Capitolo 6 : sincronizzazione

Table of contents

- Sincronizzazione POSIX
- Java Monitors

In questo capitolo parleremo di sincronizzazione di processi che lavorano in un ambiente di **memoria comune**; per il momento, quindi, non terremo conto dello scambio di messaggi, ne del sistema operativo.

Facciamo un'astrazione:
Abbiamo più processi che vanno in esecuzione, questi hanno un'area di memoria in comune sulla quale essi vogliono lavorare.

Questo problema è stato risolto utilizzando un **buffer circolare**, che non viene mai riempito fino all'orlo, proprio perchè le variabili **in e out** ci segnalano la memoria restante nel buffer. Quando i due puntatori sono sovrapposti, il buffer è completamente vuoto; per questo motivo possiamo utilizzare solo **BUFFER_SIZE-1** elementi, perchè se utilizzassimo tutto lo spazio, i due puntatori sarebbero nuovamente sovrapposti, ma indicherebbero che il buffer è pieno, e non vuoto.

Con questo metodo, il processo **produttore legge dal puntatore in**, e **scrive sul puntatore out**. In modo speculare, il processo **consumatore, legge il puntatore in, e modifica out.** I problemi nascono nel momento in cui entrambi i processi vogliono modificare la stessa variabile in memoria comune.
Link al capitolo precedente.

Soluzione al problema

Produttore

```
while(true){
   /*produce un item n*/
   while(counter == BUFFER_SIZE)
       ;    //do nothing
  buffer[in] = next_produced;
  in = (in + i) % BUFFER_SIZE;
  counter++;
}
```

Consumatore

```
while(true){
   while(counter == 0)
       ;    //do nothing
   next_consumed = buffer[out];
  out = (out + i) % BUFFER_SIZE;
  counter--;
}
```

Questo tipo di codice, ha senso solo nel momento in cui abbiamo diversi processi. Questo perchè abbiamo il loop principale con all'interno un ulteriore loop, che viene eseguito finché **counter** non viene modificata. Counter può essere modificata solo perchè è una variabile allocata in un'area di memoria **condivisa**, e verrà quindi modificata da un altro processo!

`Morale della favola:` purtroppo, questo metodo, non funziona.

🏁 05-13 1:42

Perchè non funziona?

Ci sono due processi che condividono un'area di memoria. Uno dei processi tende ad **incrementare** il puntatore, mentre l'altro tenderà a **decrementarlo**.

In Assembly non abbiamo **effettivamente** un'istruzione per incrementare o decrementare una variabile in memoria. Questo vuol dire che per incrementare o decrementare sono costretto ad usare (a linguaggio macchina) una **sequenza di istruzioni** del tipo:

```
load   register   val
addi   register    1
save   register   memory
```

È quindi semplice capire che come minimo abbiamo bisogno di 3 istruzioni per incrementare una variabile; cosa succede se queste tre operazioni (per incrementare) si sovrappongono alle altre 3 operazioni dell'altro processo (per decrementare)?

La prima operazione (di addizione) potrebbe perdere la CPU nel mentre sta eseguendo l'addizione, e la seconda operazione (di sottrazione) utilizzerebbe i dati nei registri ancora "sporchi" dall'operazione precedente, andando ad invalidare il risultato.

Problema della sezione critica

Consideriamo un sistema di *n* processi $\{p_0, p_1, \dots, p_{n-1}\}$

Ogni processo ha una sezione di codice detta **sezione critica**:

- Il processo potrebbe cambiare variabili comuni, aggiornare tabelle, scrivere files ecc.
- Quando un processo è in una sezione critica, nessun'altro processo dovrebbe essere nella sua sezione critica

Ogni processo deve chiedere il permesso per entrare nella sezione critica nella **entry section**.

`Morale della favola:` Ogni processo ha una porzione di codice in cui dice espressamente di voler agire per conto suo senza che nessun altro processo "gli dia fastidio"; questo perché in questo modo evita di fare danni (come nel caso dell'incremento e decremento di variabili da due processi diversi).

L'idea è la seguente:

```
do{
   //entry section
      critical section
   //exit section
      remainder section
}while(true);
```

Ad un certo punto il processo richiede l'ingresso, entra in sezione critica, e poi esce. Il tutto è racchiuso all'interno di un while true.

Soluzione al problema della sezione critica

1. **Mutual Exclusion:** Se un processo P1 è all'interno della sezione critica, nessun altro processo può eseguire la propria sezione critica.
2. **Progresso:** Supponiamo che un processo voglia entrare nella sua sezione critica. Se non ci sono altri processi che vogliono entrare nella loro sezione critica, esso non può essere ritardato.
3. **Attesa limitata** : se il processo è in attesa per entrare nella sua sezione critica, un altro processo non può "scavalcare" il processo in attesa entrando nella sua sezione critica. Indipendentemente dal timing dei processi, prima o poi ogni processo deve avere la possibilità di entrare nella sua sezione critica.
   Esiste un limite di volte in cui un processo può entrare nella sua sezione critica.

Soluzione di Peterson

Questa soluzione **vale solo per due processi**, e sopratutto è una soluzione di tipo **software**; inoltre potrebbe non funzionare sui sistemi moderni.

L'ipotesi è quella di ragionare solo su due processi, questi due quindi si contendono l'accesso alla propria sezione critica. Assumiamo che le operazioni di **load e store** in linguaggio macchino siano **atomiche** (ovvero non possono essere interrotte).

