Concetti basilari di concorrenza

Index

Processi multipli

Per i SO moderni, è essenziale supportare più processi in esecuzione in uno di questi tre modi:

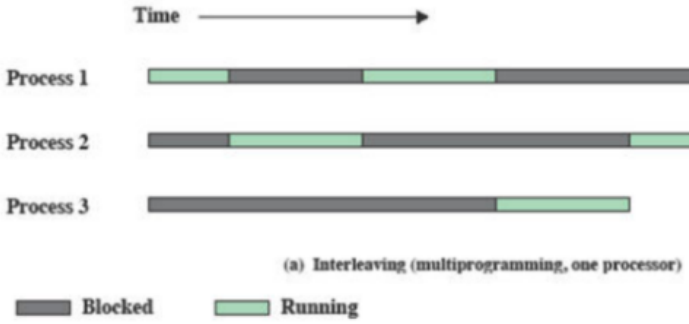
- multiprogrammazione
- multiprocessamento (*multiprocessing*)

- computazione distribuita (cluster)

Il grosso problema ora da affrontare è la **concorrenza**, ovvero gestire il modo con cui questi processi interagiscono

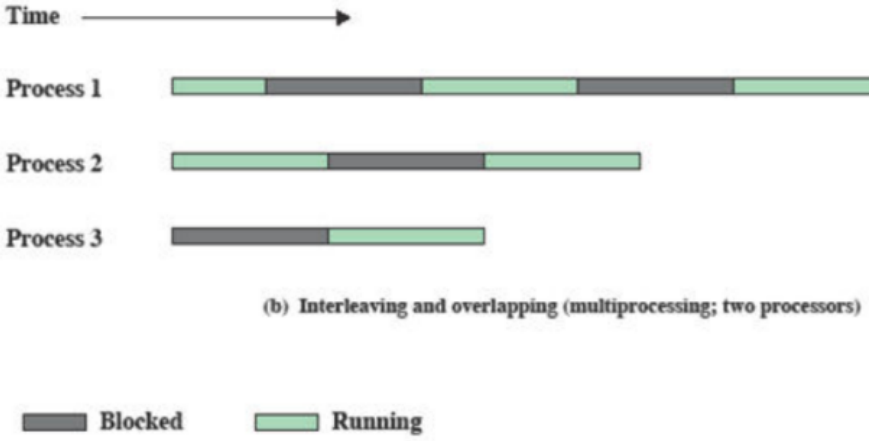
Multiprogrammazione

Se c'è un solo processore, i processi si alternano nel suo uso (*interleaving*)



Multiprocessing

Se c'è più di un processore, i processi si alternano (*interleaving*) nell'uso di un processore, e possono sovrapporsi nell'uso dei vari processori (*overlapping*)



Concorrenza

La concorrenza si manifesta nelle seguenti occasioni:

- applicazioni multiple → c'è condivisione del tempo di calcolo (a carico del SO, le altre no)
- applicazioni strutturate per essere parallele → perché generano altri processi o perché sono organizzate in thread
- struttura del sistema operativo → gli stessi SO operativi sono costituiti da svariati processi o thread in esecuzione parallela

Difficoltà

La difficoltà principale quando si parla di concorrenza è il fatto che **non si può fare alcuna assunzione sul comportamento dei processi** né sul comportamento dello scheduler

Un altro problema lo si ha nella **gestione delle risorse** (es. stampante), ovvero quando più processi tentano di accedere ad una risorsa ma che può essere acceduta da un solo processo (ma anche più semplice come quando 2 thread che accedono alla stessa variabile globale)

Inoltre si ha il problema della **gestione dell'allocazione delle risorse condivise** (decidere se dare o no una risorsa condivisa ad un processo), infatti la concorrenza fa in modo che non esista una gestione ottima del carico in quanto si potrebbe incorrere in race condition (es. processo potrebbe richiedere un I/O e poi essere rimesso ready prima di usarlo: quell'I/O va considerato locked oppure no?)

