

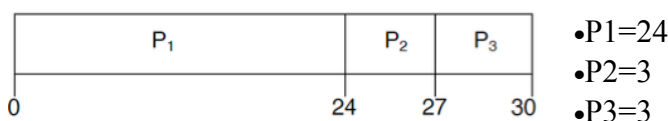
# ARCHITETTURE DI CALCOLO LEZIONE 18

## Scheduling della CPU

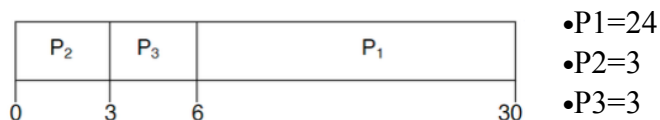
Tra le diverse modalità di scheduling della CPU ci sono:

- **FCFS (First Come First Served)**, nel quale il primo processo arrivato, è il primo ad essere eseguito (coda FIFO).

Se consideriamo tre processi P1, P2, P3, arrivati tutti al tempo  $t=0$ , con un tempo di burst:



Otterremo lo schema di Gantt a destra ed un tempo di waiting medio pari a  $(24+3+3)/3=17$ . In questa tipologia di scheduling si può venire a creare l'effetto "convoglio", che si verifica quando un processo "lungo" ritarda l'esecuzione di processi "brevi". Per esempio, consideriamo tre processi, arrivati nell'ordine P2, P3, P1, con un tempo di burst:



Otterremo lo schema di Gantt a destra e un tempo di waiting medio pari a  $(6+0+3)/3=3$ , di molto inferiore rispetto l'esempio precedente

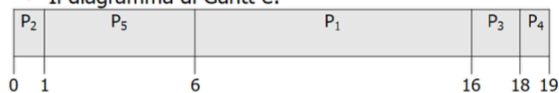
- **Con priorità**: utilizzato dalla maggior parte dei SO (come Windows e Linux), assegna ad ogni processo un intero che rappresenta la priorità; la CPU viene assegnata al processo con più alta priorità (es. il più piccolo intero = la priorità più alta)

Lo scheduling con priorità si divide in:

Processo	Tempo di burst	Priorità
P <sub>1</sub>	10	3
P <sub>2</sub>	1	1
P <sub>3</sub>	2	4
P <sub>4</sub>	1	5
P <sub>5</sub>	5	2

Priorità massima

- Il diagramma di Gantt è:



- Tempi di attesa: P<sub>1</sub>→6, P<sub>2</sub>→0, P<sub>3</sub>→16, P<sub>4</sub>→18, P<sub>5</sub>→1
- Tempo medio di attesa  $T_a = (6+0+16+18+1)/5 = 8.2\text{msec}$

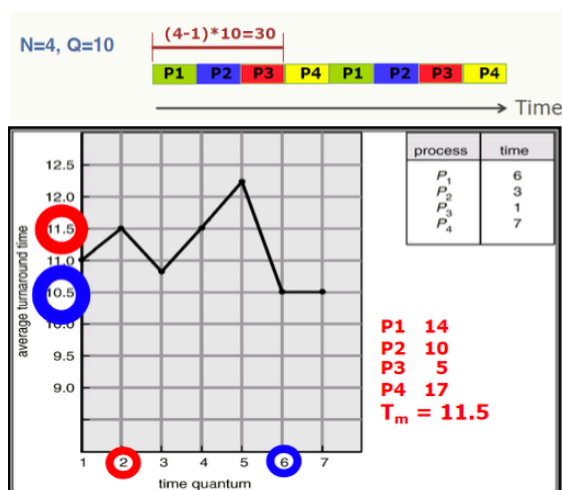
• **Nonpreemptive** (senza prelazione): il processo assegnato non può essere sospeso prima di completare il suo CPU burst.

• **Preemptive** (con prelazione): se arriva un nuovo processo con un CPU burst più breve del tempo rimanente per il processo corrente, viene servito.

Questa tipologia di scheduling, così come

l'SRTF (spiegato in seguito), vanno incontro ad una problematica detta "starvation", ovvero i processi a più bassa priorità potrebbero non essere mai eseguiti; la soluzione a ciò è l'aging, ossia, al trascorrere del tempo di attesa, si incrementa la priorità del processo che attende (approccio usato in diversi ambiti, tra cui quello sanitario; es. lista delle prenotazioni di esami o visite mediche).