I due processi condividono due variabili:

- una variabile intera che dice di chi è il turno per accedere (turn).
- Variabile booleana, un flag che indica il desiderio di entrare nella sezione da parte dei processi (fag[2]).

```
while(true){
   flag[i] = true;
   turn = j;
   while(flag[j] && turn == j)
      ;
   /* sezione critica */
   flag[i] = false;
   /* sezione di reminder */
}
```

Nonappena il processo desidera entrare nella propria sezione critica, pone il proprio flag a **true**, manifestando quindi l'interesse ad entrare.

Successivamente, assegna la variabile turn all'altro processo, ovvero dice che il turno è dell'altro processo. Dopo questo passaggio c'è un loop che attende vedendo se anche l'altro processo vuole entrare in sezione critica; se **flag[j]** fosse **false**, il processo entra direttamente in sezione critica, e turn non viene nemmeno controllata.

Una volta terminata la sezione critica, pone il proprio flag a false, dichiarando di non voler più entrare in sezione critica

Se tutti e due i processi vogliono entrare nella sezione critica, ovvero entrambe le variabili flag sono true, "perde" chi ha salvato per primo il valore **turn**, e di conseguenza **l'altro** thread entra nella sua sezione critica (questo perchè ogni processo scrive sulla variabile **turn** che il turno è dell'altro processo).

🏁 05-14 00:33

Perchè la soluzione di Peterson non funziona sull'hardware moderno?

I processori moderni, riordinano le istruzioni e le eseguono **fuori ordine**! Di conseguenza, nel momento in cui il processore non esegue le istruzioni nell'esatto ordine in cui le abbiamo scritte, non c'è nessuna garanzia che il metodo continui a funzionare.

Questo metodo è solo un metodo "didattico", utilizzato per far capire come si può risolvere il problema, anche se essa non è una soluzione plausibile.

Come viene risolto realmente il problema della sincronizzazione?

Disattivazione delle interruzioni (sistemi "antichi")

La soluzione **classica**, utilizzata in molti SO quando il processore era singolo, era quello di **disabilitare le interruzioni**.

Supponiamo di avere un SO tradizionale (CPU singolo); un processo vuole entrare nella sua sezione critica, e quindi non vuole essere interrotto (non vuole perdere la CPU finchè non ha finito). Siccome tutti i meccanismi di commutazione da un processore all'altro, fanno utilizzo del **sistema delle interruzioni**.

Siccome tutti i processori consentono in hardware di impostare un **flag** per **disabilitare le interruzioni**, vuol dire che eventuali segnali di interruzioni vengono congelati finchè questo flag non è nuovamente abilitato.

Questa è sicuramente una soluzione valida per ottenere la **mutua esclusione** su una piccola porzione di codice. Non è decisamente, però, una soluzione gradevole; questo per diversi motivi, il primo è sicuramente il fatto che è una soluzione funzionante **solo su sistemi a singola CPU.**

Inoltre la sezione di codice da eseguire in questa modalità deve essere **molto breve**, proprio perchè in quel periodo il **SO è completamente assente**.

Sistemi moderni

Ci sono diverse soluzioni hardware adottate sui sistemi moderni:

- Barriere di memoria
- Istruzioni Hardware
- Variabili Atomiche

Istruzioni Hardware

Il metodo più utilizzato sui sistemi moderni è quello delle **istruzioni hardware**:

Se in qualche modo mi invento delle operazioni che a livello hardware sono totalmente **atomiche** e mi permettono di risolvere questo problema, invece di cercare di risolvere il problema via software, risolvo il problema via hardware.

Istruzioni hardware speciali permettono di utilizzare le istruzioni:

- **Test-and-set**
- **Compare-and-swap**

Test And Set

```
boolean test_and_set(boolean *target){
  boolean rv = *target;
  *target = true;
  return rv;
}
```

Chi progetta il processore, deve garantire di poter eseguire un'operazione simile a quella riportata sopra. Più precisamente, deve garantire che il codice venga eseguito **atomicamente**, anche in sistemi aventi processori multipli.

Il codice esamina una locazione di memoria, e quindi preleva dalla locazione un valore **booleano**, del tipo libero/occupato, dopodiché indipendentemente da quello che è stato occupato, scrive all'interno della locazione di memoria "occupato" (**true**).

Con questo meccanismo si può implementare la **sezione critica** in maniera molto semplice.

Soluzione utilizzando test_and_set()

Partiamo dal presupposto che anche su un sistema avente processori multipli, la funzione **test_and_set()** è sempre una funzione eseguita in **maniera atomica**, e questo vuol dire che se ci sono due invocazioni della funzione, da due processori diversi sulla stessa locazione di memoria, verrà sempre eseguita prima una e poi l'altra, e mai **contemporaneamente**.

```
do{
  while(test_and_set(&lock))
    ;    //do nothing

  // sezione critica

  lock = false;

  // sezione reminder
}while(true);
```

La sezione critica si può semplicemente proteggere con una **test_and_set()** che va a prelevare il valore di una variabile **lock** che originariamente è posta a **false**.

Questa variabile globale, in condizioni normali (nessun processo vuole accedere alla sezione critica), vale sempre **false**. Di conseguenza effettuiamo un **while** su questa variabile, e finchè essa è **true** (ovvero un processo vuole entrare/è dentro la sua sezione critica) non facciamo nulla.

Appena questa variabile diventa **false**, vuol dire che possiamo entrare nella nostra sezione critica.

Finita la sezione critica, la variabile **lock** viene riposta a false, per permettere ad altri processi di accedere alla propria.

il **problema** di questo tipo di soluzione, è che non è **garantita l'attesa limitata**; questo perchè un processo potrebbe richiedere (all'infinito) di entrare nella propria sezione critica, lasciando "a bocca asciutta" gli altri processi. Non c'è quindi garanzia che questa soluzione garantisca l'attesa limitata.
È possibile utilizzare la **test_and_set()** che assicuri l'attesa limitata, anche se l'implementazione è più complicata.

compare_and_swap

La **compare_and_swap()** è un'altra istruzione che alcuni processori hanno (eseguita atomicamente); questa istruzione effettua uno **swap atomico** (che non viene interrotto da altri processi).

