30-03-2023

Comunicazione fra processi

- In certi casi **l'output di alcuni processi diventano input di altri processi** (***pipeline***)
  - *Un processo non può andare avanti finchè non ha il risultato da un altro processo*

> *Si deve bypassare il problema dell'ipotesi di **introduzione di interrupt**.*

Per costruire un'efficiente scambio di dati fra processi si devono attenzionare (*vale anche per i thread*):

- come **scambiare i dati** fra processi (si possono avere ***parti di memoria condivise*** fra processi)
- evitare **l'accavallamento di operazioni sulle parti di memoria condivise**
- **sincronizzazione**: le operazioni devono essere **COORDINATE**.

> *Queste problematiche si incontrano nei compiti svolti dal kernel*

Corse critiche (race conditions)

Fenomeno in cui due o più processi leggono o scrivono dati su ***RISORSE CONDIVSE***. Il risultato dipende da **CHI** e **QUANDO** ha eseguito l'operazione

Esempi

1. versamenti su conto corrente: in caso di versamento in corso e improvvisa interruzione (*operazione non completata*) potrebbe creare problemi. In caso di ripartenza, l'operazione riprende esattamente da dove si era fermato. Si deve realizzare un "*versamento senza interruzioni*"
2. operazioni di kernel. Ci sono 2 approcci:
   1. **Kernel preemptive**: consente a un processo di poter **essere prelevato**, **stoppato** e **rimosso**. ("*Basta, ora ti fermi e do spazio ad altro!*")
      - Visto che c'è un tempo prestabilito, un processo può essere prelevato e rimosso nel **momento preciso** di manipolazione di una variabile globale (*condivisa*)
   2. **Kernel non-preemptive**: **NON** consente il prelevamento di un processo ma è il **processo stesso che rilascia** in maniera volontaria **l'esecuzione.**
      - In questo caso (a differenza del caso precedente (che introduce race conditions) **NON** c'è il pericolo che due processi si accavallano perchè è il processo stesso rilascia la CPU quando termina le sue operazioni

Soluzione

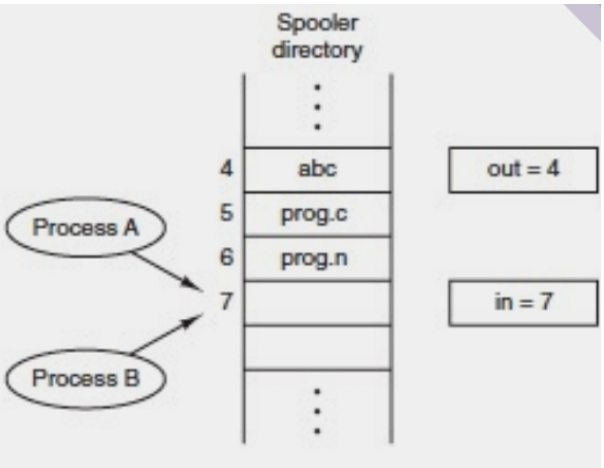
Il problema ***RACE CONDITIONS*** viene risolto con il principio della ***MUTUA ESCLUSIONE***: in ogni singolo istante, **un solo processo alla volta può accedere alla risorsa condivsa** e **può uscire** dalla parte condivisa solo **dopo aver completato l'attività**.

- *Se **SOLO UN PROCESSO** sta lavorando sulla parte conivisa, allora quest'ultima non si può occupare.*

Esempio pratico: Spool di Stampa (*stampante condivisa*)

E' un processo demone, cioè ogni tanto si sveglia, opera e poi si addormenta.

Nella **DIRECTORY DI SPOOL** vengono memorizzati i nomi dei file che devono essere stampati e questo rappresenta il mezzo di comunicazione fra processi.



