

Sincronizzazione dei Processi:

6. Strumenti di sincronizzazione

7. Esempi di sincronizzazione

- Introduzione
- Il Problema della Sezione Critica
- Sincronizzazione via hardware
- Semafori
- Esempi di Sincronizzazione

6.1 introduzione

- Più processi possono cooperare per compiere un determinato lavoro e, di solito, condividono dei dati.
- E' fondamentale che l'accesso ai dati condivisi da parte dei vari processi **non produca dati inconsistenti**
- processi che cooperano condividendo dei dati devono quindi **agire in modo ordinato, cioè sincronizzarsi** quando ciascuno di loro vuole accedere a quei dati
- Problema: mentre un processo P sta elaborando dati che dovranno essere usati anche da altri processi, non sa quando il SO deciderà di toglierlo dalla CPU. Altri processi non devono avere accesso ai dati condivisi fino a che la loro elaborazione da parte di P non è stata completata.

6.1 Esempio: Produttore - Consumatore con n elementi

- usiamo una variabile condivisa *counter* inizializzata a 0 che indica il numero di elementi nel buffer:

“produttore”

while counter ^{se pieno faccio nulla} *== n do nop;*
buffer[in] = nextp;
in = in + 1 mod n;
counter++;

“consumatore”

while counter ^{se vuoto faccio nulla} *== 0 do nop;*
nextc = buffer[out];
out = out + 1 mod n;
counter--;

Ma questi programmi funzionano?

6.1 Esempio: Produttore - Consumatore con n elementi

- I due programmi sono corretti se considerati separatamente, ma possono **non funzionare** quando vengono eseguiti
- il problema è **l'uso della variabile condivisa `counter`**
- che succede se **produttore** esegue `counter++` mentre “contemporaneamente” **consumatore** esegue `counter--`?
- Se **`counter`** all'inizio vale 5, dopo `counter++` e `counter--` **può valere indifferentemente 4, 5 o 6!**
$$C = C + 1;$$
- N.B.: sopra c'è scritto che *possono non funzionare*, e non che *non funzionano*. Perché facciamo questa precisazione?

6.1 Esempio: Produttore - Consumatore con n elementi

- Il problema si verifica perché *counter++* e *counter--* non sono operazioni atomiche
- guardiamo le corrispondenti (ipotetiche, ma del tutto plausibili) operazioni macchina:

produttore: *counter++*
load(registro1,counter);
add(registro1,1);
store(registro1,counter);

consumatore: *counter--*
load(registro1,counter);
sub(registro1,1);
store(registro1,counter);

imp
es

6.1 Esempio: Produttore-Consumatore

- l'esecuzione concorrente dei due processi fa sì che le rispettive istruzioni possano “mischiarsi” fra loro, producendo risultati diversi da quelli attesi. Ad esempio:

Context switch



- | | | |
|----------------------|----------------------------------|-----|
| • <u>produttore</u> | <i>load(registro1,counter);</i> | (5) |
| • <u>produttore</u> | <i>add(registro1,1);</i> | (6) |
| • <u>consumatore</u> | <i>load(registro1,counter);</i> | (5) |
| • <u>consumatore</u> | <i>sub(registro1,1);</i> | (4) |
| • <u>produttore</u> | <i>store(registro1,counter);</i> | (6) |
| • <u>consumatore</u> | <i>store(registro1,counter);</i> | (4) |

6.1 Uso di variabili e risorse condivise.

Principio fondamentale:

- Quando devono accedere e **modificare** dati condivisi i processi che usano **quei dati devono sincronizzarsi**, in modo che ognuno abbia la possibilità di eseguire **completamente le operazioni che deve compiere** sui dati condivisi prima che un altro processo abbia il diritto di accedere agli stessi dati
- Da notare che il problema non si pone se tutti i processi coinvolti nell'accesso ad un insieme di dati condivisi devono solo leggere quei dati.

6.1 esempio dalla vita reale...

- Vogliamo mantenere una certa quantità di latte nel frigorifero. Risultato: troppo latte in frigorifero!

ore	Persona A	Persona B
3:00	<i>In frigo non c'è latte</i>	
3.05	<i>Esce per il supermarket</i>	<i>In frigo non c'è latte</i>
3.10	<i>Compra il latte</i>	<i>Esce per il supermarket</i>
3.15	<i>Lascia il negozio</i>	<i>Compra il latte</i>
3.25	<i>Mette il latte in frigo</i>	<i>Lascia il negozio</i>
3.30		<i>Mette il latte in frigo... OPS!!!</i>

- Cosa non ha funzionato? Qual è la variabile condivisa in questo caso? Cosa si poteva fare per evitare il problema?

6.2 Le Sezioni Critiche

- Siano dati n processi P_1, \dots, P_n , che usano (anche) delle variabili condivise da tutti gli n processi.
- Ogni processo ha normalmente (almeno) una porzione di codice, detta **sezione critica**, in cui il processo manipola le **variabili condivise** (o anche solo un loro sottoinsieme).
- Quando un processo P_i è **dentro alla propria sezione critica** (di P_i), nessun altro processo P_j può eseguire codice della propria sezione critica (di P_j) **perché userebbe le stesse variabili condivise** (o anche solo un loro sottoinsieme)
- l'esecuzione delle sezioni critiche di P_1, \dots, P_n deve quindi essere **mutualmente esclusiva**