Sincronizzazione in Risc-V (dal libro di Patterson, parte del corso su architettura)

Servirebbe un'operazione di lettura/scrittura atomica, in modo da leggere il contenuto di una variabile doc e renderlo immediatamente occupato, prima che un altro processo possa "metterci le mani".

La soluzione potrebbe essere quella di usare un'operazione singola o con una **coppia di istruzioni atomiche** (da eseguire insieme, cosa difficile da fare).

La soluzione realmente adottata sia da **RISCV che da MIPS** è quella di utilizzare delle istruzioni apposite che permettono di creare una struttura che comunque garantisca **l'atomicità**, ma utilizzando un "trucco" software.

In particolare, quello che avviene è la presenza di due istruzioni, indipendentemente dalla specifica di formato (lr**.d** rd, dove il **.d** sta per "double", ovvero un'operazione effettuata a 64bit):

Istruzione **load reserved**:

```
lr.d    rd,    (rs1)
```

Istruzione **Store conditional**

```
sc.d    rd, (rs1), rs2
```

Il **load reserved** effettua il caricamento in un registro **destinazione** (rd) a partire da una locazione di memoria indicata dal contenuto di un altro registro (rs1). Fin quì tutto normale (pari al ld).

Questo load è **reserved**, perchè il processore tiene traccia di quell'indirizzo di memoria; di conseguenza il processore si accorge se un altro processo ha scritto in quel registro prima di me, e posso continua ad effettuare il load finchè non c'è nessun altro che si "mette in mezzo" tra il load e lo store, rendendo il tutto **atomico**.

Lo **store conditional** lavora "a braccetto" con l'operazione di load. Questo perchè lo store viene effettuato solo ed esclusivamente se nessun altro, nel frattempo, ha modificato quella locazione di memoria. Se quella locazione di memoria è stata modificata, lo store non viene eseguito, e viene **ritornato** nel registro **rd**, che è un **terzo registro** usato come parametro dell'istruzione, dove viene salvato il **risultato dell'operazione**, l'esito dell'operazione.

Esempio di Atomic Swap in RISCV

```
again:  lr.d   a0,  (s4)
        sc.d   a1,  (s4),  s7          // a1 = status
        bne    a1,  zero,  again       //branch if store
        failed
        addi   s7,  a0,  0             // s7 = loaded value
```

Viene effettuato un load reserved all'indirizzo s4, dopodiché lo **store conditional** tenta di scrivere il valore di s7 all'indirizzo s4; in a1 viene ritornato lo stato. Successivamente, controllo sul registro a1 il risultato, e se **non è andato a buon fine**, salto ad **again** dove tento nuovamente di effettuare lo **store**.

Lo store avrà successo solo quando nessun altro processo modificherà quella locazione di memoria.

🏁 05-14 01:07

Soluzione Mutex locks (mutual exclusion locks)

L'utilizzo di questo meccanismo per la realizzazione della sezione critica è abbastanza frequente?

Molto spesso capita di avere dei processi che devono accedere alla sezione critica, di conseguenza è ragionevole che il SO fornisca una soluzione nativa.

Di conseguenza capiamo che le soluzioni precedenti sono complicati e generalmente inaccessibili dai programmatori di applicazioni. E' per questo motivo che i progettisti di SO hanno creato dei tool software per risolvere il problema della **sezione critica**.

la soluzione più semplice è proprio il **Mutex Lock**:

Proteggiamo la sezione critica prima effettuando un **acquire()** di un lock, e poi il **release()** del lock. Una variabile booleana indica se il **lock** è disponibile oppure no.

Le chiamate **acquire() e release()** devono essere **atomiche**, infatti sono solitamente implementate con **istruzioni hardware atomiche**, come la **compare-and-swap**.

Il problema

Ovviamente il problema più chiaro di questa soluzione è proprio l'efficienza, proprio perchè sopratutto sui sistemi a CPU singola, il processo è continuamente in un **loop** effettuando degli accessi alla memoria, proprio per controllare se la variabile **lock** è true o false.

In un sistema a CPU singola, quindi, se il processo utilizza la CPU per controllare la variabile, per forza di cose essa non può essere cambiata da altri processi, proprio perchè l'unica CPU disponibile è utilizzata dal processo in questione.

In un sistema a CPU multiple questa soluzione ha senso.

Semafori

Questa soluzione è ancora in uso tuttora, quindi abbastanza valida. Essa serve a risolvere in modo brillante il problema della **mutua esclusione** *(ovvero entrare in una sezione di codice uno alla volta)*, ma anche a risolvere problemi di sincronizzazione (ovvero il problema di *sincronizzazione su condizione, ovvero attendere finchè non è valida una certa condizione che mi abilita a fare qualcosa*).

In qualsiasi sistema operativo, è tuttora presente il supporto ai semafori, possiamo quindi chiamare le due primitive che ci permettono di eseguire operazioni sui semafori. Vengono inoltre utilizzati molto spesso nei sistemi **non real time**, dove vengono utilizzati altri sistemi di sincronizzazione.

Dovuta precisazione

Per "semaforo", in inglese "semaphore", si intende il semaforo utilizzato sulle ferrovie, ben diverso dai semafori stradali, in inglese "traffic light".

