Concorrenza: mutua esclusione e sincronizzazione

Il sistema operativo si occupa della gestione dei processi e dei thread: multiprogramming (gestione di più processi in sistemi solo-core) , multiprocessing (gestione di più processi in sistemi multi-core) , distributed processing (gestione di più processi in sistemi distribuiti es. cluster).

La concorrenza interessa tre differenti contesti: applicazioni multiple (inventate affinchè possano condividere il tempo di processamento), applicazioni strutturate (estensione della programmazione strutturata), struttura dell'OS (a sua volta insieme di processi e thread).

DEFINIZIONI:

Operazione atomica: una funzione o un'azione implementata come una sequenza di una o più istruzioni che appaiono indivisibili; cioè nessun altro processo può vedere uno stato intermedio o interrompere l'operazione. E' garantita un'esecuzione completa o nel caso evitata un'esecuzione parziale. L'atomicità garantisce isolamento dai processi concorrenti.

Critical section: Una sezione di codice in cui un processo ha bisogno di un permesso d'accesso affinchè possa condividere informazioni. Non possono essere eseguiti più processi contemporaneamente su quel blocco di istruzioni.

Deadlock: Una situazione in cui due o più processi sono impossibilitati a procedere poiché tutti aspettano che sia fatto qualcosa da altri processi in deadlock.

Livelock: una situazione in cui due o più processi cambiano continuamente il loro stato in base ai cambiamenti degli altri processi coinvolti senza però effettuare lavoro utile.

Mutua esclusione: garantisce l'accesso ad un singolo processo della sezione critica.

Race condition: una situazione in cui più processi o thread leggono o scrivono un dato condiviso e il risultato finale dipende dal tempo di accesso a tale risorsa.

Starvation: Situazione in cui un processo in stato RUNNABLE è bloccato per un tempo indefinito dallo scheduler; nonostante è pronto per procedere non viene mai scelto.

FINE DEFINIZIONI

Il multiprogramming consiste nel produrre un risultato finale (output) indipendente dalla velocità di esecuzione degli altri processi concorrenti.

I principi della concorrenza sono la convivenza e la sovrapposizione. Purtroppo la velocità d'esecuzione di un processo non può essere prevista a priori poiché dipende da più fattori come l'attività degli altri processi, la modalità di gestione degli INTERRUPT da parte dell'OS o la politica di scheduling dell'OS.

Le difficoltà consistono nel condividere risorse globali, un allocazione ottimale della memoria da parte dell'OS o la difficoltà di individuare errori.

Una race condition di verifica nel momento in cui più processi o thread scrivono o leggono dati e il risultato finale dipende dall'ordine di esecuzione. Il perdente sarà colui che per ultimo aggiorna il valore e da cui dipenderà il risultato finale.

L'OS ha un ruolo fondamentale in questo ruolo poiché deve poter tener traccia di ogni processo, poterne allocare e deallocare le risorse in maniera efficiente, proteggere dati dall'interferenza di processi ed assicurarsi che i risultati finali non dipendano dalla velocità di computazione dei processi.

Tra due processi che non si vedono tra loro la relazione è indicata come COMPETIZIONE, i risultati di un processo non dipendono da risultati di altri processi però può cambiare il tempo di esecuzione di un processo. I possibili problemi di controllo sono la mutua esclusione, lo stallo o lo starvation.

Tra processi che vedono altri processi indirettamente la relazione si chiama COOPERAZIONE TRAMITE CONDIVISIONE, i risultati di un processo possono dipendere da risultati di altri processi e anche qui il tempo di esecuzione può cambiare, da aggiungere ai problemi di prima è il controllo della coerenza dei dati.

Tra processi che vedono altri processi direttamente la relazione si chiama COOPERAZIONE TRAMITE COMUNICAZIONE, i risultati di un processo possono dipendere da risultati di altri processi e anche qui il tempo di esecuzione può cambiare, i problemi di controllo sono lo stallo(deadlock) o lo starvation.

In una RESOURCE COMPETITION più processi entrano in conflitto per l'uso di una stessa risorsa e vanno quindi affrontati questi problemi: mutua esclusione, deadlock, starvation.