`out` : indica la prima posizione del primo processo che deve andare in stampa (*file successivo da stampare*)
`in` : prima posizione vuota, dove si deve scrivere il file che ha richiesto di essere stampato (***RISORSA CONDIVISA***)

Nel caso in cui **due processi richiedono la stampa**: *Processo A e B*:

- *A* accede, legge `in` = 7 e subito dopo viene **INTERROTTO**
- *B* legge `in` (non ancora modificato da *A*) e stampa nella posizione successiva di in (*che viene giustamente incrementata*) **DOPO AVER INTERROTTO A**
- *A* riprende e ha il **VECCHIO VALORE** di `in` che valeva 7 e **riscrive B** e si ha una **sovrascrittura**
- Il risultato (*output*) di *B* viene ***eliminato***

> In definitiva, **mentre A sta lavorando su `in` allora non può essere interrotto da B finchè esso non completa le sue operazioni**

- Se i processi devono **SOLAMENTE LEGGERE**, allora il *problema non si pone*.
- Se i processo **LEGGONO e SCRIVONO**, allora il *problema è presente* e bisogna garantire la **mutua esclusione** (come nei lock delle transazioni in *Basi di Dati* -> `read_lock` e `write_lock`)

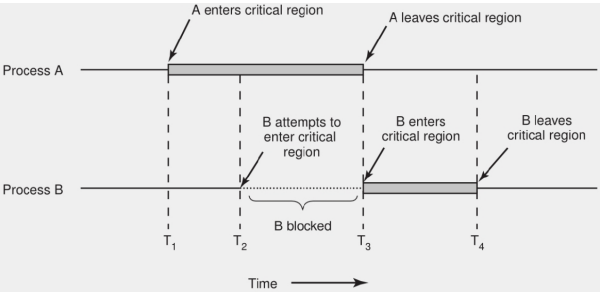
Regione/sezione critica

Non tutti i processi dello stesso programma agiscono sulle risorse condivise. Si può avere, per *esempio*:

- un blocco di codice che prende **input** da tastiera
- un altro che li sovrasscrive su un file letto da un secondo processo (***RISORSA CONDIVISA***)
- un altro processo stampa a video

Con la **REGIONE / SEZIONE CRITICA** ci si focalizza sull'insieme di istruzioni del processo che agiscono sul file condiviso.

> *Un processo **non deve entrare** nella regione critica mentre **un altro processo è presente nella stessa regione critica**.*



Un processo ha una `SEZIONE CRITICA` (*parte di processo che racchiude **le istruzioni che lavorano sulla parte condivisa***) e una `SEZIONE NON CRITICA`.

Soluzione race conditions

Si devono soddisfare **contemporaneamente** le seguenti **CONDIZIONI**:

1. **MUTUA ESCLUSIONE**: garantire che se la **sezione critica è occupata**, **nessun altro processo può accedere** alla stessa sezione critica
2. **nessuna assunzione** sulla ***VELOCITA' DI ESECUZIONE*** oppure sul ***NUMERO DI CPU***: non si fanno i conti sul tempo di esecuzione di un processo
3. se un processo *A* legge le informazioni da tastiera, *B* può accedere alla sezione critica di *A*:
   - **nessun processo** che è **FUORI** dalla sua sezione critica (*A*) può **bloccare un altro processo**
4. **nessun processo** sta in **ATTESA INFINITA** per entrare nella sua sezione critica