6.2 Le Sezioni Critiche

- Mentre P_i sta eseguendo codice dentro la propria sezione critica, potrebbe essere tolto dalla CPU dal SO a causa del normale avvicendamento tra processi.
- Fino a che P_i non avrà terminato di eseguire il codice della sua sezione critica (e dunque avrà avuto tempo di tornare in esecuzione per proseguire il lavoro nella sua sezione critica, e potrebbero volerci più context switch):
nessun altro processo P_j che deve manipolare le stesse variabili condivise potrà eseguire il codice della propria sezione critica!
- Notate che P_j può eseguire del codice, quando entra in esecuzione, ma non il codice della propria sezione critica

6.2 Le Sezioni Critiche

- **Sezione critica:** porzione di codice che deve essere eseguito senza *intrecciarsi* (nell'avvicendamento in CPU) col codice delle sezioni critiche di altri processi che usano le stesse variabili condivise
- Di fatto quindi sono le **variabili condivise a determinare le sezioni critiche.**

6.1 Esempio: Produttore - Consumatore con n elementi

“produttore”

```
while counter == n do nop;  
buffer [in] = nextp;  
in = in + 1 mod n;
```

```
load(registro1, counter);  
add(registro1, 1);  
store(registro1, counter);
```

Una sezione critica del
produttore associata alla
variabile condivisa counter

“consumatore”

```
while counter == 0 do nop;  
nextc = buffer[out];  
out = out + 1 mod n;
```

```
load(registro1, counter);  
sub(registro1, 1);  
store(registro1, counter);
```

Una sezione critica del
consumatore associata alla
variabile condivisa counter

6.2 Il Problema della Sezione Critica

13

- Consiste nello stabilire un protocollo di comportamento usato dai processi che devono usare le variabili condivise:
 - un processo deve “chiedere il permesso” per entrare nella sezione critica, usando una opportuna porzione di codice detta **entry section**
 - un processo che esce dalla sua sezione critica deve “segnalarlo” agli altri processi, usando una opportuna porzione di codice detta **exit section**

6.2 Il Problema della Sezione Critica

- Quindi un generico processo P_i contenente una sezione critica avrà la seguente struttura (fig. 6.1 modificata):



- naturalmente P_i può avere più sezioni critiche, e per ciascuna di queste ci dovrà essere una entry ed una exit section.

6.2 Il Problema della Sezione Critica

- Siano dati n processi P_1, \dots, P_n , che usano delle variabili condivise da tutti i processi.
- Una soluzione corretta al problema della sezione critica per P_1, \dots, P_n deve soddisfare i seguenti tre requisiti:
 1. **Mutua esclusione:** Se un processo P_i è entrato nella propria sezione critica ma non ne è ancora uscito (*attenzione, P_i non è necessariamente il processo in esecuzione, cioè quello che sta usando la CPU*), nessun altro processo P_j può entrare nella propria sezione critica.

6.2 Il Problema della Sezione Critica

2. **Progresso:** Se un processo lascia la propria sezione critica, deve permettere ad un altro processo P_j di entrare nella propria (di P_j) sezione critica. Se la sezione critica è vuota e più processi vogliono entrare, uno tra questi deve essere scelto in un tempo finito (*in altre parole, esiste un processo che entrerà in sezione critica in un tempo finito*)
- Osservazione: questa condizione garantisce che l'insieme dei processi P_1, \dots, P_n (o anche solo un loro sottoinsieme) **non finisca in una condizione di deadlock:** tutti fermi in attesa di riuscire ad entrare nella loro sezione critica

6.2 Il Problema della Sezione Critica

3. **Attesa limitata**: se un processo P_i ha già eseguito la sua entry section (ossia ha già chiesto di entrare nella sua sezione critica), esiste un limite al numero di volte in cui altri processi possono entrare nelle loro sezioni critiche prima che tocchi a P_i (*in altre parole, qualsiasi processo deve riuscire ad entrare in sezione critica in un tempo finito*)
- Osservazione: quest'ultima condizione assicura che il processo P_i **non subisca una forma di starvation**: non riesce a proseguire la sua computazione perché viene sempre sopravanzato da altri processi.

6.2 Il Problema della Sezione Critica

- Una qualsiasi soluzione corretta al problema della sezione critica deve permettere ai processi di portare avanti la loro computazione **indipendentemente dalla velocità relativa** a cui essi procedono (ossia da quanto frequentemente riescono ad usare la CPU), **purchè questa sia maggiore di zero.**
- **ATTENZIONE:** non potrà funzionare nulla se un processo entra nella sua sezione critica e si mette ad eseguire un loop infinito, **senza passare mai alla *exit section* della sezione critica...**

6.2 Il Problema della Sezione Critica

- Quindi, una corretta soluzione al problema della sezione critica deve garantire le proprietà di **mutua esclusione**, **progresso** e **attesa limitata**, indipendentemente dalla velocità relativa a cui procedono i processi.
- Ma **non deve preoccuparsi di cosa fanno i processi mentre sono dentro alle loro sezioni critiche.**

6.2 Il Problema della Sezione Critica

- Notate: dire che la soluzione deve essere indipendente dalla velocità relativa a cui procedono i processi significa, più tecnicamente, che:
- la soluzione non deve dipendere dal tipo di scheduling della cpu adottato dal SO (ossia dall'ordine e dalla frequenza con cui i processi vengono eseguiti)
- (purché, chiaramente, si usi un algoritmo di scheduling ragionevole, come quelli che abbiamo visto...)

