Module 6: Synchronization

Cos'è la concorrenza?

- **Tema centrale nella progettazione dei S.O. riguarda la gestione di processi *multipli***
  - *Multiprogramming*
    - più processi su un solo processore
    - parallelismo *apparente*
  - *Multiprocessing*
    - più processi su una macchina con processori multipli
    - parallelismo *reale*
  - *Distributed processing*
    - più processi su un insieme di computer distribuiti e indipendenti
    - parallelismo *reale*

Cos'è la concorrenza?

- **Esecuzione concorrente:**
  - due programmi si dicono in esecuzione concorrente se vengono eseguiti in parallelo (con parallelismo reale o apparente)
- **Concorrenza:**
  - è l'insieme di notazioni per descrivere l'esecuzione concorrente di due o più programmi
  - è l'insieme di tecniche per risolvere i problemi associati all'esecuzione concorrente, quali *comunicazione* e *sincronizzazione*

Dove possiamo trovare la concorrenza?

- **Applicazioni multiple**
  - la multiprogrammazione è stata inventata per condividere del processore da parte di più processi indipendenti
- **Applicazioni strutturate**
  - estensione del principio di progettazione modulare; alcune applicazioni possono essere progettate come un insieme di processi o thread concorrenti
- **Struttura del sistema operativo**
  - molte funzioni del sistema operativo possono essere implementate come un insieme di processi o thread

Multiprocessing e multiprogramming: differenze?

- **In un singolo processore:**
  - processi multipli sono "*alternati nel tempo*" per dare l'impressione di avere un multiprocessore
  - ad ogni istante, al massimo un processo è in esecuzione
  - si parla di *interleaving*
- **In un sistema multiprocessore:**
  - più processi vengono eseguiti *simultaneamente* su processori diversi
  - i processi sono "*alternati nello spazio*"
  - si parla di *overlapping*

Multiprocessing e multiprogramming: differenze?

- **A prima vista:**
  - si potrebbe pensare che queste differenze comportino problemi distinti
  - in un caso l'esecuzione è simultanea
  - nell'altro caso la simultaneità è solo simulata
- **In realtà:**
  - presentano gli stessi problemi
  - che si possono riassumere nel seguente:
    - non è possibile predire la velocità relativa dei processi

Un esempio semplice

- **Si consideri il codice seguente:**

In C:

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

In Assembly:

```
.text
modifica:
  lw $t0, totale
  add $t0, $t0, $a0
  sw $t0, totale
```

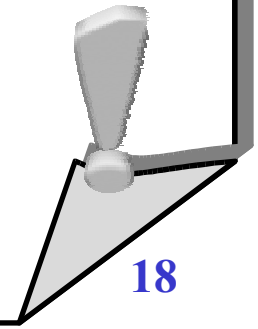
- **Supponiamo che:**
  - Esista un processo P1 che esegue **`modifica(+10)`**
  - Esista un processo P2 che esegue **`modifica(-10)`**
  - P1 e P2 siano in esecuzione concorrente
  - **`totale`** sia una variabile condivisa tra i due processi, con valore iniziale 100
- **Alla fine, `totale` dovrebbe essere uguale a 100. Giusto?**

Scenario 1: multiprogramming (corretto)

P1	`lw $t0, totale`	totale=100, $t0=100, $a0=10
P1	`add $t0, $t0, $a0`	totale=100, $t0=110, $a0=10
P1	`sw $t0, totale`	totale=110, $t0=110, $a0=10
S.O.	`interrupt`	
S.O.	`salvataggio registri P1`	
S.O.	`ripristino registri P2`	totale=110, $t0=? , $a0=-10
P2	`lw $t0, totale`	totale=110, $t0=110, $a0=-10
P2	`add $t0, $t0, $a0`	totale=110, $t0=100, $a0=-10
P2	`sw $t0, totale`	totale=100, $t0=100, $a0=-10

Scenario 2: multiprogramming (errato)