Definizione classica

Le due operazioni sui semafori, sono chiamate **P() e V()**, dalle parole olandesi che stanno per rispettivamente **wait() e signal()**.

wait():

```
wait(S){
   while(S <= 0)
      ; // busy wait
   s--;
}
```

Pensiamo ad una situazione in cui il semaforo inizialmente vale 1; se l'operazione viene eseguita atomicamente (cosa che accade), e più processi tentano di effettuare una **wait()**, un solo processo troverà il valore "1", azzera quindi il semaforo con **S--**, gli altri processi trovano 0 ed attendono.

signal():

Si rilascia la mutua esclusione con questa segnalazione; la signal semplicemente incrementa il valore del semaforo:

```
signal(S){    s++;}
```

Se voglio effettuare la mutua esclusione, basta che abbia un semaforo inizializzato ad 1, il protocollo di ingresso è chiamare un **wait()**, il protocollo di uscita è chiamara **signal()**.

####

Notiamo che...

Il semaforo non viene mai decrementato al di sotto di zero, questo perchè il codice prevede l'esecuzione di **S--** solo nel momento in cui S > 0, e quindi il semaforo vale sempre **o zero, o più di zero.**

Questo è il modo classico di definire il semaforo.

Il problema

Diamo per assodato che la **wait() e la signal()** siano operazioni eseguite atomicamente. Usando questa soluzione, non risolviamo il problema che avevamo con la **test-and-set()**, ma lo trasferiamo ad un altro livello.

Di conseguenza, per avere la mutua esclusione su queste operazioni, esse dovranno essere implementate **a basso livello**, con dei meccanismi, come la test-and-set(), che permettano alle operazioni di essere eseguite atomicamente.

Busy waiting e metodo efficiente

Il busy waiting, cosiddetta attesa attiva, è un modo di attendere che il semaforo divenga 1.

Visto che il loop viene implementato a bassissimo livello da chi programma il sistema operativo, possiamo "aspettare" in un modo più intelligente: sospendiamo il processo, e lo riattiviamo finchè la condizione S non ritorna "vera".

Ordine di arrivo

Supponiamo che un processo sia entrato nella zona critica (trovando 1), gli altri processi trovano 0, nel momento in cui viene eseguita la segnalazione, chi entra (valore 1)?

Con questa soluzione non è detto che il processo in attesa da più tempo sia il primo ad entrare in zona critica (o altro); il semaforo non prevede una **disciplina di risveglio di tipo FIFO**.

L'ordine di esecuzione dipende dal modo in cui il semaforo viene implementato.

Com'è composto il semaforo?

In generale il semaforo è composto da una variabile intera, in alcune implementazioni viene fornito un semaforo di tipo **binario**, per la sua facilità di implementazione.

Utilizzo del semaforo

Counting semaphore: valore intero avente range su un dominio non ristretto.

Semaforo binario: valore intero avente range solo tra 0 ed 1; stesso concetto del **mutex lock**

Consideriamo P1 e P2, ed abbiamo bisogno che S1 avvenga prima di S2; creiamo un semaforo **synch** inizializzato a zero:

```
P1:    S1;    signal(synch);P2:    wait(synch);    S2;
```

Implementazione di un semaforo

Bisogna garantire che due processi **non** possano eseguire **wait() e signal()** sullo stesso semaforo allo stesso momento.

I semafori possono essere implementati da una terza persona, che a noi non interessa.

Implementazione semaforo senza busy waiting

Supponiamo che il semaforo sia stato implementato con le operazioni:

- **block** : pone il processo invocando l'operazione sulla **waiting queue** appropriata
- **wakeup** : rimuove uno dei processi nella **waiting queue** ed pone il processo nella **ready queue** .

Per eseguire l'operazione in maniera atomica, chi implementa il semaforo sarà costretto ad usare una **test-and-set()** a basso livello, quindi l'attesa attiva è stata apparentemente eliminata, ma è ancora presente.

Nonostante ciò questo metodo continua ad essere efficiente, proprio perchè l'attesa attiva è presente solo per una piccola porzione di codice, ovvero quella nei "pressi" della **test-and-set()**, e quindi nella sezione critica (utilizzando i semafori) posso scrivere un codice anche esageratamente lungo.

Semafori vs spinning

Tutte le attese regolate da spinning (spin loop, ovvero loop di attesa) sono, e devono, essere tutte attese brevi, mentre tutte le attese regolate da **semafori**, non richiedono attese regolate da spin loop, e quindi l'efficienza è buona.

Implementazione effettiva

```
wait(semaphore *S){    S->value--;    if(S->value < 0){        add this process to S->list;        block();    }}signal(semaphore *S){    S->value++;    if(S->value <= 0){        remove a process P from S->list;        wakeup(P);    }}
```

In questo tipo di implementazione, il semaforo può diventare anche negativo; se ad esempio il semaforo vale -3, vuol dire che ci sono 3 processi in attesa.

Bisogna notare che questo tipo di implementazione è un'invenzione di **Silberschatz**, e solitamente i semafori sono implementati come detto precedentemente, ovvero con valori pari a zero, o maggiore di zero.

Problemi con i semafori

Cosa succede se utilizzo male i semafori?

Solitamente, per entrare nella sezione critica, pongo un **wait()** prima della sezione, ed un **signal()** alla fine; ma che succede se dimentico di chiamare **signal()** dopo aver eseguito la sezione critica?

Esiste un meccanismo ad alto livello che si accorga del mio errore? Risposta breve: no.

Per questo motivo, non bisogna pensare che i semafori siano **semplici da usare** per via della loro accessibilità; per questo motivo, dopo gli anni 70 si è iniziato a pensare a dei nuovi meccanismi, più facili da usare, per risolvere i problemi di sincronizzazione.

