Lecture 2: Intro to Concurrent Processing

- The SR Language.
- Correctness and Concurrency.
- Mutual Exclusion & Critical Sections.
- Software Solutions to Mutual Exclusion.
- Dekker's Algorithm.
- The Bakery Algorithm.

A Model of Concurrent Programming

- A concurrent program may be defined as the interleaving of sets of sequential atomic instructions.
  - i.e. a set of interacting sequential processes, execute at the same time, on the same or different processors.
  - processes are said to be *interleaved*, i.e. at any given time each processor is executing one of the instructions of the sequential processes.
  - relative rate at which the instructions of each process are executed is not important.
- Each sequential process consists of a series of atomic instructions.
- *Atomic instruction* is an instruction that once it starts, proceeds to completion without interruption.
- Different processors have different atomic instructions , and this can have a big effect.

My First Piece of SR Code

```
Var N : Int := 0;
# Two Processes share a common
# variable

Process P1                    Process P2
      N := N + 1                    N := N + 1
end                           end
```

- Obviously different interleavings can produce different results.
- This code is written in a language called SR or *Synchronising Resources*.
- SR has an exceptionally rich set of concurrency mechanisms.
- It will serve as the main language for demonstrating concurrency in this course.
- Details on SR syntax can be found at `http://elvis.rowan.edu/~hartley/OSusingSR/SR.html`

A Digression into SR

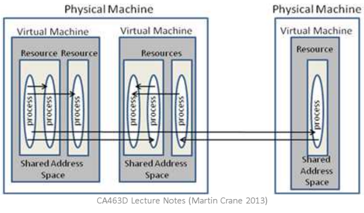
- SR concurrent programming language has been around, in various forms, for a number of years.
- Later versions have provided additional mechanisms for remote procedure call, dynamic process creation, and semaphores, as well as a means for specifying distribution of program modules.
- An SR program can execute within multiple address spaces, located on multiple physical machines.
- Processes within a single address space can also share objects.
- Thus, SR supports programming in distributed environments as well as in shared-memory environments.

A Digression into SR (cont'd)

- SR's model of computation allows a program to be split into one or more address spaces called *virtual machines*.
- Each virtual machine defines an address space on one physical machine.
- Virtual machines are created dynamically and referenced indirectly through *capability variables*.
- Virtual machines contain instances of two related kinds of modular components: **global**s and **resource**s. Hence an SR program is a collection of *resources* and *globals*.

A Digression into SR (cont'd)

- The figure summarizes SR's model of computation.
- In its simplest form, an SR program consists of a single VM running on one physical machine, maybe a shared-memory multiprocessor.
- A program can also consist of multiple virtual machines executing on multiple physical machines.
- Hybrid forms are possible and in fact useful.



A Digression into SR (cont'd)

- Data & processor(s) are shared within a VM; different VMs can be placed on (distributed across) different physical machines.
- Processes on the same or different VMs can communicate through operation invocation.
- Operations may be invoked directly through the operation's declared name or through a resource capability variable or indirectly through an operation capability variable.
- Formally, a *resource* is a template for resource instances from which resource instances can be dynamically created and destroyed.
- A *global* is basically a single, unparameterised, automatically created instance of a resource.

SR: Resources

- A resource is an abstract data object that consists of two parts:
  - a *specification* that specifies the interface of the resource, and
  - a *body* with code implementing the behaviour of abstract data object.
- The general form of a resource is:

```
resource resource_name
       imports # maybe it uses other resources (more later)
       constants, types, or operation declarations
body resource_name (parameters)
       imports
       declarations, statements, procs
       final code
end resource_name
```


SR: Resources (cont'd)

- Some code to define a Stack resource is shown

```
resource Stack
   type results = enum(OK,OFLOW,UFLOW)
   op push (item:int) returns r:result
   op pop (res_item:int) returns
                      r:result

body Stack (size:int)
   var store [1:size]:int, top:int := 0

   proc push (item) returns r
        if top < size ->
                store[++top] := item
                r := OK
        [] top = size ->
                r := OFLOW
        fi
   end

   proc pop (item) returns r
        if top > 0 ->
                item := store[top--]
                r := OK
        [] top = 0 ->
                r := UFLOW
        fi
   end
end Stack
```