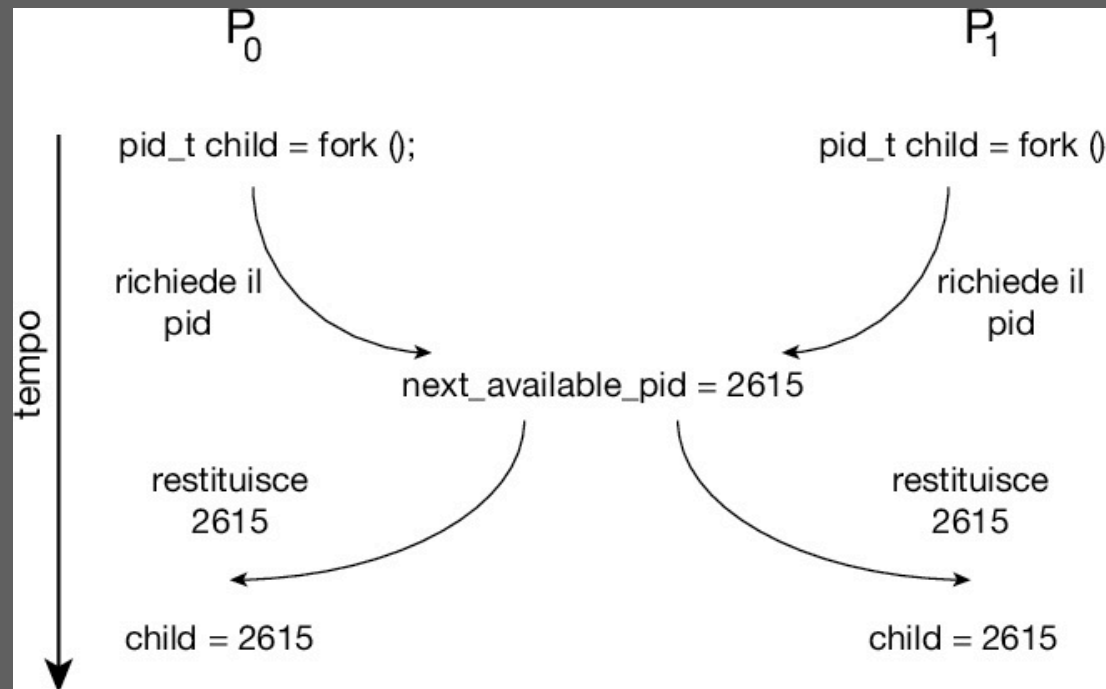
6.2 Il Problema delle Sezioni Critiche del sistema operativo

- Il problema della sezione critica **è particolarmente delicato quando sono coinvolte strutture dati del sistema operativo.**
- Ad esempio, se due processi utente eseguono una *open* sullo stesso file, vi saranno due accessi concorrenti alla stessa struttura dati del SO: **la tabella dei file aperti nel sistema.**
- è importante che questa tabella (come tutte le strutture dati del SO) **non venga lasciata in uno stato inconsistente** a causa dell'accesso concorrente dei due processi.

es
open

6.2 Il Problema delle Sezioni Critiche del sistema operativo

- Un altro esempio che coinvolge le strutture dati del Kernel è il caso di due processi che eseguono “insieme” una fork: senza opportune precauzioni, potrebbero nascere nel sistema **due nuovi processi con lo stesso PID!** (fig. 6.2)



6.2 Il Problema delle Sezioni Critiche del sistema operativo

- Il progettista del SO deve decidere come vanno gestite le sezioni critiche del sistema operativo, e le due scelte possibili sono di sviluppare un kernel *con* o *senza* diritto di prelazione.
- **In un kernel con diritto di prelazione** un processo in kernel mode può essere interrotto da un altro processo (ad esempio perché è scaduto il quanto di tempo).
- **In un kernel senza diritto di prelazione,** un processo in kernel mode non può essere interrotto da un altro processo. (secondo voi questo potrebbe essere rischioso?)

6.2 Il Problema delle Sezioni Critiche del sistema operativo

- Un kernel senza diritto di prelazione è più facile da implementare: basta disattivare gli interrupt quando un processo è in kernel mode.
- Non c'è più bisogno di preoccuparsi dell'accesso concorrente alle sezioni critiche del kernel: un solo processo alla volta può accedere alle strutture dati del kernel, perché un solo processo alla volta può essere in kernel mode.
- Se il codice del SO è scritto correttamente, la disabilitazione degli interrupt sarà temporanea e di breve durata, e tutto continuerà a funzionare normalmente.

6.2 Il Problema delle Sezioni Critiche del sistema operativo

- Del resto, i kernel con diritto di prelazione sono più adatti per le applicazioni real time, in cui la disabilitazione degli interrupt (che potrebbero essere allarmi che vanno gestiti immediatamente) non è accettabile.
- In generale, i kernel con diritto di prelazione hanno un tempo di risposta inferiore, per ovvie ragioni.
- Windows 2000 e XP erano kernel senza diritto di prelazione, mentre i loro successori sono stati tutti progettati con diritto di prelazione
- La maggior parte delle versioni recenti di Solaris, Unix e Linux sono kernel con diritto di prelazione.

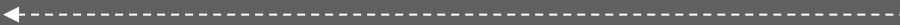
6.2 Il Problema delle Sezioni Critiche del sistema operativo

- Quindi, un kernel senza diritto di prelazione disabilita gli interrupt per il tempo necessario al codice del SO (ad esempio di una system call) per accedere in modo mutuamente esclusivo ad una qualche struttura dati del SO.
- Domanda: perché una tale soluzione non è adatta per proteggere le strutture dati condivise da due processi utente, ossia per implementare le sezioni critiche dei processi utente?