Modi di realizzazione della mutua esclusione

1. In caso di macchina con **MONOPROCESSORE** (in un **singolo istante** c'è un **SINGOLO PROCESSO** in esecuzione) si possono **disattivare gli interrupt**. Verranno riabilitati all'uscita dalla sezione critica di quel processo.
   - In questo caso, un processo utente gestisce gli interrupt non è una scelta del tutto saggia. Non si hanno garanzie sulla riattivazione degli interrupt.
   - In linea teorica *questo metodo funziona* ma è *molto rischioso*
   - In caso di **MULTIPROCESSORE**, gli interrupt vengono disabilitati sul **SINGOLO PROCESSO** che accede alla propria sezione critica **MA NON SUGLI ALTRI**
   - Generalmente, gli interrupt devono essere gestiti dal **KERNEL** (dal *sistema operativo*)
2. **Gestione a livello SOFTWARE**: usare **variabili di lock**:
   - può avere solo 2 valori: **0** = accesso **libero** , **1** = accesso **negato**
   - Ogni volta che un processo deve accedere alla sezione critica, va a leggere la variabile di lock e in base al suo valore entra oppure no.
   - Se si **entra** nella sezione critica, allora la **variabile di lock** viene cambiata in **1**
   - Quando il **processo termina** le proprie operazioni ed esce dalla propria sezione critica, la **variabile di lock** va cambiata e **diventa 0**
   - Anche questa soluzione riscontra il problema visto nella stampa.
     - Un processo A legge lock = 0 (non è un'operazione ATOMICA perchè può essere interrotta). Accede alla sezione. Non riesce a completare la commutazione di lock e avviene un interrupt (visto che comunque sono attivi)
     - Un processo **NON deve essere interrotto** se non ha ancora completato le sue operazioni (*come nell'esempio soprastante*)

Alternanza stretta

```
int N=2
int turn

function enter_region(int process)
    while (turn != process) do
        nothing

function leave_region(int process)
    turn = 1 – process
```

- Se un processo vuole accedere a una sezione critica già occupata, rimane sempre in **ATTESA ATTIVA** (while) e questo metodo è detto **BUSY WAITING**: si ha l'azione di testare continuamente una variabile (`turn`) finchè da essa non si ha il valore desiderato.
- Il continuo test *stressa la CPU* e la occupa di conseguenza.

> In caso che la variabile da verificare attivamente è un **lock**, si parla di ***SPIN LOCK***

L'alternanza stretta **è efficace** quanto **l'alternanza è breve**. Se un processo è più lento rispetto ad un altro, allora l'approccio non è più efficiente e si viola la **condizione 3**. (nessun processo può bloccare un altro processo se il primo è fuori dalla propria sezione critica)

> **SE SOLO IL PROCESSO 0** entra nella sua sezione critica, in questo esempio, può entrare esclusivamente una volta perchè turn vale 1 e il processo è 0. Affinchè il processo 0 rientri nella propria sezione critica dovrebbe accadere che l'1 dovrebbe accedere alla propria sezione critica così da resettare a 0 la variabile `turn`. In questo caso si viola la condizione 3, cioè un processo blocca un altro processo se esso è fuori dalla sua sezione critica.

Soluzione di Peterson

```
int N=2
int turn //variabile condivisa
bool interested[N] //contiene N processi. Inizialmente sono tutti falsi o 0

function enter_region(int process)
        other = 1 – process //gli altri(other) processi
        interested[process] = true
        turn = process
        while (interested[other] = true and turn = process)
                do nothing //l'altro non deve aver espresso l'intenzione di accedere, SOLO
IO(interested[process]) L'HO ESPRESSO

function leave_region(int process)
        interested[process] = false
```

Questa soluzione presenta 2 **aspetti negativi**:

- ogni processo che aspetta il proprio tuno, lo aspetta in ***MANIERA ATTIVA*** (pesa sulla *CPU*) -> **BUSY WATING**
- quando si lavora su sistemi multicore/multiprocessore.
- quasi simultaneamente più di un processo chiama `enter_region()`.
  - In questo caso sembra che compaia un problema ma effettivamente non è cosi
  - Il primo processo che entra è quello che si acaparra l'utilizzo della CPU perchè comunque è garantita la proprietà "*In un singolo istante è presente solo un processo*"

Istruzioni TSL e XCHG

Sono approcci che richiedono **supporto** da parte dell'**HARDWARE**.
TSL(*Test and Set Lock*): (operazione **ATOMICA** cioè non si può interrompere e ***blocca l'accesso al bus di memoria***)