Risulta infine **difficile tracciare gli errori di programmazione**, quando viene violata la mutua esclusione ad esempio potrebbe essere un caso molto particolare dovuto alle scelte dello scheduler (tentando di ricreare la situazione che ha generato inizialmente l'errore potrebbe capitare che l'errore non avvenga)

Terminologia

Per poter affrontare il problema della concorrenza al meglio è necessario imparare della terminologia:

- **Operazione atomica** → sequenza indivisibile di programmi; il disparcher non può interrompere queste operazioni fino alla loro terminazione (nessun altro processo può vedere uno stato intermedio della sequenza o interrompere la sequenza)
- **Sezione critica** → una parte del codice di un processo in cui viene fatto un accesso ad una risorsa condivisa
- **Mutua esclusione** → questo è il problema principe quando si parla di concorrenza (gli altri sono più o meno collegati); avviene quando due processi provano ad accedere ad una risorsa condivisa, ma che è fatta in modo che solo un processo alla volta la può usare
- **Corsa critica** (*race condition*) → caso in cui la mutua esclusione viene violata (a causa di errori di programmazione)
- **Stallo** (*deadlock*) → situazione nella quale due o più processi non possono procedere con la prossima istruzione (tutti i processi della catena aspettano un

processo all'interno della catena)

- **Stallo attivo** (*livelock*) → situazione nella quale due o più processi cambiano continuamente il proprio stato, l'uno in risposta all'altro, senza fare alcunché di "utile"
- **Morte per fame** (*starvation*) → un processo, pur essendo ready, non viene mai scelto dallo scheduler

---

Esempio facile

Immaginiamo di avere questa procedura

```
/* chin e chout sono globali */
void echo() {
        chin = getchar(); // prende char in input
        chout = chin;
        putchar(chout); // stampa a schermo il char
}
```

Esempio su un processore

Supponiamo che ci siano due processi che tentano di eseguire la stessa procedura su un processore

```
Process P1                  Process P2
    .                           .
chin = getchar();               .
        .                       chin = getchar();
chout = chin;                   .
        .                       chout = chin;
putchar(chout);                 .
        .                       putchar(chout);
        .                           .
```

In questo caso avremmo in output lo stesso carattere nonostante ai due processi siano stati dati due input diversi (in P1 viene scritto il valore dato in input a P2)

Esempio su più processori

Non necessariamente l'avere più processori risolverebbe in automatico il problema

```
Process P1                    Process P2
    .                             .
chin = getchar();                 .
        .                     chin = getchar();
chout = chin;                 chout = chin;
putchar(chout);                   .
        .                     putchar(chout);
        .                         .
```

In questo caso infatti avremmo nuovamente lo stesso problema

Restrizione all'accesso singolo

Risolvere il problema dell'esempio precedente risulta essere particolarmente semplice. La soluzione infatti sta nel permettere l'esecuzione della funzione `echo` ad un solo processo alla volta (può essere richiesta da tutti i processi ma solo uno alla volta la può eseguire).
Questo viene chiamato rendere **atomica** una funzione

Race condition

Si ha una *corsa critica* quando più processi o thread leggono e scrivono la stessa risorsa condivisa e lo fanno in modo tale che lo stato della risorsa dipende dall'ordine di esecuzione dei processi e thread
In particolare, il risultato più dipendere dal processo o thread che finisce per ultimo.
La **sezione critica** è la parte di codice di un processo che può portare ad una corsa critica

---

Per ciò che riguarda il SO

Quali problemi di progetti e gestione sorgono dalla presenza della concorrenza?
Il SO deve:

- tener traccia di vari processi
- allocare e deallocare risorse (processore, memoria, file, dispositivi di I/O)
- proteggere dati e risorse dall'interferenza (non autorizzata) di altri processi

- assicurare che processi ed output siano indipendenti dalla velocità di computazione (ovvero dallo scheduling, non importa in che ordine li eseguo)

Interazione tra processi

Comunicazione	Relazione	Influenza	Problemi di controllo
Nessuna (ogni processo pensa di essere solo)	Competizione	Risultato di un processo indipendente dagli altri. Tempo di esecuzione di un processo dipende dagli altri	Mutua esclusione; deadlock; starvation
Memoria condivisa (i processi sanno che c'è qualche altro processo)	Cooperazione	Risultato di un processo dipendente dall'informazione data da altri. Tempo di esecuzione di un processo dipendente dagli altri	Mutua esclusione; deadlock; starvation; coerenza dei dati
Primitive di comunicazione (i processi sanno anche i PID di alcuni altri processi)	Cooperazione	Risultato di un processo dipendente dall'informazione data da altri. Tempo di esecuzione di un processo dipendente dagli altri	Deadlock; starvation

---

Processi in competizine

I problemi dei processi relazionati dalla competizione (ogni processo pensa solo a sé, ma per l'accesso alle risorse deve fare una richiesta al sistema tramite syscall) sono:

- Necessità di mutua esclusione (sezioni critiche)
- Deadlock
- Starvation

Mutua esclusione

Per risolverlo basterebbe fare in modo che i processi, per accedere ad una risorse, chiamino una syscall che entra nella sezione critica, fa l'operazione, esce.
Tuttavia ciò non è sempre possibile in quanto potrebbe essere necessario fare una richiesta esplicita di "bloccaggio" sulla risorsa, e in caso ciò avvenisse si ricade nel caso dei processi cooperanti

```
/* PROCESS 1 */
void P1
{
  while (true) {
    /* preceding code /;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 */
void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

. . .