- **Scheduling Round Robin:** ogni processo è assegnato alla CPU per un intervallo



temporale fissato (quanto di tempo o time slice) dell'ordine delle decine di millisecondi e, quando il tempo è trascorso, il processo viene tolto dalla CPU e inserito nella ready queue (che viene trattata come una coda FIFO).

Se ci sono  $N$  processi nella ready queue e il quanto di tempo è  $Q$ , allora ogni processo ottiene  $1/N$  del tempo della CPU a blocchi di lunghezza  $Q$ . Inoltre, nessun processo attende più di  $(N-1)Q$  unità di tempo; questa formula permette di conoscere il tempo di attesa massimo di ogni processo, valore importante per l'interattività del sistema.

Infine, è importante che  $Q$  sia maggiore del tempo di context switch, altrimenti la produttività della CPU risulterà estremamente scarsa; empiricamente, il tempo di context switch deve essere più lungo del tempo di CPU burst del 80% dei processi, implicando che 8 volte su 10, non è il quanto di tempo a scadere ma il processo.

In sintesi, lo scheduling con priorità comporta un maggior tempo di turnaround (tempo di completamento dei processi) dello scheduling SJF ma garantisce un miglior tempo di risposta.

La lunghezza della time slice influenza il tempo di completamento, come si può osservare nel grafico a destra; nell'esempio, con  $Q=2 \rightarrow$  tempo di turnaround medio = 11.5, mentre con  $Q=6 \rightarrow$  tempo di turnaround medio = 10.5.

- **SJF (Shortest Job First):** che associa ad ogni processo la lunghezza del prossimo CPU burst e usa questi tempi per schedulare il processo con la lunghezza minima.

I processi sono ordinati nella ready queue in base al loro prossimo CPU burst in ordine crescente (il primo processo ha il minimo CPU burst).

Il vantaggio più evidente è il poter ottenere il minimo tempo medio di waiting.

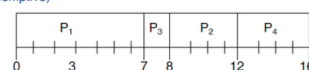
Lo scheduling SJF si divide in:

- **Nonpreemptive**

- **Preemptive** : Questo schema è conosciuto come Shortest-Remaining-Time-First (SRTF).

Processo	Tempo Arrivo	Tempo di Burst
$P_1$	0	7
$P_2$	2	4
$P_3$	4	1
$P_4$	5	4

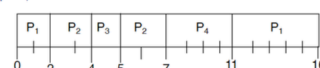
SJF (non-preemptive)



Tempo di attesa medio =  $(0 + 6 + 3 + 7)/4 = 4$

Processo	Tempo di Arrivo	Tempo di Burst
$P_1$	0	7
$P_2$	2	4
$P_3$	4	1
$P_4$	5	4

SJF (preemptive)

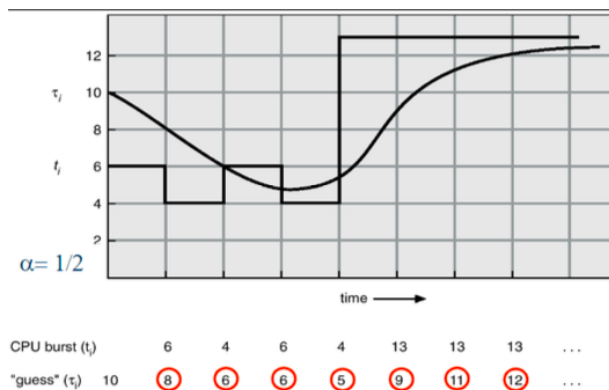


Tempo di attesa medio =  $(9 + 1 + 0 + 2)/4 = 3$

La problematica maggiore nello scheduling SJF è conoscere in anticipo la durata dei processi da elaborare, per poterli, poi, eseguire in ordine crescente di durata. Siccome non si può conoscere la lunghezza del prossimo CPU burst, questo è un valore che deve essere stimato tramite la formula:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

lunghezza dell'n-esimo CPU burst  
 valore predetto del prossimo CPU burst  
 $0 < \alpha < 1$  (generalmente  $\alpha = 1/2$ )



Nella stima viene effettuata la somma pesata della durata dell'evento immediatamente precedente ( $t_n$ ) e della storia della durata dell'evento ( $\tau_n$ ), ovvero la stima effettuata al passo precedente, usando una media esponenziale.

