# CS 3211 – Parallel & Concurrent Programming Introduction

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

**Also see IVLE Lesson Plan.**



## Sequential Programming

- Single thread of control flow.
- One program counter.
  - Advances by executing an instruction.
- Standard programming languages
  - C, Java.
  - Sequential Java program may have many passive objects
    - Only one active flow of control.

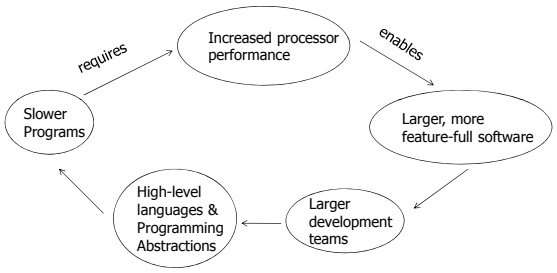


## Why Concurrent Programming?

- Wide rise of multi-cores
  - Machines with 4 or more cores are common.
  - Intel already has a processor with 80+ cores !!
  - Latest news: 48 core processor for smart-phones
    - http://www.techspot.com/news/50665-intel-is-developing-a-48-core-processor-for-smartphones-and-tablets.html
  - Why so ?
    - Processor speeds have enabled more complex programming languages and tasks.
    - But, is this not self-sustaining?



## The cycle [Larus, MSR-TR 08]






## Free lunch must end!

`` For the past three decades, improvements in semiconductor fabrication and processor implementation produced steady increases in the speed at which computers executed existing sequential programs. The architectural changes in multicore processors benefit only concurrent applications and therefore have little value for most existing mainstream software. For the foreseeable future, today's desktop applications will not run much faster than they do now. In fact, they may run slightly slower on newer chips, as individual cores become simpler and run at lower clock speeds to **reduce power consumption** on dense multicore processors ….That brings us to a fundamental turning point in software development, at least for mainstream software. Computers will continue to become more and more capable, but programs can no longer simply ride the hardware wave of increasing performance unless they are highly concurrent. Although multi-core performance is the forcing function, we have other reasons to want concurrency: notably, to improve responsiveness by performing work asynchronously instead of synchronously. For example, today's applications must move work off the GUI thread so it can redraw the screen while a computation runs in the background."

- *from "Software and the Concurrency Revolution", ACM Queue 05.*

- **[By now, the Free Lunch has ended! ]**



## On concurrency

- Being integrated into mainstream languages
  - Java, C#
- Harder to program and understand
  - Many inter-leavings even when each thread has one path.
  - Cyclic debugging not possible – cannot reproduce an observable error !

| Thread 1 | Thread 2 |
|---|---|
| X = 1; // should be 0 | X = 2;<br>Y = X;<br>printf("%d", y); |

## Possible runs

- X = 1;
- X = 2;
- Y = X;
- Print Y  Error not exhibited.

- X = 2;
- X= 1;
- Y = X;
- Print Y  Error is exhibited.



## In this course

- Concurrent Programming (primarily)
  - Principles, rather than tricks
    - Sometimes high-level modeling languages used to convey principles.
    - Java is used to illustrate implementation issues.
  - 2/3 of the course
- Parallel Programming (approx 3 lectures)
  - Material will be given in E-reserves
  - 1/3 of the course.



## Books

- Textbook (closely followed for Concurrent Programming, but no coverage of parallel programming)
  - *Concurrency: State Models & Java Programs* by Jeff Magee and Jeff Kramer
    - Publisher: Wiley
    - ISBN 0-471-98710-7
  - **Available in Forum Co-op**.
- Parallel Programming
  - Principles of Parallel Programming, by
  - Calvin Lin and Lawrence Snyder
  - [*Some material in E-reserves*]



## Topics (1)

- Concurrency as a concept
  - Threads/Processes
  - Interleaving among threads
  - Communication mechanisms among threads
    - Shared Objects
    - Message Passing
- A glimpse of these concepts today



## Topics (2)

- Thread Communication in details
  - Shared obj. & Mutual exclusion
  - Monitors
  - Properties to preserve
    - No deadlock, Safety, Liveness
- Multi-threaded Java will be used in these assignments.

