TDDC47

Real-time and Concurrent Programming

Lecture 2: Processes and shared resources

Simin Nadjm-Tehrani

Real-time Systems Laboratory

Department of Computer and Information Science
Linköping university

Exercise from last time

- Questions and comments from last lecture
  - Faster pace?
  - Language?

Message from LiU

This lecture

- Processes, threads, and their representation
- Concurrency and execution
- Communication
- Synchronisation

- The mutual exclusion problem

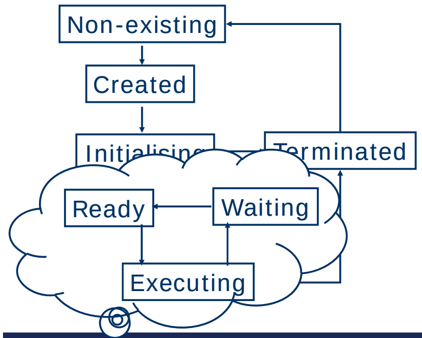
Recall from last lecture

- A concurrent program consists of a set of autonomous computation processes (logically) running in parallel
- A *process* has its own thread of control and its own address space
- Operating system may arbitrarily switch among processes and give control of the CPU to some process

Each process

- Is managed by the OS and has
  - A process ID
  - A record of its run-time data kept in the OS "process control block" – PCB
- Is allocated segments in the memory:
  - Code: the machine instructions it will run
  - Data: global variables and memory allocated at run-time
  - Stack: local variables and records of functions activated

Process life cycle



Analogies

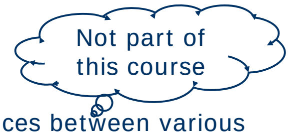
- Cooking recipe: program
- Preparing a dish: running different instances of concurrent processes
- Looking for ingredients: initialisation

Programming for processes

- Typical real-time systems: Assembler, C
  - Need support from OS to create, schedule and terminate tasks
- Languages with support for concurrent programming: Java, Ada
  - Have their own run-time system and can explicitly create processes

Program / process structure

- Static/Dynamic
- Nested/Flat



- Substantial differences between various languages wrt syntax, execution model, and termination semantics for nested processes

Threads (& tasks)

- Some modern operating systems allow a process to create (spawn) a number of parallel threads of control – in the *same* address space
- Each process has its own stack
- Processes that run with a shared address space (code, data) are called light-weight processes or *threads*
- In this course we consider processes with one thread of control and use *tasks* interchangeably with processes

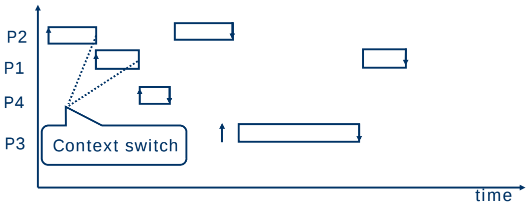
Logical parallelism

P1:	P2:
{while true do think; talk }	{while true do sleep; listen }

Which potential execution sequences (*traces*)?

Context Switch

- Consider a program that consists of processes P1,..., P4
- An execution of the concurrent program may look like:



Keeping track of execution

- Each process in PCB has:
  - Process state (which stage in life cycle)
  - Program counter (where in its running code)
  - Value of internal variables
  - The allocated memory areas
  - Open files and other resources

This lecture

- Processes, threads, and their representation
- Concurrency and execution
- Communication
- Synchronisation
- The mutual exclusion problem

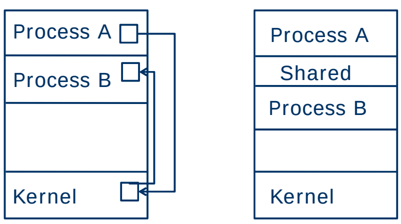
Process communication

Data sharing:
- Output for one process may be input for another process
  - Compute distance, and then compute incremental acceleration

Flow of control (synchronisation):
- One process may only start after another one finished
  - Update display only after all sensor values are received
- Sharing common resources
  - No more than one process at a time may send a packet on a shared channel

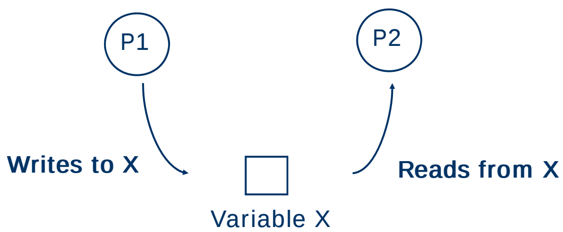
Communication

- Two modes:
  - Shared variables
  - Message passing



Basic operation

- Communication using shared variables



Problems?

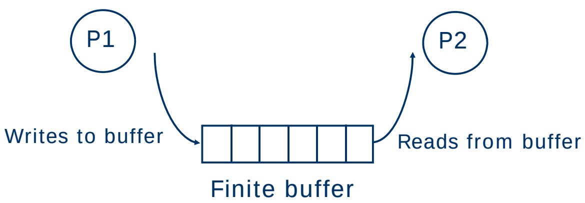
- No! Since hardware memory can support *atomic* memory update, so that only one writes at a time
- But…
- Creates problems for more complex shared data structures
  - Update date, time and stock value
- One process may write at a faster rate than the other process reads
  - Register a call to a telecom server before serving it
  - Is the call on the register the earliest unserved one? Even if server is overloaded?