La formula illustrata si traduce nel grafico di fianco:

$t_i$  = valore reale

$\tau_i$  = valore stimato

Si osserva che l'andamento dei valori stimati segue quello dei valori veri con una reattività che dipende da  $\alpha$ . Al variare di quest'ultima, si possono generare due casi limite:

- $\alpha=0$       $\tau_{n+1} = \tau_n$

la storia recente non viene presa in considerazione: le condizioni attuali sono transitorie

- $\alpha=1$       $\tau_{n+1} = t_n$

si considera solo l'ultimo CPU burst: la storia passata è irrilevante

## Scheduling a code multiple

Una volta studiate quali sono le principali tipologie di scheduling, si può comprendere anche l'uso combinato di queste, che si verifica nello scheduling a code multiple, in cui il kernel gestisce più code (es. potrebbe esservi una coda per i processi in foreground ed un'altra per i processi in background).

Per attuare questa strategia, vi è bisogno di uno scheduler che gestisca i processi all'interno delle singole code e di un ulteriore scheduler che divida il tempo di CPU tra le diverse code (può essere diviso in parti uguali oppure privilegiando una delle code, generalmente quella che garantisce una maggiore interattività).

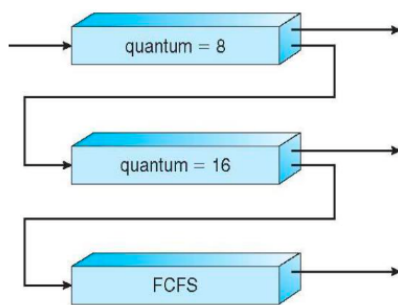
I processi si assegnano in modo permanente ad una coda, generalmente secondo qualche caratteristica (invariante) del processo. I SO moderni, generalmente, hanno quattro code:

- processi real-time (alta priorità)
- processi di sistema
- processi interattivi
- processi batch (bassa priorità)

Attraverso un feedback, un processo si può spostare tra le code. Questo evita situazioni di starvation o di eccessivo utilizzo della CPU.

Lo scheduler che usa code multiple con feedback usa i seguenti parametri:

- numero di code
- algoritmi di scheduling per ogni coda
- un metodo per "promuovere" un processo (attribuire una maggiore priorità)
- un metodo per "degradare" un processo (attribuire una minore priorità)
- un metodo per decidere in quale coda inserire un processo quando questo chiede un servizio.



Per comprendere al meglio questo meccanismo, immaginiamo ci siano tre code, di cui:

- la prima con scheduling round robin e quanto di tempo di 8 millisecondi
- la seconda con scheduling round robin e quanto di tempo di 16 millisecondi
- la terza con scheduling FCFS

Quando un processo arriva nel sistema, questo viene inserito nella prima coda; se viene completato entro il quanto di tempo (8 ms), esce dalla coda, altrimenti viene inserito nella seconda coda. Anche qui vale lo stesso meccanismo: se il processo viene completato entro il quanto di tempo (16 ms), esce dalla coda, altrimenti viene inserito nella terza coda, dove non c'è un tempo di completamento massimo, ma ne deriva che il tempo di risposta sarà maggiore. Questa strategia di scheduling favorisce i processi brevi, in quanto saranno completati più rapidamente.

Gli svantaggi di questo algoritmo sono:

- dover gestire tre code
- dover gestire tre algoritmi di scheduling
- dover suddividere il tempo di CPU tra le code

## Schedulazione multiprocessing

Consideriamo il caso di una macchina con più core, in cui la schedulazione diviene più complessa.

Si ipotizza di avere processori omogenei, dove il carico di lavoro viene suddiviso (load sharing) tra i vari processori.

I possibili approcci di schedulazione sono due:

- **Multiprocessamento asimmetrico:** solo un processore (il master) prende le decisioni relative allo scheduling. Gli altri processori fanno solo elaborazione.
- **Multiprocessamento simmetrico (SMP):** ciascun processore schedula sé stesso selezionando un processo dalla coda comune dei processi pronti o da una coda specifica per sé stesso. La maggior parte dei sistemi moderni sono simmetrici. L'unico elemento di sincronizzazione è la coda ed è necessario fare in modo che venga preservata.