Requisiti per la mutua esclusione: deve essere forzata, un processo che si trova in una non critical section deve poter svolgere le proprie attività senza interferire con altri processi. Non si devono verificare deadlock o starvation. Non deve essere negato l'accesso a una cs se nessun altro è in quella cs. Non si devono effettuare assunzioni sulla velocità dei processi o il loro numero. Bisogna che un processo rimanga nella cs per un tempo finito.

HARDWARE SUPPORT FOR MUTUAL EXCLUSION

Possiamo disabilitare gli interrupt su sistemi single core in modo tale da poter garantire la mutua esclusione. Purtroppo però non si può effettuare lo stesso ragionamento su architetture multi-core e inoltre l'efficienza di esecuzione potrebbe risentirne in negativo.

Oppure possiamo utilizzare le COMPARE&SWAP INSTRUCTION, ossia istruzioni che effettuano una compare tra un valore in memoria e un valore di test e se i valori sono gli stessi viene effettuato uno swap del valore in memoria.

```
int compare_and_swap(int* reg, int oldval, int newval){
        ATOMIC();
        int old_reg_val = *reg;
```

```
        if(old_reg_val == oldval)
                *reg = newval;
        END_ATOMIC();
        Return old_reg_val;
}
```

Es. uso in cs:

```
While(compare_and_swap(&reg, 0, 1) == 1) //do nothing
//se supero entro in cs
reg = 0 //permetto ad altri di entrare
```

Oppure usando una funzione che scambia i valore di due registri

```
While(true){
        Int keyi = 1;
        do exchange(&keyi, &reg)
        while(keyi != 0) //prova a scambiare, ma nel registro avremo 1 finchè altri processi sono in
                                in cs, quando uno esce ci mette 0, io lo acchiappo e esco da while
        reg = 0          //permetto ad altri di entrare in cs
```

Vantaggi: Applicabile a un numero qualsiasi di processi sia su processori single-core sia su multi-core. Semplice e facile da verificare. Ogni cs può essere gestita da una variabile globale diversa. Svantaggi: c'è un approccio busy-waiting, finchè un processo occupa la cs gli altri verificano la condizione nel while. E' possibile la starvation nel caso ci siano molti processi in attesa della flag di entrata. Deadlock è possibile.

DEFINIZIONI DI MECCANISMI PER LA CONCORRENZA

Semaforo: Un valore intero usato come flag per i processi. Su di esso si possono effettuare solo tre operazioni, tutte e tre atomiche: inizializzazione, decremento, incremento. Il decremento potrebbe bloccare un processo, l'incremento sbloccarlo.

Semaforo binario: Un semaforo che prende come valori solo 0 o 1.

Mutex: Simile a un semaforo binario ma con la differenza che il processo che blocca il semaforo (lo setta a 0) è lo stesso che lo sblocca (setta a 1).

Condition Variable: Un particolare dato che blocca un processo o un thread finchè una data condizione non si verifica.

Monitor: Un costrutto del linguaggio di programmazione che contiene variabili, procedure di accesso e codice di inizializzazione. Un solo processo alla volta può accedere al monitor e alle variabili all'interno di esso si può accedere solo tramite le procedure dettate dallo stesso monitor. Quest'ultime sono considerate critical section. Il monitor potrebbe avere una coda d'attesa per l'accesso a dette variabili.

Event flags: Una parola di memoria usata come meccanismo di sincronizzazione. Applicazioni ad esempio possono associare a differenti eventi diversi bit di flags. Un thread può rimanere in attesa che si verifichi uno o più eventi osservando detti bit di flags. Il bloccaggio può avvenire in AND (ossia tutte le condizioni devono verificarsi) on in OR (ossia almeno una).

Mailboxes/Messages: Un modo per due o più processi per scambiarsi messaggi, anche usati per sincronizzarsi.

Spinlocks: Un meccanismo di mutua esclusione in cui un processo esegue infiniti loop di attesa aspettando che una variabile di accesso torni di nuovo disponibile.