Race condition

- Consider two processes P1 and P2 that are allowed to write to a shared data structure
- If the order of updating the data structure by respective processes can affect the outcome of the computation then the system suffers from a *race condition*
  - Analogy: Deposit 5000kr in the account, calculate accrued interest
- Process synchronisation is used to avoid race conditions

Decoupling from process rates

- Finite buffers



Issues

- Must check that buffer is not full when writing to it
- Must check that buffer is not empty when reading from it
- Must ensure that two processes do not write on one buffer position at the same time

General problems

- Conditional action
  - Examples:
    - Compute the interest when all transactions have been processed
    - Check that seats are available before booking
- Mutual exclusion
  - Example:
    - Two customers shall not be booked on the same seat

Analogies

Summary so far…

- **Synchronisation**
  - Mash the potatoes once they are cooked
- **Data sharing**
  - Two sets of guests
- **Mutual exclusion**
  - Two appliances cannot use the same electricity socket at the same time, e.g. toaster and electric kettle

This lecture

- Processes, threads, and their representation
- Concurrency and execution
- Communication
- Synchronisation

- The mutual exclusion problem

Example

[Garg 2005]

- Consider the two processes using a shared variable x initialised at 0:

```
P0              P1
{x= x+1         {x= x+1
}                }
```

- What is the outcome of running them both to completion?

It depends…

Machine instructions

```
LD R, x   // load register R from x
INC R     // increment register R
ST R, x   // store register R to x
```

- The program may then be compiled into many different interleavings

Non-atomic operation

```
P0: LD R, x
P0: INC R
                    P1: LD R, x
                    P1: INC R
P0: ST R, x
                    ST: R, x
```

What is the value of x after this trace?

Atomic update

- To ensure single process update to a shared data area the application needs to manage the mutual exclusion problem!

Mutual exclusion

- Consider n processes that need to exclude concurrent execution of some parts of their code

```
Process Pi
{
entry-protocol
     critical-section
exit-protocol
non-critical-section
}
```

- Fundamental problem to design entry and exit protocols for critical sections

First attempt

```
process P1                       process P2
  loop                             loop
   flag1 = up                       flag2 = up
   while flag2 == up do             while flag1 == up do
      nothing                          nothing
      (* busy waiting *)               (* busy waiting *)
   critical-section                 critical-section
   flag1 = down                     flag2 = down
   non-critical-section             non-critical-section
  end                              end
```

[Dijkstra 1965]

Second attempt

```
process P1                       process P2
loop                             loop
  while flag2 == up do             while flag1 == up do
      nothing                          nothing
      (* busy waiting *)               (* busy waiting *)
  flag1 = up                       flag2 = up
  critical-section                 critical-section
  flag1 = down                     flag2 = down
  non-critical-section             non-critical-section
end                              end
```


Third attempt

```
process P1                  process P2
loop                        loop
  while turn == 2 do          while turn == 1 do
   nothing                     nothing
   (*busy waiting*)            (*busy waiting*)
  critical-section            critical-section
  turn = 2                    turn = 1
  non-critical                non-critical
  section                     section
end                         end
```


Peterson's algorithm

```
process P1
  loop
     flag1 = up
     turn = 2
     while flag2 == up and turn == 2 do
          nothing
     end
     critical-section
     flag1 = down
     non-critical-section
  end
```


Programming language support

- Implementing synchronisation and concurrency with shared variables, using only sequential programming constructs, is difficult and error prone
- Java: Early versions used Suspend/Resume constructs that led to race condition!
- Ada: built-in run-time support with explicit task synchronisation entry points (Rendezvous)
- Both have some support for defining concurrent processes but none has support for real-time in the core language

Ada tasks: example

```
procedure Morning is          task body
  task Get_Breakfast;           Take_Shower is
  task Take_Shower;           begin
task body                       Use_Shower;
  Get_Breakfast is              Dry_Hair;
begin                         end Take_Shower;
  Make_Coffee;
  Make_Toast;                 begin -- Morning
  Boil_Egg;                   null;
  Eat_Breakfast;              end Morning;
end Get_Breakfast;
```

Java threads: Example

```
public class Prepare Extends Thread
{
  private int items;
  public Prepare(int: Breakfast_items)
  {
   items = Breakfast_items
  }
  public void run()
  {
     while(true)
     {
          Cooking.Make(items);
     }
  }
}
```


Java threads: Example

```
final int eggs = 1;
final int toast = 2;
final int coffee = 1;
Prepare P1 = new Prepare(eggs);
Prepare P2 = new Prepare(toast);
Prepare P3 = new Prepare(coffee);
. . .
P1.start();
P2.start();
P3.start();
```


Further reading?

- An equivalent reading material to book chapters posted on the web can be found in an electronically available book from LiU Library through the ebrary service
- It uses Java syntax to describe similar code snippets
- Check out for the link on the literature page on the course web if interested!

Questions?