## Scheduling Real-time

Per quanto riguarda i sistemi Real-time, esistono due possibili algoritmi di scheduling:

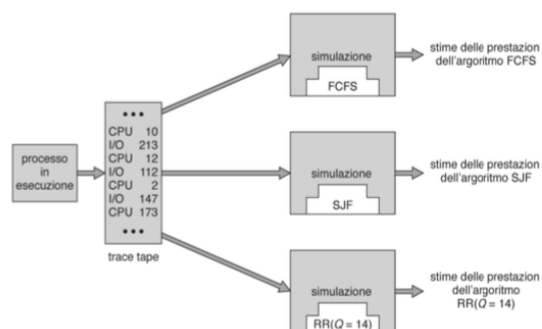
- **Sistemi hard real-time:** si basano sulla prenotazione delle risorse, in cui ad una richiesta da parte del processo (con indicazione sul tempo di completamento) segue un completamento solo in seguito ad accettazione della richiesta da parte del processore; qualora le risorse (ad esempio la memoria disponibile o il tempo di utilizzo della CPU a disposizione) non siano sufficienti, il processo non viene eseguito.
- **Sistemi soft real-time:** i processi critici ricevono priorità maggiore rispetto ai processi "normali" (non accetta prenotazione).

## Come scegliere un algoritmo di scheduling adatto/ottimale?

Si potrebbe provare a creare manualmente degli scenari di utilizzo dell'algoritmo e confrontarli con gli esiti di un altro algoritmo ma ipotizzare tutti i possibili valori di input e tempi di arrivo, burst, attesa, ecc. risulta in realtà impossibile da attuare.

Solitamente, i punti da seguire sono:

- Fissare i criteri di ottimizzazione
- Usare metodi di valutazione:
  - Modellazione Deterministica (valutazione analitica)  
Fissati i diversi carichi di lavoro, definisce le prestazioni dei diversi algoritmi per ognuno dei carichi analizzati.
  - Modelli di code  
Il sistema viene modellato come un insieme di server con le code associate; date le distribuzioni degli arrivi delle richieste si calcola la lunghezza media delle code, il tempo medio di attesa, etc.
  - Realizzazione
  - Simulazione: si danno una serie di input agli algoritmi, si ottengono tanti risultati e si fa una statistica.



## Scheduling in Windows

Schedulazione a code multiple con feedback basato su priorità dinamica e preemption (prelazione).

È di tipo di threading in cui un thread viene eseguito fino a che:

- Non è sottoposto a prelazione da parte di un thread con priorità più alta
- Non finisce il quanto di tempo assegnato
- Esegue una chiamata di sistema bloccante
- Termina

I thread sono divisi in classi, ognuna con intervalli di priorità:

- Classe variabili (da 1 a 15), sono processi non di sistema.
- Classe real-time (da 16 a 31), sono processi di sistema con priorità maggiore

Il dispatcher (scheduler) ha una coda per ogni priorità. Percorre le code dalla più alta alla più bassa fino a che trova un thread pronto.

La massima priorità è 31 e tali processi vengono eseguiti per primi.


Ogni coda è gestita attraverso l'algoritmo Round Robin, tranne l'ultima che è FCFS.

Un thread a priorità variabile può cambiare coda quando si verificano due eventi (feedback dello scheduler):

- Termina il suo quanto di tempo
  - La priorità del thread viene ridotta (limita l'uso della CPU ai thread CPU-bound)
- Viene sospeso a causa di un'operazione di attesa
  - La priorità del thread viene alzata
  - La quantità dell'incremento dipende dall'evento che si attende
  - Questo permette di favorire, ad esempio, le interfacce grafiche

- La schedulazione di un thread dipende dalla classe di priorità ma anche da una priorità relativa associata ad ogni thread

- Soft-real time

Priorità a classe variabile (da 1 a 15) 

Priorità relativa	Priorità di classe					
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Classe real-time (da 16 a 31)

Lo scheduler, inoltre, distingue fra processi in foreground e background; i processi in foreground hanno un quanto di tempo maggiore (tipicamente di un fattore 3). Tale sistema nasce da constatazioni empiriche a partire dai concetti di aging e priorità. Osserviamo nella tabella il concetto di code e priorità di windows.

## Scheduling di Linux

Anche Linux usa un sistema di priorità che va da 0 a 140. La priorità è decrescente (0=max priorità, 140 =min priorità); ciò che cambia tra le varie priorità è il quanto di tempo.

1. **real-time** - priorità tra 0 e 99.
2. **nice** - priorità tra 100 e 140.

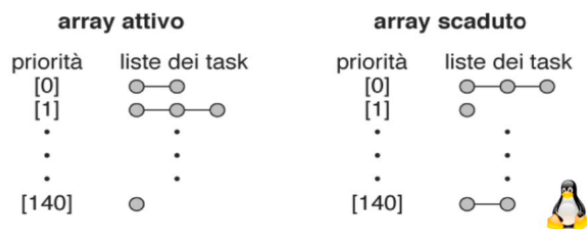


Quando un task consuma il suo quanto di tempo deve attendere che tutti gli altri abbiano consumato il loro, prima di essere eseguito.

