TDDD36
Secure Mobile Systems
Systems Software
Lecture 1: Processes and shared resources
Simin Nadjm-Tehrani
Real-time Systems Laboratory
Department of Computer and Information Science
Linköping University

Course overview

- Selected topics from core computer systems curriculum that are
  - Essential and you only see in this course
  - Or slightly overlap with later courses but you benefit from while designing and implementing the project

- Concurrency, Processes, and Resource sharing
- Distributed system models
- Dependable systems & Availability
- Adaptive Network resource allocation

People & organisation

- Lecturer and examiner:
  - Simin Nadjm-Tehrani
  - Takes care of the resource session in the schedule too

- Lessons:
  - Laurent Delosières (2 scheduled occasions)

- Expert consultant in the project:
  - Laurent Delosières (until 15th October)
  - Mikael Asplund (after 15th October)
  - Note: different role compared to the theory part of the course!

From last year’s evaluations

We had two points to act on:

- Code examples: some students asked for more example codes in the lectures.
  - Will do!

- Some students were asking for answers to exercises.
  - Will be available but handed out after you have attempted!

Literature & handouts

- Detailed sources and instructions on the web, see also e-book with Java syntax

- Please ask me if you are unsure!

- Note: lecture slide handouts are from last year until one day before the lecture!

Questions?
This lecture

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

The notion of Process

- An abstraction in computer science used for describing program execution and potential parallelism
- What other abstractions do you know?
  - Functions
  - Classes, Objects, Methods
- Processes emphasise the run-time behaviour

Concurrent Programs

A sequential program has a single thread of control.

A concurrent program has multiple threads of control allowing it perform multiple computations “in parallel” and to control multiple external activities which occur at the same time.

[Magee and Kramer 2006]

Concurrency

- A mathematical term for modelling computational processes that could in principle be run in parallel
- Three possibilities:
  - Single processor, shared memory
  - Multi-processor, shared memory
  - Multiple processors not sharing memory

Related terms

<table>
<thead>
<tr>
<th>Concurrent programs</th>
<th>Define actions that may be performed simultaneously</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel programs</td>
<td>A concurrent program that is designed for execution on parallel hardware</td>
</tr>
<tr>
<td>Distributed programs</td>
<td>Parallel programs designed to run on network of autonomous processors that do not share memory</td>
</tr>
</tbody>
</table>

Processes on a single CPU

- 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

Theory on processes in this course covers single CPU
Run-time behaviour

Running processes on a single CPU:

Time

C
B
A

Logical parallelism

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

Which potential execution sequences (traces)?

Concurrent on single CPU

- More efficient use of the processor
- Increased responsiveness, e.g. user interaction can be given priority

Historical data...

- To fully utilise the processor [Wellings 2004]

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

Non-existing
Created
Initialising
Ready
Waiting
Executing
Terminated
Keeping track of execution

- Each process in its 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

Program/process structure

- Static/Dynamic
- Nested/Flat

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

Not part of this course

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 one has its own stack
- Processes that run with a shared address space (code, data) are called light-weight processes or threads
- Here we consider processes with one thread of control and use tasks interchangeably with processes

Context Switch

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

Processes interact

To share data:
- Output for one process may be input for another process
  - Compute distance, and then compute incremental acceleration

To influence flow of control - synchronisation:
- One process may only start after another/others finished
  - Update display only after all sensor values received
- No more than one process at a time may send a packet on a shared channel

Processes must notify each other

This lecture

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

- Two modes:
  - Shared variables
  - Message passing

Mutual exclusion

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

  $P_0$
  $\{x = x + 1\}$

  $P_1$
  $\{x = x + 1\}$

  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 operations

- $P_0: \ LD \ R, x$
- $P_0: \ INC \ R$
- $P_1: \ LD \ R, x$
- $P_1: \ INC \ R$
- $P_0: \ ST \ R, x$
- $P_1: \ ST \ R, x$

  What is the value of $x$ after this trace?

Basic communication operation

- Communication using shared variables

  $P_1$  \hspace{1cm}  $P_2$

  Writes to $X$  \hspace{1cm}  Reads from $X$

  Variable $X$

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
Decoupling from process rates

- Finite buffers

![Diagram: Finite buffer with processes P1 and P2]

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

Need mechanisms for synchronisation and sharing data

General problems

- Conditional action
  - Examples:
    - Compute the interest when all transactions have been processed
    - Book a flight seat only if seats are available

- Mutual exclusion
  - Example:
    - Two customers shall not be booked on the same seat

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, and this is unintended, then the system suffers from a race condition
- Process synchronisation can be used to avoid race conditions
  - Analogy: Deposit 5000 kr in the account, calculate accrued interest

Recap

- Programmer wants to write code for processes that run independently as far as possible
- This makes running on alternative architectures a possibility (recall process as a mathematical abstraction with interleaved traces)
- Process dependencies are unavoidable (sharing values, waiting for conditions to hold)
- But you want to enforce a sequence (wait for conditions) only when it is part of requirements, i.e. avoid race conditions

This lecture

- Processes, threads, and their representation
- Concurrency and execution
- Communication
- Synchronisation
- Solutions to mutual exclusion problem
Atomic update

- To ensure that a single process updates a shared data structure at any one time 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

Critical section: example

```java
Process A
While true do
{
    ...
    X = X+1;
    Z = X^2;
    ... print(X)
    print(Z);
}
```

First attempt

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

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

[Dijkstra 1965]
```

Second attempt

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

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

Third attempt
Peterson's algorithm

```plaintext
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
- Some programming languages have explicit support:
  - 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)

Ada tasks: example

```plaintext
procedure Morning is
    task Get_Breakfast;
    task Take_Shower;
    task body
    Get_Breakfast is
        begin
            Make_Coffee;
            Make_Toast;
            Boil_Egg;
            Eat_Breakfast;
        end Get_Breakfast;
end Morning;
```

Java threads: Example

```java
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

```java
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();
```

Questions?