P1	`lw $t0, totale`	totale=100, $t0=100, $a0=10
S.O.	`interrupt`	
S.O.	`salvataggio registri P1`	
S.O.	`ripristino registri P2`	totale=100, $t0=? , $a0=-10
P2	`lw $t0, totale`	totale=100, $t0=100, $a0=-10
P2	`add $t0, $t0, $a0`	totale=100, $t0= 90, $a0=-10
P2	`sw $t0, totale`	totale= 90, $t0= 90, $a0=-10
S.O.	`interrupt`	
S.O.	`salvataggio registri P2`	
S.O.	`ripristino registri P1`	totale= 90, $t0=100, $a0=10
P1	`add $t0, $t0, $a0`	totale= 90, $t0=110, $a0=10
P1	`sw $t0, totale`	totale=110, $t0=110, $a0=10



Scenario 3: multiprocessing (errato)

- **Il due processi vengono eseguiti simultaneamente da due processori distinti**

Processo P1:

```
lw $t0, totale

add $t0, $t0, $a0
sw $t0, totale
```

Processo P2:

```
lw $t0, totale
add $t0, $t0, $a0

sw $t0, totale
```

- **Nota:**
  - i due processi hanno insiemi di registri distinti
  - l'accesso alla memoria su **`totale`** non può essere simultaneo

Alcune considerazioni

- **Non vi è sostanziale differenza tra i problemi relativi a multiprogramming e multiprocessing**
  - ai fini dei ragionamenti sui programmi concorrenti si ipotizza che sia presente un "processore ideale" per ogni processo
- **I problemi derivano dal fatto che:**
  - non è possibile predire gli istanti temporali in cui vengono eseguite le istruzioni
  - i due processi accedono ad una o più risorse condivise

Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

- Processes/thread that execute concurrently need to interact in order to communicate and synchronise
  - exchange of data
  - correct use of shared data structures
- several mechanisms proposed
- for simplicity we will refer mainly to processes
  - similar mechanisms exist for threads

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.

Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
         while (count == BUFFER_SIZE)
                    ; // do nothing
               buffer [in] = nextProduced;
               in = (in + 1) % BUFFER_SIZE;
               count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                 count--;

                     /*  consume the item in nextConsumed
 }
```

Bounded Buffer

```
public class BoundedBuffer {
   public void enter(Object item) {
            // producer calls this method
   }


   public Object remove() {
            // consumer calls this method
    }
    // potential race condition on count
    private volatile int count;
}
```

enter() Method

```
// producer calls this method
public void enter(Object item) {
        while (count == BUFFER_SIZE)
                ; // do nothing
        // add an item to the buffer
        ++count;
        buffer[in] = item;
         in = (in + 1) % BUFFER_SIZE;
}
```

remove() Method

```
// consumer calls this method
   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;
   }
```

An execution leading to an incorrect value of `count`

Statement `++count` can become, in assembly:

$$\begin{aligned} \text{register}_1 &= \texttt{count} \\ \text{register}_1 &= \text{register}_1 + 1 \\ \texttt{count} &= \text{register}_1 \end{aligned}$$

Similarly for `--count`:

$$\begin{aligned} \text{register}_2 &= \texttt{count} \\ \text{register}_2 &= \text{register}_2 - 1 \\ \texttt{count} &= \text{register}_2 \end{aligned}$$

The above instructions can be interleaved in a concurrent execution of `++count` and `--count`, for instance thus:

(initial value of `count` is 5)

$$\begin{aligned}
\text{register}_1 &= \texttt{count} \\
\text{register}_1 &= \text{register}_1 + 1 \\
\text{register}_2 &= \texttt{count} \\
\text{register}_2 &= \text{register}_2 - 1 \\
\texttt{count} &= \text{register}_1 \\
\texttt{count} &= \text{register}_2
\end{aligned}$$

(final – and incorrect – value of `count` is 4)

- The problem: both thread manipulate `count` concurrently
- Race condition
- we need ways to synchronise the access to `count` from the 2 threads

Critical region: part of a process that accesses shared data structures

- it represents a potential point of interference

Objective: make sure that the critical regions of two processes are never executed concurrently (mutual exclusion)

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Worker Thread