```
/* PROCESS n */
void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

Mutua esclusione per processi cooperanti

In questo caso sono gli stessi processi che devono essere scritti pensando già alla cooperazione; infatti, usando opportune syscall, devono preoccuparsi di scrivere `entercritical` ed `exitcritical` in quando le specifiche dei processi potrebbero richiedere comportamenti particolari


```
/* PROCESS 1 */
void P1
{
  while (true) {
    /* preceding code /;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 */
void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

. . .

```
/* PROCESS n */
void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

Deadlock

Supponiamo che ci siano due processi: A che richiede accesso prima alla stampante e poi al monitor e B il contrario. Capita quindi che lo scheduler faccia andare B in mezzo alle due richieste di A (ha richiesto la stampante ma non il monitor) giusto il tempo necessario per richiedere il monitor.

Quindi adesso si ha nuovamente in esecuzione A che però diventa blocked perché sta aspettando il monitor (concesso a B), l'esecuzione passa allora a B che però diventa anche lui blocked poiché sta aspettando la stampante (concesso ad A)

Starvation

Supponiamo che ci siano due processi che richiedono la stampante, la stampante viene concessa al processo A mentre B sta facendo altro, quindi A rilascia la stampante ma per qualche motivo lo scheduler decide di lasciare A in esecuzione.
Quindi A fa in tempo a effettuare una seconda richiesta della stampante che gli viene nuovamente concessa. Ciò avviene indefinitamente mandando B in starvation

---

Requisiti per la mutua esclusione

Qualsiasi meccanismo si usi per offrire la mutua esclusione, deve soddisfare i seguenti requisiti:

- solo un processo alla volta può essere nella sezione critica per una risorsa
- niente deadlock né starvation
- nessuna assunzione su scheduling dei processi, né sul numero dei processi
- un processo deve entrare subito nella sezione critica, se nessun altro processo usa la risorsa
- un processo che si trova nella sua sezione non-critica non deve subire interferenze da altri processi (in particolare non può essere bloccato)
- un processo che si trova nella sezione critica ne deve prima o poi uscire (più in generale ci vuole cooperazione, es. non bisogna scrivere un processo che entra nella sua sezione critica senza chiamare `entercritical`)

Mutua esclusione for dummies

```
int bolt = 0;
void P(int i) {
        while (true) {
                bolt = 1;
                while (bolt == 1) /* do nothing */;
                /* critical section */;
                bolt = 0;
                /* remainder */
```

```
        }
}

// n processi iniziano in contemporanea P()
parbegin(P(0), P(1), ..., P(n))
```

In questo caso basta che lo scheduler faccia eseguire i 2 processi in interleaving ed è deadlock (rimangono bloccati nel while) oppure basta che ci sia un processo solo (anche lui bloccato, **attesa attiva**, nel while)

Provo quindi a scambiare dove assegno `bolt`

```
int bolt = 0;
void P(int i) {
        while (true) {
                while (bolt == 1) /* do nothing */;
                bolt = 1;
                /* critical section */;
                bolt = 0;
                /* remainder */
        }
}
```

In questo caso quando entra un solo processo questo non rimane incastrato nel while e viene rispettata la mutua esclusione nel caso in cui andasse in esecuzione un secondo processo che rimarrebbe bloccato nel while finché il primo processo non è uscito dalla sezione critica impostando `bolt` a 0 e lasciando continuare il secondo processo (`bolt` è una variabile globale)
Però anche in questo caso se un secondo processo viene mandato in esecuzione dopo la fine del while ma prima di `bolt=1` entrambi i processi potranno attraversare la critical section in contemporanea

---

Scheduler e livello macchina

Un'altra corsa critica meno evidente è il fatto che lo scheduler interrompe a livello di istruzione macchina
Supponiamo infatti che `P(0)` venga eseguito fino a `bolt = 1` compreso, si potrebbe pensare che almeno così la mutua esclusione sia rispettata e invece no. Infatti `P(1)` potrebbe essere arrivato in precedenza fino a "metà" del `while (bolt==1)` (ovvero

fino a caricare il valore della variabile `bolt`, che a quel punto era ancora 0, dentro un registro); quando il controllo ritorna a `P(1)`, per lui `bolt` vale 0