SR: Creating Resources Instances

- Since several instances of a resource can be created some mechanism is necessary to distinguish between the different resource instances.
- Done by *resource capabilities*, pointers to a specific resource instance.
- The code below creates 2 instances of a **stack** resource:

```
resource Stack_User
   import Stack         # import as want to use Stack's procs
   var x: Stack.result
   var s1,s2: cap Stack # stack capability variables, each with own
   var y:int            # local vars store & top

   s1:=create Stack(10) # create a stack of size 10
   s2:=create Stack(20) # create a stack of size 20
        ...
   s1.push(4);          # how operations can be ref'd outside Stack via CVs
   s1.push (37); s2.push (98)
   if (x := s1.pop(y)) != OK -> ... fi
   if (x := s2.pop(y)) != OK -> ... fi
        ...
end
```

CA463D Lecture Notes (Martin Crane 2013) 59

SR: Destroying Resources Instances

- The execution of an *SR* program begins with the implicit creation of one instance of the program's `main` resource.
- The initial code of the main resource can in turn create instances of other resources.
- A resource instance can be destroyed by the destroy statement:

  `destroy resource_capability`

- When an *SR* program terminates, the initially created instance of the `main` resource is destroyed after executing its final code.
- This `final` code can in turn destroy other instances of resources.

CA463D Lecture Notes (Martin Crane 2013) 60

SR Processes

- *SR* uses the *process* as its unit of concurrent computation. This is an independent thread of control executing sequential code, with the form

```
process process_name (quantifier, quantifier, ...)
        block
end
```

- The code demonstrates the use of processes for parallel matrix multiplication.

```
resource main( )
   const N := 20
   var a[N,N], b[N,N], c[N,N]: real

   # read in some initial values for a,b
        ...
   # multiply a,b in parallel,result=c      final
                                               # output result in c
   process multiply(i:=1 to N,j:=1 to N)       fa i := 1 to N ->
        var inner_prod:real := 0.0              fa j:= 1 to N ->
                                                      write (c[i, j], ' ')
        fa k := 1 to N ->                       af
           inner_prod+:=a[i,k]*b[k,j]           write
        af                                     af
        c[i,j] := inner_prod                end
   end                                    end mult
```

CA463D Lecture Notes (Martin Crane 2013) 61

A First Attempt to Define Correctness

- If the processor includes instructions like **INC** then this program will be correct no matter which instruction is executed first.
- If all arithmetic must be performed in registers then the following interleaving does not produce the desired results.

```
P1:    load reg,   N
P2:    load reg,   N
P1:    add reg,    #1
P2:    add reg,    #1
P1:    store reg,  N
P2:    store reg,  N
```

- A concurrent program *must be* correct under all possible interleavings.

CA463D Lecture Notes (Martin Crane 2013) 62

Correctness: A More Formal Definition

- If $P(\vec{a})$ is a property of the input (pre condition), and $Q(\vec{a}, \vec{b})$ is a property of the input and output (post condition), then correctness is defined as:
- Partial correctness:

$$P(\vec{a}) \wedge \text{Terminates}\{Prog(\vec{a}, \vec{b})\} \Rightarrow Q(\vec{a}, \vec{b})$$

- Total correctness:

$$P(\vec{a}) \Rightarrow [\text{Terminates}\{Prog(\vec{a}, \vec{b})\} \wedge Q(\vec{a}, \vec{b})]$$

- Totally correct programs terminate. A totally correct specification of the incrementing tasks is:

$$a \in \mathbb{N} \Rightarrow [\text{Terminates}\{\mathbf{INC}(a, a)\} \wedge a = a + 1]$$

CA463D Lecture Notes (Martin Crane 2013) 63

Types of Correctness Properties

There are 2 types of correctness properties:

1. Safety properties	These must *always* be true.
Mutual exclusion	Two processes must not interleave certain sequences of instructions.
Absence of deadlock	Deadlock is when a non-terminating system cannot respond to any signal.
2. Liveness properties	These must *eventually* be true.
Absence of starvation	Information sent is delivered.
Fairness	That any contention must be resolved.

CA463D Lecture Notes (Martin Crane 2013) 64

Correctness: Fairness

- There are 4 different way to specify *fairness*.
  - *Weak Fairness* — If a process continuously makes a request, eventually it will be granted.
  - *Strong Fairness* — If a process makes a request infinitely often, eventually it will be granted.
  - *Linear waiting* — If a process makes a request, it will be granted before any other process is granted the request more than once.
  - *FIFO* — If a process makes a request, it will be granted before any other process makes a later request.