Conclusioni

I semafori sono quindi sicuramente una soluzione per la sincronizzazione più ad alto livello della **test-and-set()** vista precedentemente, ma non è la soluzione perfetta.

Monitors

L'astrazione ad alto livello fornisce un meccanismo conveniente ed efficace per la sincronizzazione tra processi; purtroppo molti anni fa la maggiore astrazione possibile era quello dell'**abstract data type**, ovvero variabili interne accessibili solo da codice all'interno della procedura.

Un solo processo alla volta può essere attivo all'interno il monitor. L'obbiettivo del **monitor*** è quindi quello di fornire un ulteriore livello di astrazione, che assicuri una programmazione più agevole mediante il tipo di dato astratto.

```
monitor monitor-name{
    // variabili shared
    function P1(...){...}
    function P2(...){...}
    function Pn(...){...}

    initialization code(...) {...}    //inizializzare le variabili shared
}
```

Pensiamo al monitor come un oggetto che offre una memoria comune a dei processi, che invocano il monitor stesso per utilizzarla.

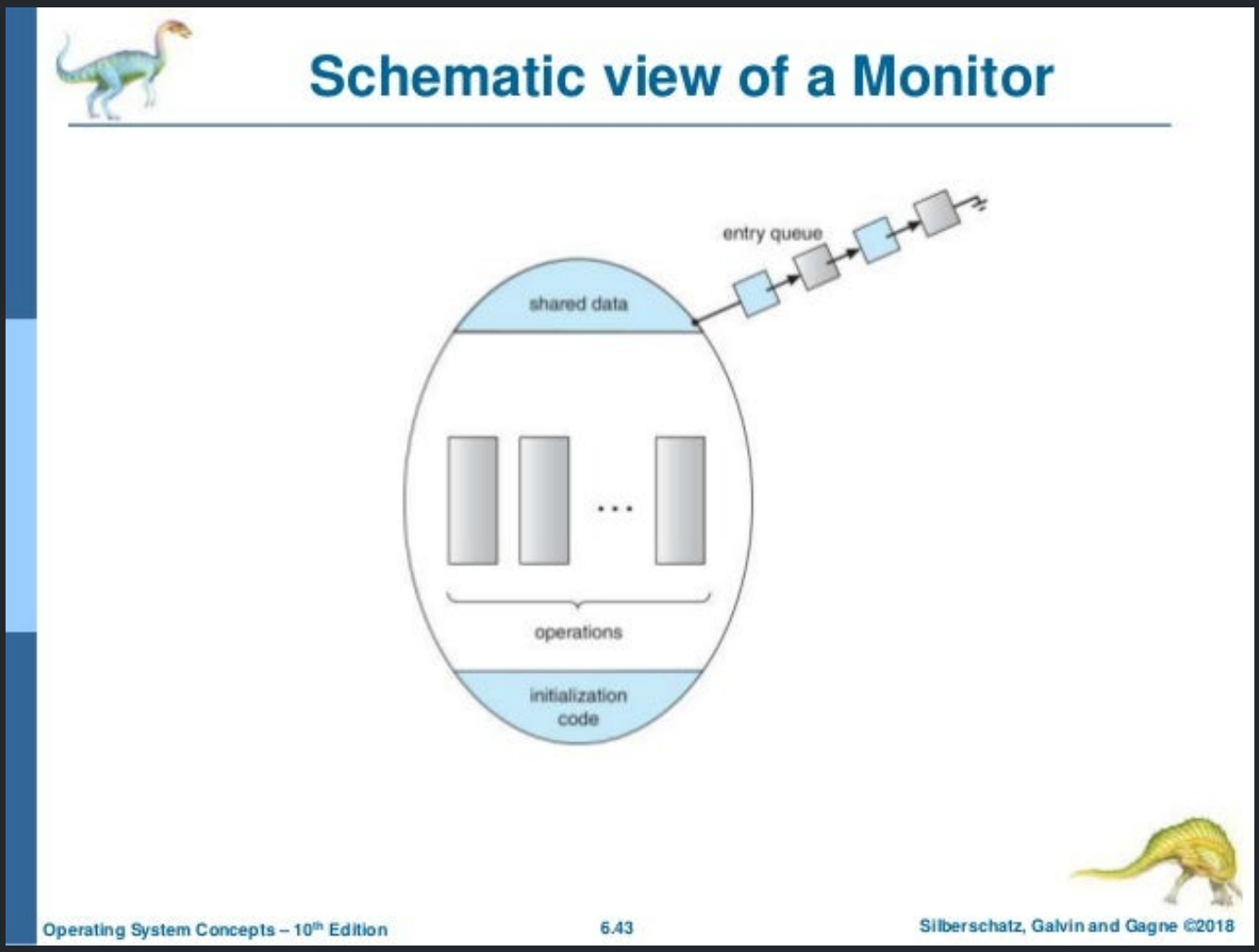
Ad esempio se dovessimo creare un **buffer** condiviso, non andiamo direttamente a salvare e prelevare dal buffer, ma usiamo un **monitor** che gestisce il buffer, e richiamiamo solo le funzioni che lavorano sulla memoria, per salvare e prelevare.

L'implementazione del monitor mi assicura che con il suo utilizzo venga rispettata la **mutua esclusione**; il programmatore non deve più preoccuparsi della mutua esclusione, deve semplicemenete invocare le funzioni del monitor, ed esse verranno eseguite una per volta.

Il problema principale del monitor

Pensiamo a processi che vogliono semplicemente **leggere** i contenuti della memoria. Con il monitor non è possibile; questo perchè la funzione di lettura viene eseguita, anche in questo caso, una alla volta.