```
public class Worker extends Thread {
   public Worker(String n, int i, MutualExclusion s) {
      name = n;
      id = i;
      shared = s;
   }
   public void run() { /* see next slide */ }

   private String name;
   private int id;
   private MutualExclusion shared;
}
```

run() Method of Worker Thread

```
public void run() {
    while (true) {
        shared.enteringCriticalSection(id);
        // in critical section code
        shared.leavingCriticalSection(id);
        // out of critical section code
    }
}
```

MutualExclusion Abstract Class

```
public abstract class MutualExclusion {
    public static void criticalSection() {
            // simulate the critical section
    }

    public static void nonCriticalSection() {
            // simulate the non-critical section
    }
    public abstract void enteringCriticalSection(int t);
    public abstract void leavingCriticalSection(int t);
    public static final int TURN_0 = 0;
    public static final int TURN_1 = 1;
}
```

Testing Each Algorithm

```
public class TestAlgorithm
{
   public static void main(String args[]) {
            MutualExclusion alg = new Algorithm_1();

            Worker first = new Worker("Runner 0", 0, alg);
            Worker second = new Worker("Runner 1", 1, alg);

            first.start();
            second.start();
   }
}
```

Algorithm 1

```
public class Algorithm_1 extends MutualExclusion {
    public Algorithm_1() {
            turn = TURN_0;
    }
    public void enteringCriticalSection(int t) {
            while (turn != t)
                    Thread.yield();
    }
    public void leavingCriticalSection(int t) {
            turn = 1 - t;
    }
    private volatile int turn;
}
```

Algorithm 2

```
public class Algorithm_2 extends MutualExclusion  {
   public Algorithm_2() {
            flag[0] = false;
            flag[1] = false;
   }
   public void enteringCriticalSection(int t) {
            // see next slide
    }
    public void leavingCriticalSection(int t) {
             flag[t] = false;
   }

   private volatile boolean[] flag = new boolean[2];
}
```

Algorithm 2 – enteringCriticalSection()

```
public void enteringCriticalSection(int t) {
   int other = 1 - t;
   flag[t] = true;
          while (flag[other] == true)
                   Thread.yield();
  }
```

Algorithm 3

```
public class Algorithm_3 extends MutualExclusion {
   public Algorithm_3() {
            flag[0] = false;
            flag[1] = false;
            turn = TURN_0;
   }
   public void enteringCriticalSection(int t) {/* see next slides */ }
   public void leavingCriticalSection(int t) {{/* see next slides */ }
   private volatile int turn;
   private volatile boolean[] flag = new boolean[2];
}
```

Algorithm 3 – enteringCriticalSection()

```
public void enteringCriticalSection(int t) {
   int other = 1 - t;
   flag[t] = true;
   turn = other;

   while ( (flag[other] == true) && (turn == other) )
            Thread.yield();
  }
```

Algo. 3 – leavingingCriticalSection()

```
public void leavingCriticalSection(int t) {
        flag[t] = false;
}
```

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

P (*S*): **while** *S*≤ 0 **do** *no-op*;
S--;

V(*S*): *S++*;

Semaphore as General Synchronization Tool

Semaphore S; // initialized to 1

P(S);

CriticalSection()

V(S);

Semaphore Eliminating Busy-Waiting

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

Synchronization Using Semaphores

```
public class FirstSemaphore {
  public static void main(String args[]) {
    Semaphore sem = new Semaphore(1);
    Worker[] bees = new Worker[5];

    for (int i = 0; i < 5; i++)
      bees[i] = new Worker(sem, "Worker " + (new Integer(i)).toString() );

    for (int i = 0; i < 5; i++)
      bees[i].start();
  }
}
```

Worker Thread

```
public class Worker extends Thread {
    public Worker(Semaphore) { sem = s;}
    public void run() {
            while (true) {
                    sem.P();
                    // in critical section
                    sem.V();
                    // out of critical section
            }
    }
    private Semaphore sem;
}
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process