FINE DEFINIZIONI

SEMAFORI
E' una variabile che un valore intero su cui solo tre operazioni sono definite, tutte e tre atomiche:
-Inizializzazione a un valore intero non negativo.
-semWait operation che decrementa il valore
-semSignal che incrementa il valore

Non c'è modo di sapere prima se una volta decrementato il semaforo quel processo verrà bloccato. Ne quale processo continuerà (su un processore single-core) l'esecuzione se ce ne sono due attivi contemporaneamente. Ne puoi sapere se un processo sta aspettando, quindi il numero di processi unblocked potrebbe essere 0 o 1.

```
struct semaphore{
	int count;
	queueType queue;
}

void semWait(semaphore s){
	s.count--;
	if(s.count < 0){
		/il processo va bloccato e messo in coda/
	}
}

Void semSignal(semaphore s){
	s.count++;
	if(s.count <= 0){
		/un processo rimosso dalla coda e messo nella runnuble list/
	}
}
```

O per il semaforo binario:

```
struct binary_semaphore{
	enum {zero, one} value;
	queueType queue;
```

```
}

void semWaitB(binary_semaphore s){
        if(s.value == one)
                s.value = zero;
        else{
                /il processo va bloccato e messo in coda/
        }
}

Void semSignalB(binary_semaphore s){
        If(s.queue is empty())
                s.value = one;
        else{
                /un processo rimosso dalla coda e messo nella runnuble list/
        }
}
```

La coda è utilizzata per mantenere un processo in attesa del semaforo.
Abbiamo due tipi di semaforo: Strong Semaphore, che basa la sua implementazione su una FIFO, oppure un Weak Semaphore, la cui rimozione dalla coda non è specificata.

Es. mutua esclusione con i semafori:

```
while(true){
        semWait(s); //s un semaforo
        /critical section/
        semSignal(s)
}
```

PROBLEMA PRODUTTORE/CONSUMATORE

Uno o più processi generano dati e li conservano in un buffer. Un singolo consumatore può prelevare e utilizzare questi dati dal buffer. Un sono processo (PROD/CONS) può accedere al buffer condiviso. (Ipotesi generale, NB possiamo avere anche più consumatori).
Bisogna assicurarsi inoltre che produttori non inseriscano dati in un buffer pieno o che consumatori prelevino dati da buffer vuoti.
Analizziamo ora il caso di buffer infinito evitando il problema di buffer pieno:
Possiamo utilizzare o un binary semaphore in questo modo

```
binary_semaphore s = 1, delay = 0;

void producer(){
        while(true){
                produce();
                semWaitB(s);
                append();
                n++;
```

```
            if(n == 1) semSignalB(delay);
            semSignalB(s);
        }
}

void consumer{
        int m; //uso m  perché così salvo il valore da verificare per la wait su delay da modifiche da parte di producer…. Se producer schedula prima dell'if del consumer vado a consumare un elemento che non esiste.
        semWaitB(delay);
        while(true){
            semWaitB(s);
            take();
            n--;
            m = n;
            semSignalB(s);
            consume();
            if(m==0) semWaitB(delay);
        }
}
```

Oppure general solution:

```
semaphore n = 0, s = 1;

void producer(){
        while(true){
            produce();
            semWait(s);
            append();
            semSignal(s);
            semSignal(n);
        }
}

Void consumer(){
        While(true){
            semWait(n);
            semWait(s);
            take();
            semSignal(s);
            consume();
        }
}
```

O nel caso di bounded buffer bisogna stare attenti anche a buffer pieno.

```
semaphore n = 0, s = 1,e = sizeofbuffer;
```

```
void producer(){
        while(true){
                produce();
                semWait(e);
                semWait(s);
                append();
                semSignal(s);
                semSignal(n);
        }
}

Void consumer(){
        While(true){
                semWait(n);
                semWait(s);
                take();
                semSignal(s);
                semSignal(e);
                consume();
        }
}
```

I semafori possono essere implementati o attraverso software come l'algoritmo di Dekker o tramite un supporto hardware come visto in precedenza. La semWait e la semSignal DEVONO essere atomiche.
Es. basta inserire un while con un istruzione compare_and_swap tra una flag del semaforo e un valore di test, poi riportare la flag a 0 nel momento in cui finisco le operazioni sul semaforo.
Oppure inibire gli interrupts prima di effettuare le operazioni sui semafori, nel caso della wait se la decrementazione non porta al blocco del processo ristabilire gli interrupt, nel caso della signal sempre ristabilirli.