6.4.2 Sincronizzazione via Hardware

- Soluzioni semplici ed eleganti al problema della sezione critica (ma con un grave difetto, come vedremo) possono essere ottenute usando **speciali istruzioni macchina, presenti in tutte le moderne CPU** (i nomi di queste istruzioni negli instruction set di diversi processori possono ovviamente essere diversi, l'importante è ciò che fanno):
 - **TestAndSet(var1)**
testa e modifica il valore di una cella di memoria
 - **Swap(var1,var2)**
scambia il valore di due celle di memoria
 - **Importante:** sono **istruzioni macchina, e quindi atomiche:** non possono essere interrotte a metà da un context switch

6.4.2 Sincronizzazione via Hardware

- Vediamo ad esempio la TestAndSet, che potrebbe essere implementata così (fig. 6.5 modificata):
- **boolean TestAndSet(boolean *lockvariable) {**
 boolean tempvariable = *lockvariable;
 ***lockvariable = true;**
 return tempvariable;
 }
 
- Ossia:
 - salva il valore di **lockvariable* in *tempvariable*,
 - setta a true **lockvariable*,
 - restituisci il vecchio valore di **lockvariable*.

6.4.2 Sincronizzazione via Hardware

29

- Ed ecco come si può realizzare la mutua esclusione usando la testAndSet (fig. 6.6)

- Shared data: boolean lock = false;

- Processo Pi:

do {

while (TestAndSet(&lock));

sezione critica / qui dentro lock = true */*

lock = false;

sezione non critica

} while (true);

6.4.2 Sincronizzazione via Hardware

30

- Il semplice algoritmo appena visto è un esempio di soluzione al problema della sezione critica basato sull'uso di una variabile condivisa detta di **lock**.
- Si dice che **la sezione critica è controllata dal lock, e solo il processo che acquisisce il lock può entrare in sezione critica**. La struttura generale di queste soluzioni è quindi del tipo (fig. 6.10):

do {

acquisisci il lock

sezione critica

restituisce il lock

sezione non critica

} while (true)

6.4.2 Sincronizzazione via Hardware

31

- Una nota a margine: il problema della mutua esclusione è ovviamente presente anche nei **sistemi multiprocessore a memoria condivisa**, ma la realizzazione di istruzioni atomiche (come TestAndSet e Swap) è molto più complessa. Perché? (chi è interessato, può guardare il lucidi sui sistemi multiprocessore del corso di Architetture II)
- Ora verifichiamo se l'algoritmo che usa la TestAndSet garantisce le tre proprietà di una corretta soluzione al problema della sezione critica (per semplicità limitiamoci a due processi P1 e P2):

6.4.2 Sincronizzazione via Hardware

- Shared data: boolean lock = false;
- Processo Pi: do {
 while (TestAndSet(&lock));
 sezione critica / qui dentro lock = true */*
 lock = false;
 sezione non critica
} while (true);

mutua esclusione: il primo tra P1 e P2 che esegue la TestAndSet **scrive true nel lock ed entra in sezione critica**. L'altro processo trova il lock a true e deve ciclare sul while. *Ma funziona solo se la TestAndSet è atomica: cosa potrebbe succedere se si potesse verificare un context switch nel punto indicato dalla freccia di fig. 6.5?*

6.4.2 Sincronizzazione via Hardware

33

- Shared data: boolean lock = false;
- Processo Pi: do {
 while (TestAndSet(&lock));
 sezione critica / qui dentro lock = true */*
 lock = false;
 sezione non critica
} while (true);

Progresso. quando P1 lascia la sezione critica, pone
“**lock = false**”, liberando la sezione per qualche altro
processo

6.4.2 Sincronizzazione via Hardware

- Shared data: boolean lock = false;
- Processo P_i: do {
 while (TestAndSet(&lock));
 sezione critica / qui dentro lock = true */*
 lock = false;
 sezione non critica
} while (true);

Attesa limitata. NON E' GARANTITA! Infatti P₁ potrebbe uscire dalla sezione critica ("lock=false") e, sempre all'interno dello stesso quanto di tempo, tornare immediatamente a tentare di acquisire il lock, riuscendoci, e la situazione può ripetersi all'infinito! Perché un meccanismo di aging non funzionerebbe, in questo caso?

6.4.2 Sincronizzazione via Hardware

35

- Ecco la soluzione corretta per n processi

shared data boolean attesa[n], lock; // entrambi inizializzati a *false*

boolean chiave;

do {

 attesa[i] = true; // Pi annuncia di voler entrare in SC

 chiave = true;

 while (attesa[i] && chiave) chiave = TestAndSet (&lock);

 attesa[i] = false;

sezione critica

 j = (i+1) mod n;

 while ((j != i) && !attesa[j]) j = (j + 1) mod n;

 if (j == i) lock = false;

 else attesa[j] = false;

sezione non critica

} while (true);

6.4.2 Sincronizzazione via Hardware

36

- La soluzione che abbiamo visto basata sull'uso di speciali istruzioni macchina ha un problema di fondo:

while (TestAndSet(&lock));

- il processo che attende il proprio turno per entrare in una sezione critica occupata consuma CPU inutilmente. Tecnicamente, **si dice che sta facendo busy-waiting** (a volte si usa anche l'espressione “attesa attiva”).

6.4.2 Sincronizzazione via Hardware

37

- Ecco cosa succede usando il busy waiting:

questo quanto di tempo
è completamente sprecato

P1 entra nella sezione critica. Incomincia a lavorare in mutua esclusione. Scade il quanto assegnatogli. La CPU viene data a P2	P2 cerca di entrare in sezione critica, occupata da P1. P2 cicla inutilmente fino alla fine del suo quanto di tempo. La CPU viene ridata a P1	P1 termina il lavoro in sezione critica e ne esce, rilasciando il lock. Al prossimo quanto di tempo P2 potrà acquisire il lock
---	---	--

- E se invece di due processi ce ne sono N che competono per acquisire lo stesso lock, un algoritmo di scheduling round robin potrebbe produrre uno spreco del tempo di CPU pari a N-1 quanti di tempo...