Mutual Exclusion

- A seen, a concurrent program must be correct in all allowable interleavings.
- So there must be some sections of the different processes which cannot be allowed to be interleaved.
- These are called *critical sections*.
- We will attempt to solve the mutual exclusion problem using software first before more sophisticated solutions.

```
# A critical section shared by different processes
  do true ->
      Non_Critical_Section
      Pre_protocol
      Critical_Section
      Post_protocol
  od
```


Software Solutions to Mutual Exclusion Problem # 1

```
#First proposed solution

var Turn: int := 1;

process P1                          process P2
   do true ->                          do true ->
        Non_Critical_Section                Non_Critical_Section
        do Turn != 1 -> od                  do Turn != 2 -> od
        Critical_Section                    Critical_Section
        Turn := 2                           Turn := 1
   od                                  od
end                                 end
```

- This solution satisfies mutual exclusion. ☑
- Cannot deadlock, as both P1, P2 would have to loop on `Turn` test infinitely and fail.
  - Implies `Turn = 1` and `Turn = 2` at the same time.
- No starvation: requires one task to execute its CS infinitely often as other task remains in its pre-protocol.
- Can fail in the absence of contention: if P1 halts in CS, P2 will always fail in pre-protocol.
- Even if P1, P2 are guaranteed not to halt, both processes are forced to execute at the same rate. This, in general, is not acceptable.

Software Solutions to Mutual Exclusion Problem # 2

```
# Second proposed solution

var C1:int := 1
var C2:int := 1

process P1
    do true ->
         Non_Critical_Section
         do C2 != 1 ->
         od
         C1 := 0
         Critical_Section
         C1 := 1
    od
end

process P2
    do true ->
         Non_Critical_Section
         do C1 != 1 ->
         od
         C2 := 0
         Critical_Section
         C2 := 1
    od
end
```

- The first attempt failed because both processes shared the same variable.
- The Second Solution unfortunately violates the mutual exclusion requirement.
- To prove this only need to find one interleaving allowing P1 & P2 into their CS at same time.
- Starting from the initial state, we have:

P1 checks C2 and finds C2 = 1.
P1 sets C1 = 0.
P1 enters its critical section.

P2 checks C1 and finds C1 = 1.
P2 sets C2 = 0.
P2 enters its critical section.
QED

Software Solutions to Mutual Exclusion Problem # 3

```
var C1:int := 1
var C2:int := 1

process P1
    do true ->
        Non_Critical_Section # a1
        C1 := 0              # b1
        do C2 != 1 ->        # c1
          od
        Critical_Section     # d1
        C1 := 1              # e1
    od
end

process P2
    do true ->
        Non_Critical_Section # a2
        C2 := 0              # b2
        do C1 != 1 ->        # c2
                 od
        Critical_Section     # d2
        C2 := 1              # e2
    od
end
```

- The problem with the last attempt is that once the pre-protocol loop is completed you cannot stop a process from entering its critical section.
- So the pre-protocol loop should be considered as part of the critical section.
- We can prove that the mutual exclusion property is valid. To do this we need to prove that the following equations are *invariants*:

$$\mathrm{C1} = 0 \equiv at(c_1) \vee at(d_1) \vee at(e_1) \qquad \text{Eqn(1)}$$
$$\mathrm{C2} = 0 \equiv at(c_2) \vee at(d_2) \vee at(e_2) \qquad \text{Eqn(2)}$$
$$\neg\{at(d_1) \wedge at(d_2)\} \qquad \text{Eqn(3)}$$

(here $at(x) \Rightarrow$ x is the next instruction to be executed in that process.)

Software Solutions # 3 (cont'd)

- Eqn (1) is initially true:
  - Only the $b_1 \rightarrow c_1$ and $e_1 \rightarrow a_1$ transitions can affect its truth.
  - But each of these transitions also changes the value of `C1`.
- A similar proof is true for Eqn (2).
- Eqn 3 is initially true, and
  - can only be negated by a $c_2 \rightarrow d_2$ transition while $at(d_1)$ is true.
  - But by Eqn (1), $at(d_1) \Rightarrow$ `C1=0`, so $c_2 \rightarrow d_2$ cannot occur since this requires `C1=1`. Similar proof for process P2.
- But there's a problem with deadlock, if the program executes one instruction from each process alternately:

P1 assigns 0 to C1.
P1 tests C2 and remains in its **do** loop

P2 assigns 0 to C2
P2 tests C1 and remains in its **do** loop

Result Deadlock!

Software Solutions to Mutual Exclusion Problem # 4

- Problem with third proposed solution was that once a process indicated its intention to enter its CS, it also **insisted** on entering its CS.
- Need some way for a process to relinquish its attempt if it fails to gain immediate access to its CS, and try again.