Se ho un processo che modifica l'area di memoria, e tanti processi che vogliono solo leggere (e quindi essere eseguiti concorrentemente, visto che leggendo non si fanno danni, ma solo scrivendo in memoria), questi non vengono eseguiti concorrentemente, ma uno alla volta (perdendo molto tempo).



All'interno del monitor (rappresentato dall'ovale) è presente **l'area dei dati condivisi**; le operazioni del monitor (**metodi che possiamo invocare dall'esterno per lavorare sui dati shared**); infine un **codice da inizializzazione**.

La **entry queue** indica che nel monitor i processi entrano **uno alla volta**, e se ci sono più richieste è presente una coda di processi in ingresso, e quindi i processi che sono in attesa potranno entrare all'interno del monitor uno alla volta.

Variabili Condition

condition x, y;

Servono ad implementare all'interno di un monitor la **sincronizzazione su condizione**; sono **interne al monitor**, e quindi non visibili all'esterno. Per effettuare operazioni su queste variabili sarà necessario quindi utilizzare un eventuale metodo del monitor stesso.

Operazioni ammesse sulle variabili condition

Sulle variabili condition sono ammesse delle operazioni chiamate **x.wait() e x.signal()**;

ATTENZIONE! queste due operazioni sono completamente diverse da quelle viste per i **semafori**.

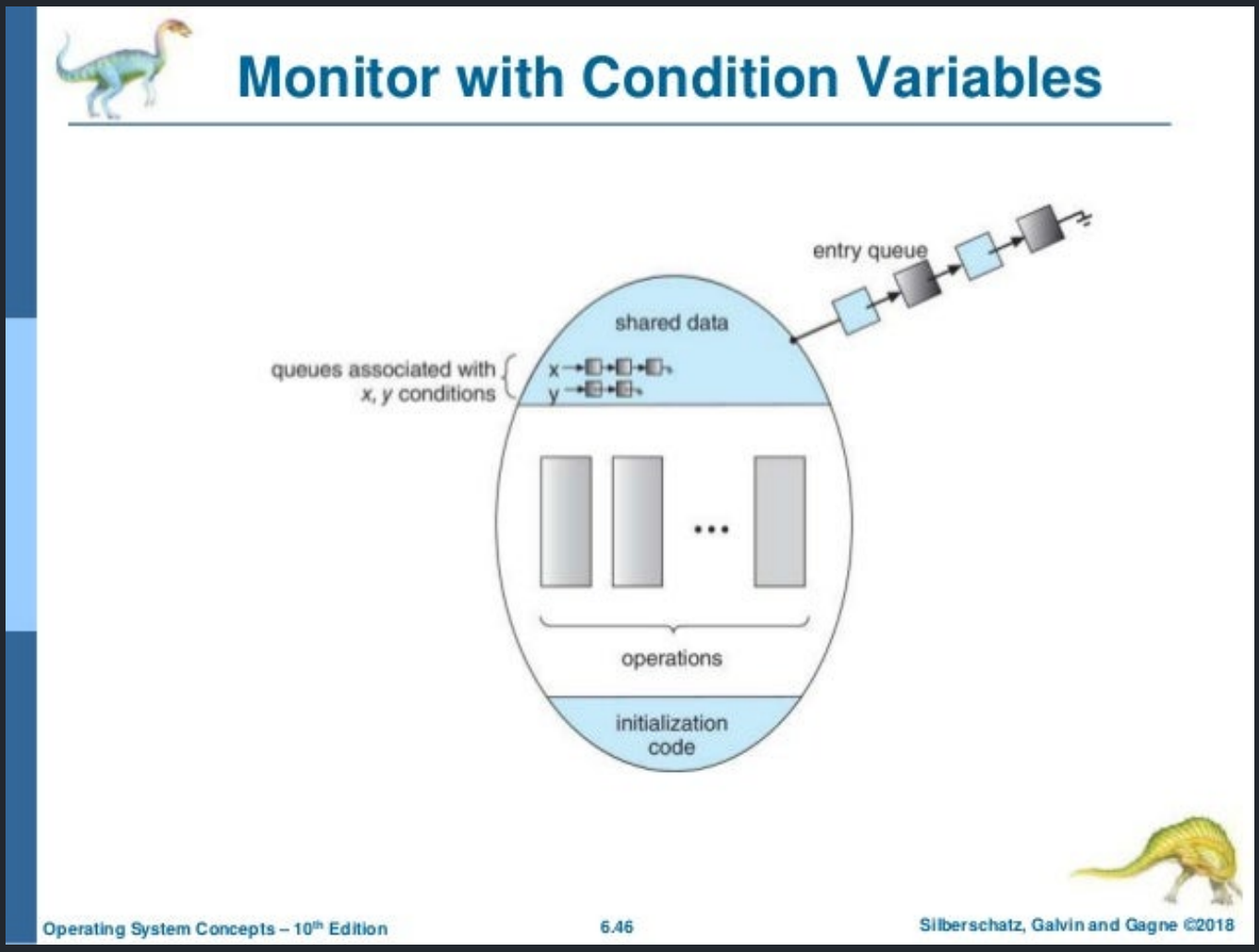
x.wait() è un modo formale di invocare l'operazione di **wait()** su una variabile condition che ho **predichiarato** chiamata x. Se un processo in esecuzione nel monitor richiama una funzione x.wait() su una variabile condition, il processo viene **sospeso.** La wait delle variabili condition, quindi, sospende irrevocabilmente il processo che la esegue.

x.signal() nel momento in cui viene eseguita va a vedere la **coda di processi sospesi sulla variabile condition x**; quindi, associata ad ogni variabile condition è presente una **coda di processi sospesi.**

Capiamo quindi che la tecnica usata per il risveglio è di tipo **FIFO**. Quando chiamiamo **signal()** viene risvegliato il processo che è in attesa **da più tempo**, cosa non garantita per i **semafori**.