6.4.2 Sincronizzazione via Hardware

38

- Una soluzione alternativa potrebbe essere di permettere ai processi utente di **disabilitare gli interrupt per tutto il** periodo in cui si trovano dentro la loro sezione critica.
- **Il problema è che in questo modo il sistema operativo perde il controllo** della macchina per un tempo arbitrario, ossia per tutto il tempo in cui un processo decide di restare nella propria sezione critica.
- Se poi il processo utente fa anche il furbo, **e** entra in sezione critica, disabilita gli interrupt e **“si dimentica” di riabilitarli quando ne esce...**
- Abbiamo bisogno di soluzioni migliori, sia del busy waiting che della disabilitazione degli interrupt...

6.6 Semafori (Dijkstra – 1965)

- Strumento di Sincronizzazione che, con l'aiuto del sistema operativo, può essere implementato senza busy-waiting
- definizione di **Semaforo** S : E'una variabile intera (per ora assumiamo che sia inizializzata a 1) **su cui si può operare solo tramite due operazioni atomiche** (attenzione! non sono implementate in questo modo) che per ora descriviamo così:

wait (S): **while** $S \leq 0$ **do** *no-op*;
 $S = S - 1$;

signal (S): $S = S + 1$;

6.6 Semafori

40

- Un semaforo può in realtà essere visto come un oggetto condiviso da tutti i processi che devono usarlo per sincronizzarsi fra loro.
- La variabile intera S viene di solito denotata col termine di **variabile semaforica**, e il suo valore corrente è detto il **valore del semaforo**.
- **Wait e Signal** sono i metodi con cui usare l'oggetto semaforo (in realtà, è anche necessaria una operazione per **inizializzare il valore** della variabile semaforica).
- La definizione non usa la terminologia della programmazione ad oggetti perché questo concetto non esisteva ancora nel 1965...

6.6 Semafori

- In letteratura i termini **Wait** e **signal** sono a volte sostituiti dalle iniziali di termini olandesi (che sono i termini originariamente usati da Dijkstra, che era olandese):
 - **P** (Proberen = verificare) al posto di Wait
 - **V** (Verhogen=incrementare) al posto di Signal
- Vengono anche usati:
 - **Down** (per la Wait, che decrementa il semaforo)
 - **up** (per la Signal, che incrementa il semaforo)

6.6.1 Uso dei semafori

42

- Adesso la soluzione del problema della sezione critica per un gruppo di processi è semplice. La variabile semaforica **mutex** (**mut**ual **ex**clusion) fa da variabile di lock:

shared variable ***semaphore mutex = 1;***

Generico processo P_i

do {

wait(mutex);

sezione critica

signal(mutex);

sezione non critica

} while (true);

6.6.1 Semafori

43

- In realtà i semafori si possono usare per **qualsiasi problema di sincronizzazione** (ne vedremo più avanti alcuni relativamente complessi).
- Ad esempio, se vogliamo eseguire una generica operazione S1 fatta dal processo P1 prima di S2, fatta dal processo P2; possiamo usare **un semaforo sync** inizializzato a 0:
- P1 esegue:
 S1;
 signal(sync);
- P2 esegue:
 wait(sync);
 S2;

6.6.1 Semafori

- **Attenzione: il nome scelto per un semaforo non ha nessuna relazione con l'uso che ne verrà fatto.**
- Un semaforo **è una variabile** (in realtà, come vedremo tra poco, è una struttura dati) che ovviamente possiamo chiamare come preferiamo.
- Naturalmente, è meglio usare nomi che ricordino l'uso che faremo di un semaforo. Quindi chiameremo *mutex* un semaforo usato per implementare una mutua esclusione, e *sync* un semaforo usato per implementare un meccanismo di sincronizzazione tra due processi.
- Ma non cambierebbe nulla se chiamassimo i due semafori rispettivamente *X* e *Y*, o anche *Pippo* e *Pluto*.
- **Ciò che importa è come li usiamo.**

6.6.2 Implementazione dei Semafori

- la definizione di wait e signal che abbiamo dato usa il busy-waiting, e questo è proprio quello che vorremmo evitare...
- **I semafori implementati attraverso il busy-waiting** esistono, e prendono di solito il nome di **spinlock** (nel senso che il processo “gira” mentre testa la variabile di lock, proprio come abbiamo visto nelle tecniche di sincronizzazione via hardware)
- **Per evitare busy-waiting** dobbiamo farci aiutare dal **Sistema Operativo, il quale mette a disposizione opportune strutture dati e system call** per l'implementazione delle operazioni di wait e signal.