Software Solutions to Mutual Exclusion Problem # 4

```
var C1:int := 1
var C2:int := 1

process P1
    do true ->
        Non_Critical_Section
        C1 := 0
        do true ->
            if C2 =1->exit fi
            C1 := 1
            C1 := 0
        od
        Critical_Section
        C1 := 1
    od
end

process P2
    do true ->
        Non_Critical_Section
        C2 := 0
        do true ->
            if C1 =1->exit fi
            C2 := 1
            C2 := 0
        od
        Critical_Section
        C2 := 1
    od
end
```

- This proposal has two drawbacks:

1. A process can be starved.

   Can find interleavings where a process can never enter its critical section.

2. The program can *livelock*.

   This is a form of deadlock. In deadlock there is no possible interleaving which allows the processes to enter their CS. In livelock, some interleavings succeed, but there are sequences which do not succeed.

Software Solutions # 4 (cont'd)

Proof of Failure of Attempt 4:

1. By Starvation

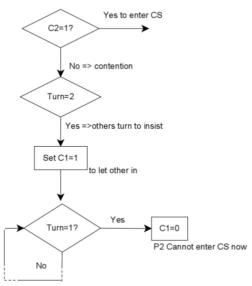
P1 sets C1 to 0.	P2 sets C2 to 0
P1 completes a full cycle:	P2 checks C1, sees C1=0 & resets C2 to 1
Checks C2	
Enters Critical Section	
Resets C1	
Executes non-Critical Section	
Sets C1 to 0	P2 sets C2 to 0 and back

2. By Livelock

P1 sets C1 to 0.	P2 sets C2 to 0
P1 tests C2 and remains in its **do** loop	P2 tests C1 and remains in its **do** loop
P1 resets C1 to 1 to relinquish attempt to enter CS	P2 resets C2 to 1 to relinquish attempt to enter CS
P1 sets C1 to 0	P2 sets C2 to 0
	etc

Dekker's Algorithm

- A combination of the first and fourth proposals:
  - The first proposal explicitly passed the right to enter the CSs between the processes,
  - whereas the fourth proposal had its own variable to prevent problems in the absence of contention.
- In Dekker's algorithm the right *to insist* on entering a CS is explicitly passed between processes.



Dekker's Algorithm (cont'd)

```
var C1:int := 1
var C2:int := 1
var Turn:int := 1

process P1
   do true ->

       Non_Critical_Section
       C1 := 0
       do true ->
          if C2 = 1-> exit fi
          if Turn = 2 ->
               C1 := 1
               do Turn !=1 -> od
               C1 := 0
          fi
       od
       Critical_Section
       C1 := 1
       Turn := 2
   od
end

process P2
   do true ->

       Non_Critical_Section
       C2 := 0
       do true ->
          if C1 = 1-> exit fi
          if Turn = 1 ->
               C2 := 1
               do Turn !=2 -> od
               C2 := 0
          fi
       od
       Critical_Section
       C2 := 1
       Turn := 1
   od
end
```


Mutual Exclusion for n Processes: The Bakery Algorithm

- Dekker's Algorithm is the solution to the mutual exclusion problem for 2 processes.
- For the *N* process mutual exclusion problem, there are many algorithms; all complicated and relatively slow to other methods.
- One such is the *Bakery Algorithm* where each process takes a numbered ticket (whose value constantly increases) when it wants to enter its CS.
- The process with the lowest current ticket gets to enter its CS.
- This algorithm is not practical because:
  - the ticket numbers will be unbounded if some process is always in its critical section, and
  - even in the absence of contention it is very inefficient as each process must query the other processes for their ticket number.

Mutual Exclusion for *N* Processes: The Bakery Algorithm (cont'd)

```
var Choosing: [N] int
var Number: [N] int

# Choosing and Number arrays initialised to zero

process P(i:int)
   do true ->
        Non_Critical_Section
        Choosing [i] := 1
        Number [i] := 1 + max (Number)
        Choosing [i] := 0
        fa j := 1 to N ->
           if j != i ->
                do Choosing [j] != 0 -> od
                do true ->
                     if (Number [j] = 0) or (Number [i] < Number [j]) or
                         ((Number [i] = Number [j]) and (i < j)) -> exit
                     fi
                od
           fi
        af
        Critical_Section
        Number [i] := 0
   od
end
```

CA463D Lecture Notes (Martin Crane 2013) 77