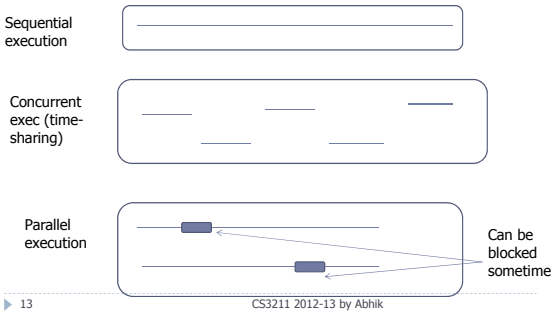


## Topics (3)

- Parallel Programming
  - Libraries to extend a sequential programming language
  - Message Passing Interface (MPI) on top of C
- Suggested text (some material to be put in E-reserves)
  - Principles of Parallel Programming, by
  - Calvin Lin and Lawrence Snyder
- Parallel programming constitutes about 1/3 of course.
  - **Assignment will involve C programming.**
  - **No tutorial on C programming will be given.**

## Pre-module survey I had given

A multi-threaded program may have concurrent execution or parallel exec depending on # of processor cores available.






## Assessment

- Mark distribution
  - Midterm: 20% [on the 7th week, in class, 7th March 2013]
  - Programming Assignments: 30%
    - Concurrent Programming: 2 assignments, 10 marks each = 20 marks
    - Parallel Programming: 1 assignment, 10 marks = 10 marks
  - Tutorial participation: 5%
  - Final : 45%
- Pre-requisites: CS2106 or CG2271
- IVLE Lesson Plan contains a lot of information.
  - Please check the Lesson Plan regularly.



## Lectures – post-it notes

- Please attend, and engage.
- I will put around 50 - 60 post it notes in different seats in the lecture hall.
- Please try to take up a seat with a post-it note.
- Any concept that you are unclear about – you can mark it in the post-it note, and post it in the lecture hall door while you leave.
- This will give me an idea about which topics to revise and/or follow up, possibly in the tutorials.



## Tutorials

- Kindly attend, and participate.
  - The 5% is for participation, not for giving correct answers.
  - So, kindly do not hesitate to participate – even if you think your answer may not be the correct one!
- Questions (without answers) will be posted.
  - Answers will be posted just before midterm, and just before final – see my first IVLE announcement [this is what I prefer]
  - Any comments on this?
  - How to address the concern of people who attend!
- Please raise any questions you may have about past lectures. These could be topics you mention in post-it.



## Additional Sessions

- We have the first one this week itself.
  - Saturday Jan 19 10 am – 12 noon at MR1 COM1 03-19
    - No new material is covered.
    - Meant as a general and dynamic discussion of all the concepts.
- Not held every week
  - I know you may not want to come on Saturdays ☺
  - Next one is in week 5, Feb 16 at Executive Classroom COM2-04-02. That session will cover ---
    - Make-up tutorial for CNY, followed by an
    - Optional session to discuss all the concepts up to then.



## The people

- My e-mail: abhik@comp.nus.edu.sg
  - Office: COM2 #03-07
  - [**Please email me if you want to meet for consultation**]
- Your Tutors [responsible for tutorials]
  - Myself
  - Dr. Jooyong Lee (Yi) jooyong@comp.nus.edu.sg
  - Qi Dawei dawei@comp.nus.edu.sg
- Queries, and help with assignments
  - Post your queries to the IVLE Discussion forum.
  - Also, for each of the 3 assignments we have a **primary contact** – who will grade the assignment. Queries may be directed to him.
    - Assignment 1: Myself
    - Assignment 2: Jooyong Lee (Yi)
    - Assignment 3: Qi Dawei

# A Concurrent Modeling Language

Abhik Roychoudhury
CS 3211
Department of CS, NUS

**Reading for this portion appears in E-reserves, see Lesson Plan.**



## We now discuss …

- SPIN --- a tool for modeling complex concurrent and distributed systems.
- Provides:
  - Promela, a protocol meta language
  - A checker
  - A random simulator for system simulation
- Why discuss it now?
  - To introduce the concepts in concurrency …
  - Without getting into full-scale multi-threaded Java programming at the very beginning.



## What is this modeling language?

- Describes concurrent systems
  - Depicts common concepts in concurrency
    - Threads / processes
    - Interleaving among threads/processes
    - Inter-process communication via shared variable updates
    - Inter-process communication via message passing
  - … and also other features such non-determinism within a process
    - Only in a modeling language.
- Yet, is higher-level than a programming language
  - Focus on concurrency concepts first, rather than details of Java

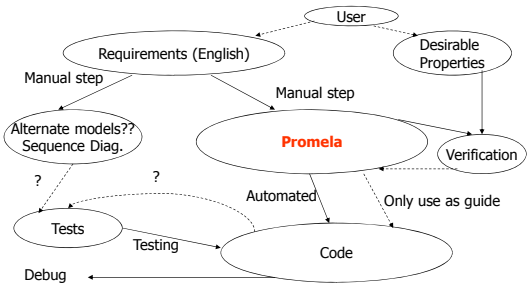


## Our Usage

- Learn Promela, a modeling language.
  - Higher-level than a programming language.
- Use it to model simple concurrent system protocols and interactions.
- Gives a feel (at a small scale)
  - What are hard-to-find errors in concurrent programming?
  - Supported by a back-end checker which can show the error traces to you
    - Each error trace is shown as a UML Sequence Diagram!