6.6.2 Implementazione dei Semafori

- Quando un gruppo di processi ha bisogno di un semaforo, lo **richiede al Sistema Operativo** mediante una opportuna system call.
- Il SO alloca un nuovo **semaforo all'interno di una lista di semafori memorizzata nelle aree dati del kernel.**
- Ogni semaforo è implementato usando due campi: **valore e lista di attesa**

```
typedef struct {  
    int valore;  
    struct processo *waiting_list;  
}semaforo;
```

6.6.2 Implementazione dei Semafori

47

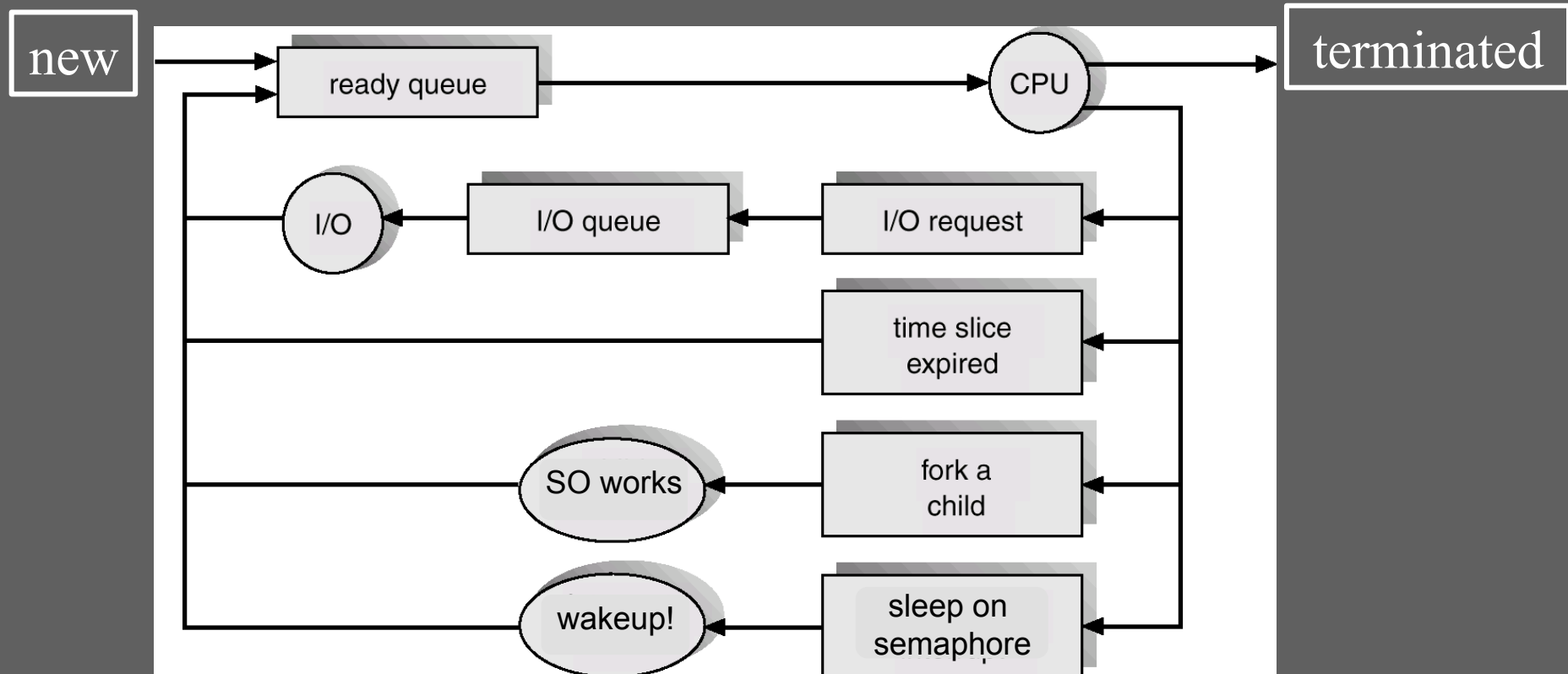
- Due system call sono disponibili per implementare le operazioni di wait e signal:
- ***sleep()***: toglie la CPU al processo che la invoca e manda in esecuzione uno dei processi in Ready Queue. Il processo che ha chiamato *sleep* non viene rimesso nella Ready Queue (N.B.: a volte la *sleep* è chiamata *block()*)
- ***wakeup(P)***: Inserisce il processo P nella Ready Queue

6.6.2 Implementazione dei Semafori

- `wait(semaforo *S) {`
 `S->valore--;`
 `if S->valore < 0 {`
 aggiungi questo processo a `S->waiting_list`
 `sleep()`;
 `}`
- la chiamata di **`sleep()`** provoca un context switch, e il processo sospeso **non consuma CPU inutilmente** (infatti il suo PCB non è più in Ready Queue, ma nella lista di attesa del semaforo su cui si è **sospeso**).
- Si dice anche che il processo si è **addormentato** sul semaforo S

3.2.1 Code di scheduling

- Riprendiamo il **Diagramma di accodamento** visto nel capitolo 3, e notiamo il caso *wait for an interrupt*, che avevamo a suo tempo ignorato: lo possiamo proprio associare al caso in cui un processo abbandona la CPU addormentandosi su un semaforo (fig. 3.4 modificata).



6.6.2 Implementazione dei Semafori

- `signal(semaforo *S) {`
 `S->valore++;`
 if `S->valore <= 0` { */* c'è qualcuno in wait... */*
 togli un processo P da `S->waiting_list`;
 wakeup(P);
 }
}
- La **Wakeup(P)** rimetti P in Ready Queue, dunque P è pronto ad usare la CPU quando sarà il suo turno. Si dice anche che P è stato **svegliato**.

6.6.2 Implementazione dei Semafori

- **NOTATE BENE**: Wait e Signal sono di solito system call direttamente messe a disposizione dal SO, **anche se a volte hanno nomi diversi**.
- Wait e Signal sono esse **stesse sezioni critiche (perché?)**
Come vengono implementate?
- di fatto, sono due sezioni critiche molto corte (circa 10 istruzioni macchina) per cui va bene implementarle con **spinlock** o **disabilitazione degli interrupt** (che avviene sotto il controllo del SO). *↳ solo single core*
- Secondo voi, quale soluzione è migliore per i sistemi monoprocessore, e quale va meglio per i sistemi multiprocessore?

