0; j < N; j++) {
    while (choosing[j]) no_op;
    while
        (ticket[j] != 0 &&
         ticket[j] < ticket[i] ||
         ticket[j] == ticket[i] && j < i) no_op;
}
```

aspetto eventualmente che gli venga rilasciato il biglietto  
se il processo j desidera entrare in SC  
e ha un biglietto precedente il mio oppure ...

per ogni processo concorrente eseguo un controllo

ha lo stesso numero ma il suo id di processo è precedente il mio, allora cedo il passo

# Soluzione per N processi

## • Codice di $P_i$ :

```
choosing[i] = true;  
ticket[i] = max_ticket + 1;  
choosing[i] = false;
```

due processi possono avere lo stesso ticket a causa di un'interleaving nell'esecuzione di questa linea di codice

```
for (j = 0; j < N; j++) {  
    while (choosing[j]) no_op;  
  
    while  
        (ticket[j] != 0 &&  
         ticket[j] < ticket[i] ||  
         ticket[j] == ticket[i] &&  
         j < i) no_op;  
}
```

so che prima o poi toccherà a me perché i ticket contengono numeri crescenti, per i quali esiste un solo ordinamento (crescente)

Quando un processo è servito deve richiedere un nuovo ticket per entrare in SC, mettendo il proprio ticket a zero

Attendo ciclando i controlli su ticket[j] finché la condizione non diventa vera e posso controllare il ticket del processo successivo

# Critica

- **Pro:**
  - la soluzione garantisce la **mutua esclusione**
  - **progresso e attesa limitata**: sono verificate
- **Contro:**
  - codice e dati sono **complessi**
  - i processi fanno **busy-waiting**, cioè anziché sospendersi attendono il turno tenendo occupata la CPU

Ci sono alternative? L'alternativa è introdurre opportuni supporti hardware

# Sincronizzazione HW



# Sincronizzazione hardware

- **Soluzione 1:** all'ingresso di una sezione critica, **disabilitare gli interrupt** per impedire il context switch.

# Sincronizzazione hardware

- **Soluzione 1:** all'ingresso di una sezione critica, **disabilitare gli interrupt** per impedire il context switch.
- Senza context switch non è possibile la prelazione!
- **Problemi:**
  - interferenze pesanti con lo scheduling della CPU (una sezione critica può essere molto lunga ...)
  - gli interrupt notificano eventi da gestire, non possono essere mantenuti disabilitati a lungo

# Sincronizzazione hardware

- **Soluzione 2:** introdurre l'uso di “lock” (lucchetti).
- Per entrare in sezione critica un processo deve avere ottenuto il giusto lock, che rilascia al termine

# Sincronizzazione hardware

- **Soluzione 2:** introdurre l'uso di “**lock**” (lucchetti).
- Per entrare in sezione critica un processo deve avere ottenuto il giusto lock, che rilascia al termine
  - a questo fine, molte architetture forniscono istruzioni che consentono di
    - (1) controllare e modificare il valore di una cella di memoria oppure
    - (2) scambiare il contenuto di due celle di memoria in modo atomico
  - in particolare: **TestAndSet** e **Swap**

# TestAndSet

- **TestAndSet è atomica**: viene eseguita con interrupt disabilitati, quindi se due processi lanciano ciascuno una TestAndSet, le due esecuzioni verranno sequenzializzate (no interleaving delle istruzioni)

# TestAndSet

- **Implementazione:**

```
boolean TestAndSet (boolean *variabile) {  
    boolean valore = *variabile;  
    *variabile = true;  
    return valore;  
}
```

- salva in una variabile locale il valore puntato dal parametro, mette a true il valore puntato dal parametro, restituisce il valore salvato
- Cosa c'è di speciale?
- L'atomicità dell'esecuzione dell'intera routine

# Uso di TestAndSet

- realizziamo le sezioni di ingresso e uscita da una sezione critica tramite TestAndSet

# Uso di TestAndSet

- occorre dichiarare una variabile globale, accessibile ai processi concorrenti, di tipo booleano
- sia essa chiamata "lock"



# Uso di TestAndSet

- realizziamo le sezioni di ingresso e uscita da una sezione critica tramite TestAndSet
- occorre dichiarare una variabile globale, accessibile ai processi concorrenti, di tipo booleano. Sia essa chiamata "lock":

```
while (TestAndSet(&lock));
```

```
<sezione critica>
```

```
lock = false;
```

sezione di ingresso

sezione di uscita

- Perché funziona?

# Uso di TestAndSet

- realizziamo le sezioni di ingresso e uscita da una sezione critica tramite TestAndSet
- occorre dichiarare una variabile globale, accessibile ai processi concorrenti, di tipo booleano. Sia essa chiamata "lock":

```
while (TestAndSet(&lock));
```

```
<sezione critica>
```

```
lock = false;
```

sezione di ingresso

sezione di uscita

- TestAndSet restituisce il valore precedente di lock che sarà falso sse nessun altro processo è in una sezione critica controllata tramite lock. Solo in questo caso si esce dal while