## Our primary usage






## Why Promela ?

- Specification language to model *finite state* systems
  - Side remark: What is finite state?
- Models finite state concurrent processes which compute and communicate.
- Different flavors of concurrency & communication
  - Via global shared variables.
  - Via message channels
    - Synchronous communication (hand-shake)
    - Asynchronous communication (buffers)

## Example 0

```
byte state = 0;

proctype A()
{  byte tmp;

     (state==0) -> tmp = state;
      tmp = tmp+1;
      state = tmp;
}

init  { run A() ;  }
```

**state** : Global Variable

**tmp** : Local Variable

**(state==0) -> tmp = state** is a guarded command (blocked if the guard is false).

Only one process created.

Final value of **state** is 1



## Concepts in Example 0

```
byte state = 0;

proctype A()
{  byte tmp;

     (state==0) -> tmp = state;
      tmp = tmp+1;
      state = tmp;
}

init  { run A() ;  }
```

*Plain sequential programming.*

*Use of guarded commands.*

*Only one active thread of control.*



## Looking inside a process

- Data Structures
  - Basic types : int, bool, bit, byte
  - Arrays
  - Structures (through typedef declarations)
- Just as in C/Java, not much going on here !
- Check SPIN manual for details
  - http://spinroot.com/spin/Man/Manual.html



## Statements

- Assignments
- Boolean expressions
  - If true, then no-op else block
- Guarded commands
  - (state == 1) ->  tmp = state;
  - Guard and body evaluated separately, be careful !!
  - If you want to evaluate them together
  - atomic {  (state ==  1) -> tmp = state; }
  - Effect of a test-and-set instruction



## Example 1

```
byte state = 0;

proctype A()
{  byte tmp;

     (state==0) -> tmp = state;
     tmp = tmp+1; state = tmp;
}

init  { run A() ; run A();  }
```

What will happen here ?

We need to define how processes are scheduled to determine behaviors.



## Process scheduling

- All processes execute concurrently
- Interleaving semantics
  - At each time step, only one of the "active" processes will execute (non-deterministic choice here)
  - A process is active, if  it has been created, and its "next" statement is not blocked.
  - Each statement in each process executed atomically.
  - Within the chosen process, if several statements are enabled, one of them executed non-deterministically.
    - We have not seen such an example yet !

## Slight de-tour in first lecture

Discusses

-Interleavings
-Data races
-Use of locks to prevent races



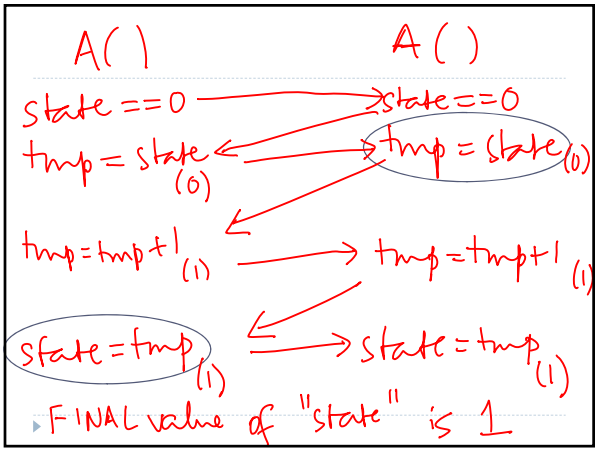
## Example 1 - Revisited

```
byte state = 0;

proctype A()
{ byte tmp;

    (state==0) -> tmp = state;
    tmp = tmp+1; state = tmp;
}

init { run A() ; run A(); }
```

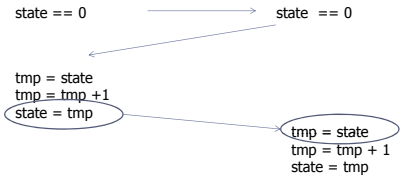
Final val. of state can still be 1 ??

Problem of arbitrary shared variable access by several threads.