6.6.2 Implementazione dei Semafori

52

- **NOTATE ANCHE:** all'inizio avevamo detto che la semantica della wait era:

wait (S): **while** $S \leq 0$ **do** *no-op*;
 $S = S - 1$;

- ma nell'implementazione mediante la *sleep* vediamo che il valore del semaforo può essere negativo. Come mai?
- Se $S \rightarrow \text{valore}$ è < 0 il suo valore assoluto ci dice quanti processi sono in attesa (in wait) su quel semaforo (andate a rivedere il codice della wait)

6.6.2 Implementazione dei Semafori

- **NOTATE ANCORA:** il valore del semaforo può anche essere un **intero maggiore di 1**. Ad esempio, se una risorsa può essere usata **contemporaneamente al massimo da tre processi**:
 - *semaphore counter = 3; // counter viene inizializzato a 3*
 - Generico processo P_i
 - repeat**
 - wait(counter);***
 - usa la risorsa
 - signal(counter);***
 - remainder section
 - until false;**

6.6.2 Implementazione dei Semafori

- Quindi, riassumendo, attraverso i semafori implementati usando *sleep()* e *wakeup(P)* i processi utente possono contenere sezioni critiche arbitrariamente lunghe senza:
 - **sprecare inutilmente tempo di CPU** (come accadrebbe se implementassimo le sezioni critiche con il busy-waiting)
 - rischiare di dare il controllo della CPU al processo (come accadrebbe se si usasse la disabilitazione degli interrupt gestita direttamente dai processi utente).
- wait e signal (esse stesse sezioni critiche) possono invece essere implementate con busy-waiting o disabilitazione degli interrupt perché il tutto avviene per poco tempo e sotto il controllo del SO

6.8.1 Deadlock & Starvation (stallo e attesa indefinita)

- I semafori sono le primitive di sincronizzazione più semplici e più usate nei moderni Sistemi Operativi, e permettono di risolvere qualsiasi problema di sincronizzazione fra processi.
- **Però sono primitive di sincronizzazione non strutturate, e sono quindi “rischiosi”**
- Usando i semafori non è poi così difficile scrivere programmi che funzionano male (formalmente, possono portare a situazioni di **deadlock o starvation**)

6.8.1 Deadlock & Starvation

- Ecco qui un esempio:

• **P 0**

① wait(S); *superato*

③ wait(Q); *wait Q (P1)*

...

signal(S);

signal(Q);

P 1

② wait(Q); *superato*

④ wait(S); *wait S (P0)*

...

signal(Q);

signal(S);

- Se S e Q sono inizializzati a 1, cosa può succedere se P0 e P1 vengono eseguiti concorrentemente?

6.8.1 Deadlock & Starvation

- **P0**

wait(S);
wait(Q);
...
signal(S);
signal(Q);

- **P1**

wait(Q);
wait(S);
...
signal(Q);
signal(S);

- Se dopo la wait(S) di P0 la cpu viene data a P1, che esegue wait(Q), e poi la cpu viene ridata a P0, P0 non può più proseguire, e quindi nemmeno P1: deadlock!

6.8.1 Deadlock & Starvation

- Il problema dei semafori è che le due operazioni di wait e signal sono indipendenti, e possono quindi essere usate in modo errato.
- Chi seguirà il corso di “Programmazione Concorrente” vedrà che esistono primitive di sincronizzazione più strutturate (**Regioni Critiche Condizionali, Monitor**), che permettono di evitare (almeno in parte) i problemi che possono sorgere con l'uso dei semafori.
- Potete leggere la sezione 6.7 del testo per farvi un'idea del concetto di Monitor.

7. Esempi di sincronizzazione: Problemi classici di sincronizzazione risolti con i semafori

- Problema dei Produttori e dei Consumatori
- Problema dei Lettori e degli Scrittori
- Problema dei Cinque Filosofi

7.1.1 Produttori-Consumatori con memoria limitata

- Usiamo un buffer circolare di SIZE posizioni in cui i produttori inseriscono i dati e i consumatori li prelevano.
- Dati condivisi e inizializzazione dei semafori:

```
typedef struct {...} item;  
item buffer [SIZE];  
semaphore full, empty, mutex;  
item nextp, nextc;  
int in = 0, out = 0;  
full = 0 ; empty = SIZE; mutex = 1;
```

7.1.1 Produttori-Consumatori con memoria limitata

- *full*: conta il numero di posizioni piene del buffer
- *empty*: conta il numero di posizioni vuote del buffer
- *mutex*: semaforo binario: per l'accesso in mutua esclusione del buffer circolare e delle variabili *in* e *out*.
- *in* e *out*: servono per gestire il buffer circolare

7.1.1 Produttori-Consumatori

- Il codice di un Produttore (fig. 7.1 modificata):

```
while (true) {  
    ...  
    produci un item in nextp  
    ...  
    wait(empty);  
    wait(mutex);  
    buffer[in] = nextp;           // inserisce nextp  
    in = (in + 1) mod SIZE; // nel buffer  
    signal(mutex);  
    signal(full);  
}
```

7.1.1 Produttori-Consumatori

- Il codice di un Consumatore (fig. 7.2 modificata):

```
while (true) {  
    wait(full)  
    wait(mutex);  
    nextc = buffer[out]           // rimuove un  
    out = out + 1 mod SIZE       // elemento  
    signal(mutex);  
    signal(empty);  
    ...  
    consuma item in nextc  
    ...  
}
```

7.1.1 Produttori-Consumatori

- Notate perché usiamo *mutex* per gestire in modo mutuamente esclusivo le variabili condivise:
- se ci sono più produttori, e *buffer* ha almeno due posizioni libere ($empty \geq 2$), ci possono essere due (o più) processi che hanno superato la $wait(empty)$ e cercano di inserire in *buffer* un item e di aggiornare la variabile *in*
- se ci sono più consumatori, e *buffer* ha almeno due posizioni piene ($full \geq 2$), ci possono essere due (o più) processi che hanno superato la $wait(full)$ e cercano di prelevare un item dal *buffer* e di aggiornare la variabile *out*

7.1.1 Produttori-Consumatori

- Come può essere semplificato il codice se possiamo supporre che esista un solo produttore?
- Come può essere semplificato il codice se possiamo supporre che esista un solo consumatore?

