3-Concorrenza

La concorrenza riguarda la gestione di processi multipli, come:

- Multiprogramming: più processi su un solo processore
- Multiprocessing: più processi su una macchina con processori multipli
- Distributed Processing: più processi su un insieme di computer.
  Due programmi si dicono in esecuzione concorrente se vengono eseguiti in parallelo.
  Con concorrenza si intende l'insieme di notazioni per descrivere l'esecuzione concorrente di due o più programmi. è l'insieme di tecniche per risolvere problemi associati all'esecuzione concorrente (comunicazione e sincronizzazione).
  In un sistema multiprocessore/distribuito si hanno dei processori in esecuzione in contemporanea, ma il problema di base è lo stesso: più processi eseguiti simultaneamente su processori diversi (overlapping). I problemi sono gli stessi perché non è possibile predire la velocità relativa dei processi. Non vi è sostanziale differenza tra i problemi relativi a multiprogramming e multiprocessing.
  Esempio con 2 processi che vogliono accedere alla stessa variabile globale (`totale`)

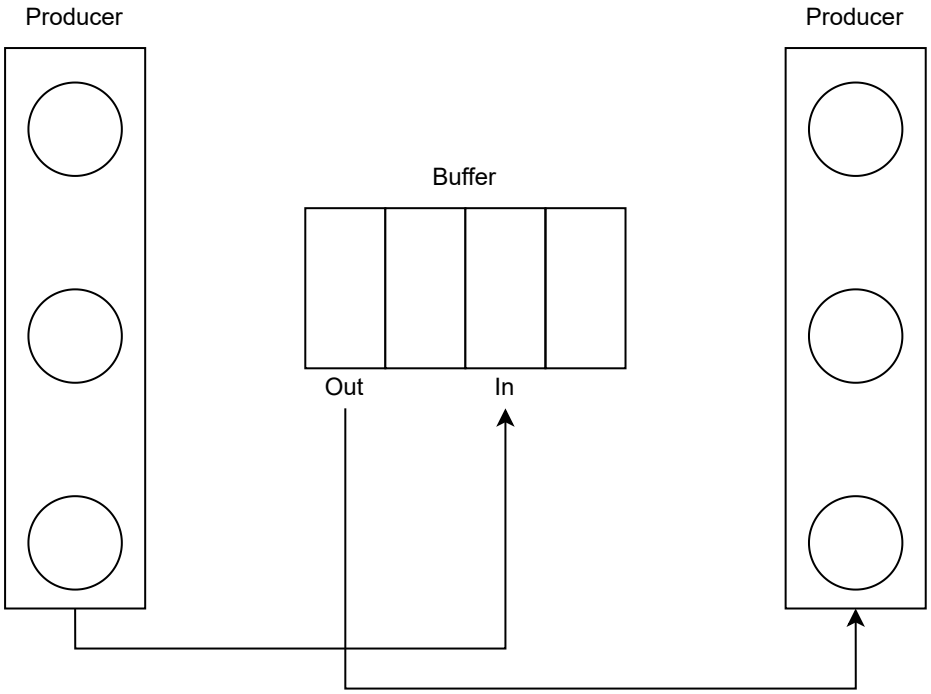
```
void modifica(int valore) {
    totale = totale + valore;
}

void main(){
    global int totale = 100;
    modifica(10); //processo 1
    modifica(-10); // processo 2
    print(totale);
}
```

I due processi hanno insiemi di registri distinti, ma l'accesso alla memoria su `totale` non può essere simultaneo, per cui non sempre `totale` avrà valore 100 alla fine dei 2 processi.

Problema Producer-Consumer

Struttura per processi che cooperano: il processo produttore crea informazione richiesta dal processo consumatore. Se ci sono più produttori e consumatori non va bene avere una relazione unica tra produttore e consumatore, anche perché la frequenza di produzione e consumo non è la stessa. Ad ogni istante la velocità di produzione/consumo deve adeguarsi. Bisogna quindi dissociare questo processo diretto usando un buffer, dove le informazioni dei produttori vengono poste nella prima cella libera del buffer, e i consumer prendono l'informazione nella casella meno recente del buffer.



Se il buffer si riempie il produttore deve aspettare che si liberi una cella. Se invece è vuoto il consumatore non deve accedere alle celle vuote.

Problemi di concorrenza

Se il buffer è una variabile globale, context switch possono assegnare informazioni in caselle già occupate perdendo cosi informazioni. Due processi possono modificare una variabile condivisa contemporaneamente, ed è sufficiente per causare errori nell'esecuzione di processi concorrenti. Bisogna quindi limitare le concorrenza quando il codice sta modificando variabili globali o condivise. Questi punti sono detti "regioni critiche" (CS) e l'esecuzione di 2 o più di queste non deve mai avvenire contemporaneamente. Questo problema può essere evitato con:

- Esclusione reciproca: il processo viene arrestato quando entra nella sua regione critica: se un processo sta eseguendo la sua CS, allora nessun altro processo può eseguire la propria CS.
- Progresso: se nessun processo sta eseguendo la sua CS e ci sono processi che vogliono entrare nella loro CS, la scelta dei processi con la prossima CS da eseguire non possono essere continuamente posticipati (starvation).
- Attesa Limitata: dopo che un processo ha richiesto l'accesso alla propria CS, deve esserci un limite al numero di volte che gli altri processi possono entrare nelle loro CS prima che la richiesta del primo processo sia soddisfatta.

Implementazione - Arbitro

L'arbitro è un algoritmo che si occupa di stabilire i turni di esecuzione delle CS dei vari processi.