- controlla il valore del lock, lo copia in un registro e lo incrementa

In particolare:

- vede se il lock era 0:
  - se era 0, lo **imposta** a 1, **accede** alla sezione critica
  - se era diversa da 0, **rimane in attesa** che il lock torni a 0
- Offre un **vantaggio** perchè sono operazioni indivisibili: cioè nessun'altro processo può accedere alla parte di memoria finchè non si sono terminate le operazioni
- quando entra nella sezione critica, **disabilita i bus della memoria** e quindi non c'è nessun'altra comunicazione e questo garantisce che se ci sono più core/processori, nessun altro processo può accedere alla risorsa condivisa
- fa uso della variabile **LOCK** che garantisce l'accesso/il non accesso alla parte condivisa

```
enter_region:
        TSL REGISTER,LOCK //copia nel registro il valore di lock e poi si deve incrementare
        CMP REGISTER,#0 // copia il valore 0 in REGISTER
        JNE enter_region // se NON si verifica la condizione (JNE) avviene un salto -> BUSY
WAITING
        RET

leave_region:
        MOVE LOCK,#0 //assegno 0 a LOCK
        RET
```

Anche in caso di utilizzo di **TSL** si ha **BUSY WAITING**, cioè il processo che non può entrare attende continuamente.

`enter_region` viene chiamata *PRIMA* di entrare nella sezione critica
`leave_region` viene chiamata *DOPO* aver finito le operazioni nella sezione critica

```
enter_region:
        MOVE REGISTER,#1
        XCHG REGISTER,LOCK //scambio i valori fra register e lock
        CMP REGISTER,#0
        JNE enter_region //iterazione se la condizione NON è verificata
        RET

leave_region:
        MOVE LOCK,#0
        RET
```

Con **XCHG** si ha sempre **BUSY WAITING**

L'idea è quella di creare **approcci alternativi** che **NON** consumino inutilmente l'unità di elaborazione, cioè **evitare il busy waiting**.

Sleep e wakeup

Tutte le soluzioni viste fino ad ora fanno *SPIN LOCK*. Si ha il problema **dell'inversione di priorità.**

Esempio:

- Se eseguo un processo con **priorità 3**, e nel frattempo arriva un processo con **priorità 5**, allora esso deve essere eseguito prima **stoppando** quello con priorità 3.
- Quello con priorità **più alta** prova a entrare nella sezione crtica ma **rimane bloccato** perchè comunque c'è il processo prima che ancora non è uscito.
- Quindi il processo con priorità più alta va in **BUSY WAITING** dipendendo da processi con priorità più bassa (**inversione di priorità**)

> Ci sono processi con priorità più alta *che vengono eseguiti per prima*
> In un sistema interattivo, la priorità ce l'ha chi deve dare la risposta all'utente
> I processi vengono eseguiti **in base alla priorità**.

Soluzione al busy waiting

Dare la possibilità al processo di bloccarsi in **modo passivo** senza rimanere nel `WHILE` controllando ogni volta che la variabile cambi, quindi si ha la **RIMOZIONE DAI PROCESSI *READY*** e viene messo in una **coda a parte**.

- **NON** si ha più busy waiting
- *riduco la probabilità* che si verifichi l'inversione delle priorità
- si deve poter "*addormentare il processo*"

> Quando la parte condivisa si **libera**, si sveglia il processo mediante un segnale apposito e si rimanda in *READY*

- si usano le due chiamate di sistema **primitive** `sleep()` e `wakeup()`:
  - **sleep** viene chiamata se il processo **NON** può usare la risorsa condivisa. Allora viene addormentato e viene aggiunto a una **CODA DI DORMIENTI** (*rimosso dai processi READY* che competono per la *CPU*)
  - **wakeup** è il segnale che viene usato per risvegliare i processi dormienti (*e si scrive come ultma istruzione della* `leave_region`)