MONITORS

Sono costruzioni del linguaggio di programmazione che permettono di avere le stesse funzionalità dei semafori ma con un controllo più semplice. Su di esso sono definite procedure, sequenze di inizializzazione e variabili locali.
Le variabili locali sono accessibili solo tramite le procedure del monitor e non da primitive esterne.
Nel momento in cui un processo chiama una procedura del monitor entra in quest'ultimo.
Solo un processo può essere eseguito nel monitor alla volta.
Sulle variabili di condizione si può operare con due funzioni: cwait(c), che sospende il processo chiamante sulla condizione c, csignal(c), che riattiva l'esecuzione di qualche processo bloccato dopo una cwait(c) quando la condizione si verifica.
Es. bounded buffer prod/cons with monitor
NB nelle funzioni producer e consumer non avremo nessuna condizione di verifica poichè nel momento che un processo chiama append o take (procedure del monitor) automaticamente vengono gestiti i casi di buffer pieno e vuoto, inoltre la mutua esclusione è assicurata dall'implementazione stessa del monitor.

```
Monitor boundedbuffer;
char buffer[N]
int nextin, nextout;
int count;
cond notfull,notempty;

void append(char x){
        if(count == N) cwait(notfull);
        buffer[nextin] = x;
        nexttin = (nextin + 1) % N;
        count++;
        csignal(notempty);
}
void take(char x){
        if(count == 0) cwait(notempty);
        x = buffer[nextout];
        nextout = (nextout + 1) % N;
        count--;
        csignal(notfull);
}
```

MESSAGE PASSING

Il message passing è un approccio che garantisce sia sincronizzazione che comunicazione tra processi. Può essere impiegato sia su sistemi distribuiti o sistemi con architettura uniprocessor o multiprocessor.
Il tutto si basa su due primitive fondamentali: la send(destination, message) e la receive(source, message).
La sincronizzazione che si ottiene tramite questo approccio soddisfa moltissime specifica: possiamo avere delle primitive (send, receive) bloccanti o non bloccanti.
Il caso più sicuro è che entrambe siano bloccanti ossia nel momento in cui un processo A invia un messaggio al processo B, il processo A si blocca finche B non lo riceve e B si blocca finche A non invia qualcosa. Una stretta sincronizzazione che viene indicata con il nome di rendezvous.
Possiamo però optare per una non blocking send ossia viene bloccato solo il processo B ricevente, mentre il processo A una volta terminato l'invio del messaggio può continuare con il suo task.
Molto più utile questa modalità che permette, ad esempio, di soddisfare più invii nella maniera più veloce possibile. Manca però la conferma di consegna che potrebbe però portare a dati parzialmente corrotti o ad errori di invio. Infine possiamo optare per un opzione non bloccante da parte di entrambe le primitive, nel caso la receive controllasse un buffer vuoto abbandonerebbe la richiesta.
Possiamo avere due tipi di addressing: diretto o indiretto.
Quello diretto identifica ogni processo con un id, da specificare ogni qual volta bisogna inviare un messaggio. Per la ricezione invece abbiamo due approcci: l'esplicito e l'implicito.
Il primo prevede la designazione di un processo sender in modo tale da poterne specificare l'id.
Il secondo invece nel parametro source della primitiva receive possiede un valore di ritorno quando è avvenuta la ricezione.
Quello indiretto invece prevede una struttura dati condivisa in cui i messaggi vengono inviati e accumulati in coda per poter essere poi prelevati dai ricevitori. La coda è definita mailboxe.

Questo approccio offre molta flessibilità poiché possiamo condividere la mailboxe tra N sender e M receiver. Ogni messaggio però conterrà una parte consistente di Header in cui verranno specificati molti parametri tra cui il tipo di messaggio, il destinatario effettivo, la lunghezza del messaggio, il codice di controllo e il messaggio vero e proprio.
Per garantire mutua esclusione in questo caso possiamo far condividere una mailboxe tra i processi in cui tutti sono receiver e tutti sono sender. Appena si ottiene un messaggio (token pass) si può entrare in critical section (receive bloccante) e poi si invia un messaggio contenente il pass per gli altri processi (send non bloccante). NB il primo messaggio lo invia il main per far partire il tutto.