- Algoritmo 1: un processo con CS in esecuzione può finire il suo quanto di tempo ed essere messo nello stato ready. Questo non deve autorizzare l'arbitro a far eseguire la sezione critica di un altro processo. Un ulteriore problema si incontra quando due processi hanno numero di richieste di accesso alle loro CS diversi e si rischia di uscire dal bounded waiting dato che ci si aspetta un alternanza esatta dell'uso della CS dei due processi: prende decisioni ignorando le richieste dei thread.

```
public static class Algorithm_1 extends MutualExclusion {
    public Algorithm_1() {
        turn = TURN_0; // turn = 0, si parte dal processo con id 0
    }
    public void enteringCriticalSection(int t) {
        while (turn != t)
            Thread.yield();
    }
    public void leavingCriticalSection(int t) {
        turn = 1 - t; // tocca all'altro quando esco dalla sezione critica
    }
    private volatile int turn;
}
```

- Algoritmo 2: se avviene un context switch dopo che la flag di un processo `t` diventa `true` (ovvero la sezione critica di `t` è in esecuzione e quindi l'altro va messo in pausa) si rischia di entrare in deadlock dato che il flag di entrambi i processi può essere `true`.

```
public static class Algorithm_2 extends MutualExclusion {
    public Algorithm_2() {
        flag[0] = false;
        flag[1] = false;
    }
    public void enteringCriticalSection(int t) {
        int other = 1 - t;
        flag[t] = true; // il thread t è in sezione critica
        while (flag[other])
            // se il flag di other è true rimango bloccato finché l'altro
            // thread non invoca il leaving critical section
            Thread.yield();
    }
    public void leavingCriticalSection(int t) {
        flag[t] = false;
    }
    private volatile boolean[] flag = new boolean[2];
}
```

- Algoritmo 3: non ha problemi, ed è una fusione dei processi dei due algoritmi precedenti (l'alternanza di 1 e il flag di 2).

```
public static class Algorithm_3 extends MutualExclusion {
    public Algorithm_3() {
        flag[0] = false;
        flag[1] = false;
        turn = TURN_0;
    }
    public void enteringCriticalSection(int t) {
        // la richiesta viene annotata sul flag e si setta turn a other
        int other = 1 - t;
```

```
        flag[t] = true;
        turn = other;
            // se l'altro processo è in esecuzione e tocca all'altro
            // processo allora aspetto.
            // per uno dei due thread (turn == other) sarà falsa e quindi si
            //potrà uscire dal while
        while ((flag[other]) && (turn == other))
            Thread.yield();
    }
    public void leavingCriticalSection(int t) {
        flag[t] = false;
    }
    private volatile int turn;
    private volatile boolean[] flag = new boolean[2];
}
```

Semafori

I semafori aiutano i programmatori a gestire le CS, indicato da un intero S e può essere modificato solamente tramite due operazioni indivisibili.

```
P(S):
    while (S <= 0) do no-op
    S--

V(S):
    S++
```

Vanno ignorate le interrupt all'interno delle operazioni di incremento e decremento e tra il `while` e il decremento. Se ci fosse un context switch tra queste due, verrebbe eseguita la CS di un altro processo. Si può sfruttare il SO per rendere operazioni semplici all'interno del SO stesso atomiche. Questo non si può fare per codice arbitrario ed è il motivo per cui bisogna scrivere programmi apposta che riducano le CS il più possibile e gestirle correttamente.
Questo è un esempio di semaforo binario, ma esistono anche semafori n-ari ($S \geq 2$).
Si dice spinlock un semaforo con attesa attiva tramite ciclo `while`, ed è usato in sezioni dove la CS è breve (il busy waiting non dura molto)

```
Semaphore S; // initialized to 1
P(S) // decremento, si chiede permesso al semaforo di entrare in sez. critica
CriticalSection()
V(S)// incremento e notifica fine sezione critica
```

Un processo alla volta può accedere alla sezione condivisa (mutual exclusion). Bisogna acquisire una chiave per entrare (`P(S)`). Se un processo vuole eseguire la sua CS non può farlo se $S = 0$ e quindi verrà messo in attesa.

Semaforo senza busy waiting

Il busy waiting è utilizzato per istruzioni estremamente brevi e semplici, ma quando non è più affidabile si usano istruzioni di incremento e decremento più complesse.
Il semaforo S ha associato una lista dei processi bloccati su di esso.

```
P(S) {
    S-- // il decremento può essere prima o dopo l'operazione di blocco
    if (S < 0) {
        S.add(P) // add this process to list
        block(P) // blocca il processo
    }
}

V(S) {
    S++
    if (S <= 0) {
        S.remove(P) // rimuove il processo dalla lista
        wakeup(P)
    }
}
```

> $P(S)$: Decrementa S , se $S < 0$ blocca il processo
> $V(S)$: Incrementa S, se $S <= 0$ sveglia il processo

Metodo `remove` chiamato dal consumatore

```
public Object remove() {
    Object item;
    while (count == 0)
        ; // do nothing
// remove an item from the buffer
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

Vogliamo eliminare i costrutti che bloccano i processi `while` loop. La gestione delle sezioni condivise deve essere lasciata interamente ai semafori, che devono bloccare i consumatori quando il buffer è vuoto e i produttori quando è pieno. Se c'è un buffer vuoto, `count == 0` e `Semaforo == 1`, decrementando il produttore con `Semaforo==0` rischia di rimanere bloccato in attesa che torni ad 1. Questo può causare deadlock.