7.1.2 Problema dei Lettori-Scrittori

- Problema: condividere un file tra molti processi
- Alcuni processi richiedono solo la lettura (**lettori**), altri possono voler modificare il file (**scrittori**).
- Due o più lettori **possono accedere al file contemporaneamente**
- Un processo scrittore deve poter accedere al file in mutua esclusione con **TUTTI** gli altri processi

7.1.2 Problema dei Lettori-Scrittori

- Strutture dati condivise:

semaphore *mutex* = 1, *scrivi* = 1;
int numlettori = 0;

- Processo scrittore (fig. 7.3 modificata):

wait(*scrivi*);

...

esegui la scrittura del file

...

signal(*scrivi*);

7.1.2 Problema dei Lettori-Scrittori

- Processo lettore (fig. 7.4 modificata):

wait(mutex); // mutua escl. per aggiornare numlettori

numlettori++;

if *numlettori == 1* **wait(scrivi);** // **il primo lettore ferma**

// **eventuali scrittori**

signal(mutex);

... leggi il file ...

wait(mutex);

numlettori--;

if *numlettori == 0* **signal(scrivi);** // *ultimo chiude*

signal(mutex);

Lo si mette in coda

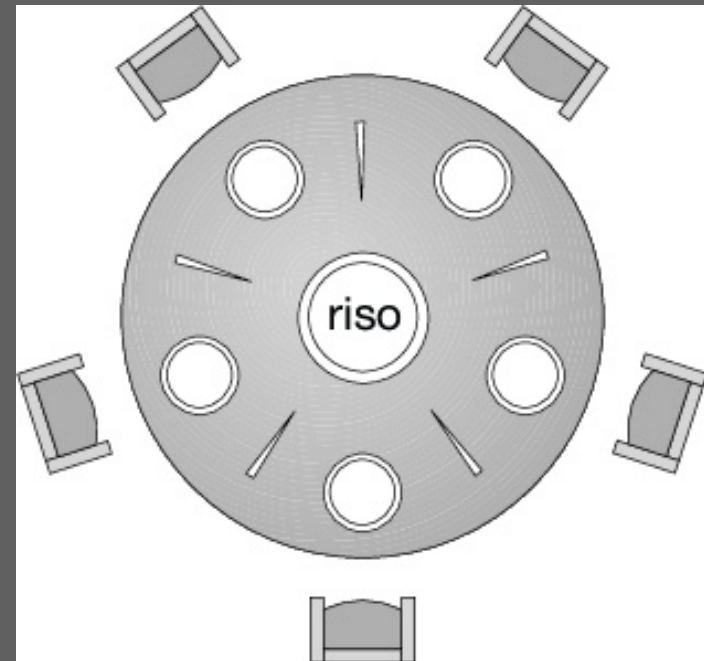
*(ma nessun altro
può entrare in mutex)*

7.1.2 Problema dei Lettori-Scrittori

- La soluzione garantisce assenza di deadlock e starvation per lettori e scrittori?
- Riuscite a pensare a soluzioni alternative, a partire da quella vista?

7.1.3 Problema dei cinque filosofi

- 5 filosofi passano la vita pensando e mangiando
- I filosofi condividono un tavolo rotondo con 5 posti.
- Un filosofo per mangiare deve usare due bacchette/risorse (fig. 7.5)



- Dati condivisi:
`semaphore bacchetta[5];` (tutti inizializzati a 1)

7.1.3 Problema dei cinque filosofi

- Ecco la soluzione “ovvia” (e sbagliata).
- filosofo i :

```

do{
    + starvation
    wait(bacchetta[i])
    wait(bacchetta[i+1 mod 5])
    ...
    mangia
    ...
    signal(bacchetta[i]);
    signal(bacchetta[i+1 mod 5]);
    ...
    pensa
    ...
}while (true)
  
```

(tutti aspettano
bacchetta destra)

nessuno
andate in
deadlock

7.1.3 Problema dei cinque filosofi

- Perché la soluzione presentata non esclude **il deadlock?**
Soluzioni migliori sono possibili:
 - solo 4 **filosofi a tavola contemporaneamente**
 - prendere **le due bacchette insieme** ossia **solo se sono entrambi disponibili**. Abbiamo bisogno di una sezione critica (perché?)
 - prelievo **asimmetrico** in un filosofo rispetto ai filosofi adiacenti
- Naturalmente, si deve anche **evitare la starvation** di un filosofo

Per chi vuole approfondire

- Sezione 6.7: i monitor
- Sulla pagina del corso:
 - Regioni critiche condizionali
 - Altri classici problemi di sincronizzazione:
 - Problema dei fumatori di sigarette
 - L'algoritmo del barbiere addormentato
 - Soluzione fair dell'algoritmo dei lettori e scrittori
 - Prima e seconda soluzione al problema dei 5 filosofi