Qualora non ci fossero processi sospesi su quella variabile condition, la signal() non ha effetti. Questo non accadeva con i semafori, visto che in quel caso sia se erano o non erano presenti processi in attesa, la variabile veniva sempre incrementata.

Monitor con variabili condition



In questa immagine notiamo le code associate alle variabili condition x e y. Questi saranno risvegliati nell'ordine in cui sono arrivati.

Reminder: nel monitor entra un processo per volta.

Situazione 1:

- Siamo dentro il monitor; una delle operazioni invoca una **wait()** , perchè ad

esempio voglio prelevare qualcosa da un buffer, ma è vuoto e devo sospendermi (processo).

- Il monitor è quindi libero ed un altro processo può entrare.

Situazione 2:

- Pensiamo al processo che chiama la signal().
- E' arrivato un processo e ci sono dei processi in sospeso perchè dovevano effettuare un'operazione ma non era ancora il momento. (la sospensione su condizione è proprio questo, ovvero un processo che attende di eseguire un'operazione finchè non è il momento)
- Il processo risvegliante esegue un'operazione ed esegue una segnalazione; effettuare una signal significa che uno dei processi in attesa sulla coda di x, può andare in esecuzione.

Ovviamete il nuovo processo **non può essere eseguito insieme al processo precedente!** Bisogna fare una scelta:

- Chi ha fatto la signal si sospende ed attende di essere risvegliato in un secondo momento. Il processo che è stato risvegliato riprende l'esecuzione immediatamente. Questa tecnica si chiama **Signal And Wait** .
  Ovviamente il processo che ha chiamato la signal() non va a finire nella **entry queue** , siccome era già all'interno del monitor, quindi "salta la fila".
- **Signal and continue:** Questa tecnica è quella più utilizzata (anche da java). Il segnalatore procede ed il processo risvegliato (in attesa) viene posto nella **entry queue** , ma andando sulla coda non è necessariamente il primo!

Riprendere un processo con un Monitor

Se diversi processi sono in coda su una variabile condition x, e viene eseguito **x.signal()**, quale processo deve essere ripreso?

A volte i processi possono avere delle **priorità**: ad esempio posso assegnare priorità alta ai processi **real time** per far sì che essi possano essere eseguiti per primi.

Con i semafori questo meccanismo non è possibile.

Capitolo 7: esempi di sincronizzazione

Problema del buffer limitato

Questa soluzione prevede l'utilizzo di 3 semafori in maniera "creativa":

C'è un semaforo che viene utilizzato esclusivamente per la mutua esclusione, chiamato **mutex**, inizializzato al valore 1.

Gli altri due semafori sono simmetrici: il primo, **full** inizializzato a 0, conta il numero delle posizioni "piene", mentre l'altro, **empty** inizializzato al valore n, conta i posti vuoti.

Struttura del processo produttore:

```
while (true) {
   // produce un item in next_produced
   wait(empty);
   wait(mutex);

   // aggiunge il prossimo prodotto al buffer

   signal(mutex);
   signal(full);
}
```

Simmetricamente, il consumatore (colui che preleva dal buffer), attende se full == 0, e dopo essere entrato segnala incrementando empty.

Problema Readers-Writers

Supponiamo di avere una struttura dati, o una variabile, che deve essere condivisa da più processi; alcuni di questi processi vogliono apportare delle modifiche, come ad esempio aggiornare il valore della variabile, mentre altri volgiono semplicemente leggerlo. Come gestiamo questo problema di sincronizzazione?

La soluzione potrebbe essere molto semplice: non mi interessa se sei lettore o scrittore, faccio operare tutti i processi in **mutua esclusione**, risolvendo così il problema. Questa soluzione funziona, ma perchè impedire a processi che vogliono solo leggere di leggere allo stesso momento?

Vorrei quindi un meccanismo leggermente più flessibile della mutua esclusione, che permetta ai lettori di sovrapporsi tra di loro.

Un po' tutti i sistemi operativi dispongono di **particolari primitive** che consentono proprio l'accesso in **lettura**, bloccando solo i processi che effettuano delle scritture.

La soluzione

L'idea è quella di trattare tutti i lettori come se fossero un unico processo, proprio perchè in realtà, se è presente un solo lettore, o molti, all'atto pratico non cambia nulla. La mutua esclusione va fatta tra il gruppo dei lettori, ed uno degli scrittori (sempre processi).

Cosa utilizza?

- Un semaforo **mutex** inizializzato ad 1 usato solo per la mutua esclusione, per essere sicuri che quando lavoriamo su **read_count** ci sia la mutua esclusione.
- **red_count** una variabile condivisa tra tutti i lettori ed inizializzata a 0
- Un altro semaforo **rw_mutex** inizializzato ad 1 che permette di gestire le attese tra lettori ed il gruppo degli scrittori.
  Questo semaforo tratta **tutti** i lettori come un unico processo (guardare il codice del lettore per capire meglio).

Struttura di un generico scrittore

```
while (true) {
    wait(rw_mutex);
    ...
    // azioni dello scrittore
    ...
    signal(rw_mutex);
}
```

Lo scrittore deve solo richiedere un accesso in mutua esclusione, siccome deve essere l'unico ad accedere alla variabile, senza altri scrittori o lettori.