In Linux, inoltre, sono presenti due array di priorità, contenenti gli indirizzi delle liste dei

task con la stessa priorità:

- Array attivo: contiene i task che hanno ancora del tempo da usare
- Array scaduto: contiene i task che hanno completato il loro tempo



Quando l'array attivo è vuoto, gli array vengono scambiati. Ogni lista è gestita da un algoritmo RR. Prima della versione 2.5, il kernel Linux utilizzava una variante dell'algoritmo tradizionale di scheduling di UNIX. Tuttavia, questo algoritmo non supporta adeguatamente sistemi multiprocessore e fornisce prestazioni

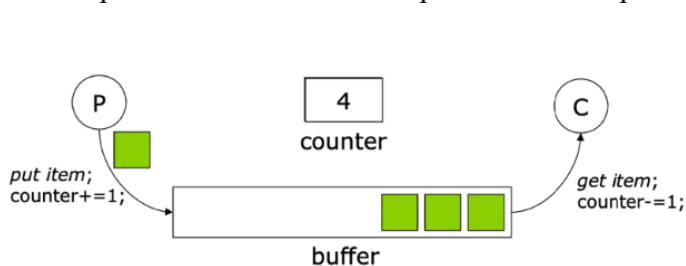
scarse nei sistemi in grado di eseguire un gran numero di processi.

Con la versione 2.5 del kernel, lo scheduler è stato rivisitato per includere un algoritmo di scheduling noto come O(1), che veniva eseguito in tempo costante indipendentemente dal numero di task nel sistema. Lo scheduler O(1) forniva anche un maggiore supporto ai sistemi SMP (symmetric multiprocessing), fra cui la gestione della predilezione e il bilanciamento del carico. Anche se lo scheduler O(1) forniva eccellenti prestazioni su sistemi SMP, portava a tempi di risposta troppo scarsi sui processi interattivi, comuni in molti sistemi desktop. Durante lo sviluppo del kernel 2.6, lo scheduler è stato nuovamente rivisto e dalla versione 2.6.23 del kernel, lo scheduler CFS (completely fair scheduler) è diventato l'algoritmo predefinito di scheduling Linux.

## SINCRONIZZAZIONE DEI PROCESSI

Se si ha un sistema con più processori la sincronizzazione tra i core è fondamentale. Se i core non fossero sincronizzati, l'accesso concorrente a risorse condivise potrebbe portare ad inconsistenze, cioè un risultato finale erroneo che è così non coerente con l'operazione.

Per comprendere tale concetto si presenta l'esempio del produttore e del consumatore o

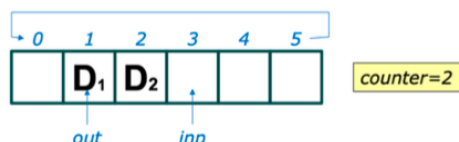


problema del buffer-limitato (bounded-buffer). Si ha un array di lunghezza limitata che viene utilizzato in condivisione da due processi, il consumatore e il produttore, per scambiarsi dei dati, gli item.



- Il compito del produttore è quello di generare ciclicamente dei dati e metterli nel buffer, mentre quello del consumatore è di rimuovere ciclicamente i dati dal buffer, uno alla volta.
- Bisogna garantire che il produttore non inserisca dati nel buffer quando questo è pieno e che il consumatore non li prelevi quando questo è vuoto.

Es.: Bounded-buffer con dimensione= 6, numDati = 2;



Il produttore inserisce l'item nel buffer tramite l'operazione "put" ed incrementa la dimensione del counter (counter +=1).

Il consumatore, invece, preleva l'item tramite l'operazione "get" e diminuisce il counter (counter -=1).

Osserviamo il codice del produttore:

- Inizialmente viene definito il buffer con la sua lunghezza e sono inizializzati a 0 il counter e l'inp (la posizione del buffer in cui il produttore inserisce l'item).
- Successivamente, si ha un ciclo while che dura fintanto che il counter ha lo stesso valore della dimensione del buffer:

### Produttore

```

BUFFER_SIZE = 10 # Define the buffer size
buffer = [None] * BUFFER_SIZE # Initialize the buffer as a list of None values
counter = 0 # Initialize the counter variable
inp = 0 # Initialize the 'inp' index variable

while True:
    # Produce an element and insert it in nextProduced
    nextProduced = ...

    while counter == BUFFER_SIZE:
        pass # Do nothing if the buffer is full

    buffer[inp] = nextProduced
    inp = (inp + 1) % BUFFER_SIZE
    counter += 1

```

- Se il counter continua ad essere uguale alla dimensione del buffer, il while persiste;
- Se il counter è diverso dal buffer size allora si avvia la seconda parte del codice: inserire nella posizione inp del buffer il nuovo prodotto. Bisogna, inoltre, riassegnare il valore di inp alla posizione successiva rispetto alla posizione precedente; la nuova posizione viene calcolata tramite modulo di (inp + 1) rispetto al buffer size per far sì che, quando si raggiunge l'estremità del buffer, la posizione ritorni a 0. Infine, si incrementa il counter di 1.

*Nota: la funzione modulo (%) restituisce il resto della divisione intera; ad esempio, se il buffer ha dimensione 8 (posizioni da 0 a 7) e inp da 7 deve essere incrementato, si ha  $(7+1)\%8$  che dà 0 (il resto della divisione 8/8).*

Il codice del consumatore prevede:

### Consumatore

```

while True:
    while counter == 0:
        pass # Do nothing if the counter is zero

    nextConsumed = buffer[out]
    out = (out + 1) % BUFFER_SIZE
    counter -= 1
    # Consume the item in nextConsumed

```

- Nella prima parte c'è un while che persiste fin quando il counter è uguale a 0
- Qualora il counter risulti diverso da 0 (un elemento è stato inserito), l'elemento in posizione out (una variabile che indica la posizione dell'item che viene preso dal consumatore) viene indicato come prossimo elemento consumato. Viene, poi, riassegnata



una nuova posizione out tramite la formula  $(out+1)\%buffer\ size$  e il contatore viene decrementato di 1. Sebbene siano corrette, se si considerano separatamente, le procedure del produttore e del consumatore possono non “funzionare” se eseguite in concorrenza. In particolare, le istruzioni  $counter+=1$  e  $counter-=1$ , a livello macchina, sono implementate nel seguente modo:

L'istruzione “**counter += 1**” può essere implementata in linguaggio macchina come:

**register<sub>1</sub> = counter**  
**register<sub>1</sub> = register<sub>1</sub> + 1**  
**counter = register<sub>1</sub>**

```
LOAD R1, CONT
ADD 1, R1
STORE R1, CONT
```

L'istruzione “**counter -= 1**” può essere implementata come:

**register<sub>2</sub> = counter**  
**register<sub>2</sub> = register<sub>2</sub> - 1**  
**counter = register<sub>2</sub>**

```
LOAD R2, CONT
SUB 1, R2
STORE R2, CONT
```

Nel caso in cui il produttore ed il consumatore cercassero di aggiornare la variabile counter contemporaneamente, le istruzioni in linguaggio macchina potrebbero risultare interfogliate (interleaved), cioè le due operazioni si vanno ad incastrare.

Figura 1: Counter è una variabile condivisa (globale).

Osserviamo un esempio di interleaving in cui la fase “salvataggio” della nuova variabile counter da parte del produttore viene anticipata dal processo del consumatore, il quale continuerà a considerare l'iniziale counter di 5 anziché il nuovo valore 6.

- Si assuma che **counter** sia inizialmente 5. Un possibile interleaving delle istruzioni è il seguente:

T	Prod: <b>register<sub>1</sub> = counter</b>	(register1 = 5)
	Prod: <b>register<sub>1</sub> = register<sub>1</sub> + 1</b>	(register1 = 6)
	Cons: <b>register<sub>2</sub> = counter</b>	(register2 = 5)
	Cons: <b>register<sub>2</sub> = register<sub>2</sub> - 1</b>	(register2 = 4)
	Prod: <b>counter = register<sub>1</sub></b>	(counter = 6)
	Cons: <b>counter = register<sub>2</sub></b>	(counter = 4)

- Il valore finale di **counter** dovrebbe essere 5, mentre in in questo caso è erroneamente pari a 4 (invertendo le ultime due istruzioni si otterrebbe erroneamente 6).

È un problema di sincronizzazione dovuto all'interleaving, che di fatto dipende dal modo in cui i due processi, produttore e consumatore, sono schedulati. Per evitare questo problema bisogna fare terminare completamente prima un processo e poi l'altro.