Struttura del lettore

```
while (true) {
    wait(mutex);
    read_count++;
    if(read_count == 1)
        wait(rw_mutex);
    signal(mutex);

    ...
```

```
    //
    ...

    wait(mutex);
    read count--;
    if(read_count == 0)
        signal(rw_mutex);
    signal(mutex);
}
```

Il lettore è più complicato, infatti la logica è la seguente:

il primo lettore, supposto non ci siano lettori in attività, deve fare una wait() come lo scrittore, in modo da assicurarsi i diritti di accesso esclusivi. Successivamente al primo possiamo aggiungere n lettori, e li tratto come un **gruppo**.

Per sapere quanti lettori ci sono in attività, usiamo la variabile **read_count**, che conta il numero dei lettori, ed è una variabile **shared**, quindi per incrementare e decrementare la variabile devo usare la mutua esclusione.

Per essere sicuro di essere in mutua esclusione utilizzo un altro semaforo, **mutex**. Quindi effettuo una wait() sul semaforo **mutex**, e quando ho l'accesso incremento il **read_count**.

Dopo aver incrementato, controllo se il **read_count** è uguale ad 1; se è uguale ad 1, quindi sono il primo lettore, devo effettuare una wait su **rw_mutex (semaforo)** per assicurarmi l'accesso. Appena ho accesso su rw_mutex rilascio con una signal **mutex**,

A questo punto posso leggere!

`Cosa succede se in questo momento arriva un altro lettore?`
Se in questo momento arriva un altro lettore, incrementa **read_count** che diventa 2, quindi salta la wait su **rw_mutex** e va direttamente alla lettura della variabile in memoria, dopo aver rilasciato **mutex**.

`Uscita` Dopo aver fatto la lettura devo ridurre il numero di lettori in attività, devo quindi richiedere la mutua esclusione per scrivere sulla variabile condivisa, se **read_counter** è uguale a zero, vuol dire che il processo che sta uscendo era anche l'ultimo dei lettori, ed in questo caso deve effettuare una **signal()** per rilasciare **rw_mutex**. Infine rilascia anche **mutex**.

Problemi della soluzione - Starvation

Con questa soluzione, viene data una priorità ai **lettori**: supponendo che arrivi sempre un nuovo lettore a leggere sulla variabile condivisa, questi si aggiungeranno uno dopo l'altro, e lo **scirttore** attenderà per sempre.

Dobbiamo quindi avere una disciplina più **flessibile**, che gestisca lo scheduling ed alterni **lettori e scrittori** per evitare una possibile **starvation** (ovvero l'attesa infinita da parte degli scrittori).

`Morale della favola:` Questa soluzione è brillante e quasi funzionante, ma non è la soluzione adatta al problema.

Problema dei Filosofi che mangiano

Ci sono 5 filosofi seduti attorno ad un tavolo, che passano la loro vita a pensare ed a mangiare, senza fare nient'altro. Al centro del tavolo è presente un piatto di riso, e tra ogni filosofo e quello adiacente è presente **una sola** bacchetta cinese per filosofo.

Se un filosofo vuole mangiare, è costretto a sincronizzarsi con il vicino di destra e di sinistra, proprio per poter utilizzare entrambe le bacchette.

Questo diventa quindi un problema di **esclusione a coppie**.

La soluzione

Una soluzione che risolve il problema, fa utilizzo di un semaforo per ogni bacchetta cinese, inizializzato ad 1.

Ogni filosofo deve quindi effettuare una **wait()** sulla bacchetta sinistra e destra, ed aspettare quindi che entrambe siano disponibili.

```
while (true){
    wait(chopstick[i]);
    wait(chopstick[ (i+1) % 5]);

    //il filosofo magna

    signal(chopstick[i]);
    signal(chopstick[ (i+1 % 5)]);

    //pensa per un po'
}
```

> codice dell'i-esimo filosofo

Purtroppo la soluzione **non funziona**.

Che problema ha la soluzione proposta?

Il problema è che se tutti tentano di mangiare più o meno contemporaneamente, tutti i filosofi effettueranno una wait sull'i-esima bacchetta, questo vuol dire che tutti riusciranno ad ottenere la mutua esclusione sulla bacchetta di sinistra, e tutti vogliono anche la bacchetta di destra, che non sarà presente per via del fatto che sono già tutte "occupate".

Diventa quindi una situazione di stallo, in gergo **deadlock**. Non possiamo quindi accettare la soluzione.

Come possiamo migliorare la soluzione?

Basta semplicemente che uno dei 5 filosofi inverta l'ordine di accesso alle bacchette.

Se i primi 4 richiedono la mutex prima sulla bacchetta di Sx e poi quella di Dx, ed il quinto invece richiede la mutex prima su quella di Dx, il problema non si verifica.

Ammesso che tutti richiedano l'accesso contemporaneamente, il quinto non riuscirà ad avere la mutex ne sulla bacchetta di Sx ne su quella di Dx.

In questo modo tutti i filosofi riescono a mangiare a **rotazione**, basta solo inserire una asimmetria.

Varie implementazioni di sincronizzazione su sistemi operativi noti

Alla fine del capitolo 7 sono presenti diversi esempi di sincronizzazione su sistemi operativi noti, che il professore salta.

Sincronizzazione POSIX

POSIX dispone di:

- mutex locks
- semafori
- variabili condition

Java Monitors

Java da la possibilità di creare dei **threads**; se un metodo viene dichiarato **synchronized**, il thread chiamante possiede il **lock** per l'oggetto, in parole povere, il metodo viene eseguito in mutex.

Inoltre, è possibile utilizzare due operazioni simili a **wait() e signal()** dei monitor; questo significa che ogni oggetto java ha un mutex definito internamente, ed un variabile **condition**.

```
public synchronized void insert(E item){
    while(count == BUFFER_SIZE){
        try{
            wait();    // corrispondente wait
        }
        catch(InterruptedException ie){}
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count ++;

    notify();           //corrispondente signal
}
```