## Another interleaving




Ordering of these two operations do matter!
They are in a "race"
Depending on who wins the race – the final observed value is different.



## Shared variable operations

- Consider pair of shared variable operations in 2 threads
  - X = 1 X = 2
    - Depending on which is executed earlier, final value is different
  - X = 1 read X
  - A more concrete version
  - X = 1 if (X > 1){
    - Depending on which is executed earlier, different value is read
  - read X read X or
  - if (X > 1){ if (X > 2){
    - No race between this pair.



## Data races

- A data race occurs when:
  - two or more threads access the same memory location concurrently, and
  - at least one of the accesses is for writing, and
  - the threads are not using any exclusive locks to control their accesses to that memory.
- When these three conditions hold, the order of accesses is non-deterministic. Many data-races are bugs in the program.

## What can you do as a programmer?

- Enclose all shared variable accesses with locks – Java provides this facility (next lecture)
  - Acquire lock before access
  - Release lock after access
- May lead to very inefficient code?
  - Can acquire locks for an entire block of code.
  - What impact does it have on concurrency?
- Alternative – Java volatile variables [not recommended]
  - More details in next week's lecture.



## Concepts in Example 1

```
byte state = 0;

proctype A()
{ byte tmp;

    (state==0) -> tmp = state;
    tmp = tmp+1; state = tmp;
}

init { run A() ; run A(); }
```

*Several threads of control.*

*Interleaved execution among threads.*

*Shared variables for inter-thread communication.*

*Surprising results due to unforeseen interleavings !!*



## End of De-tour

Initial discussion on

-Interleavings
-Data races
-Use of locks to prevent races



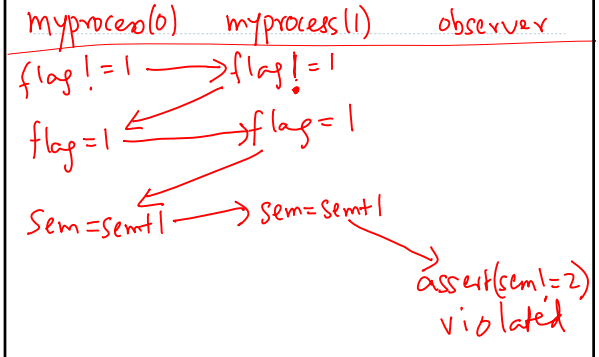
## Example 2

```
bit  flag;
byte sem;
proctype myprocess(bit i)
{  (flag != 1) -> flag = 1;
   sem = sem + 1;
   sem = sem – 1;
   flag =  0;
}
proctype observer() {
    assert( sem != 2 );
}
```

```
init {
   atomic{
        run myprocess(0));
        run myprocess(1));
        run observer();
   }
}
```

All three processes Instantiated together






## Concepts in Example 2

- Interleaved execution among threads.
- Shared variable communication
- Unintended shared variable values
  - Due to unforeseen interleavings
- And, a mechanism for
  - **Specifying the unintended behavior.**
  - Producing the interleaving that produces this unintended behavior.
  - We only do this in our modeling environment – hard to do this for real programs!

## Concepts in Example 2

- Initial values of sem, flag not given
  - All possible initial values are considered.
- The system being verified is the asynchronous composition
  - myprocess(0) || myprocess(1)
- The property is the invariant
  - always sem ≠ 2
- Local & global invariants can be specified inside code via assert statements.



## More on assert

- Of the form assert B
  - B is a boolean expression
  - If B then no-op else abort (with error).
- Can be used inside a process (local invariants)
  - proctype P( … ) { x = … ; assert( x != 2); ….}
- Or as a separate observer process (global invariants)
  - proctype observer(){ assert(x != 2);}
- Used to specify intended (and unintended) behaviors resulting from interleavings among threads.



## Example 3

```
bit flags[2];
byte sem, turn;
proctype myprocess(bit id) {
   flags[id] = 1;
   turn = 1 – id;
   flags[1-id] == 0 || turn == id;
   sem++;
   sem--;
   flags[id] = 0;
}
```

```
init() {
   atomic{
      run myprocess(0);
      run myprocess(1);
      run observer(); }
}
proctype observer() {
      assert( sem != 2 );
}
```



## Issues

- Can you prove mutual exclusion ?
  - What purpose does turn serve ?
- Arrays have been used in this example.
  - Flags is global, but each element is updated by only one process in the protocol
  - Not enforced by the language features.
- Processes could alternatively be started as:
  - active proctype myprocess(…) {
  - Alternative to dynamic creation via run statement



## So far …

- Process creation and interleaving.
- Process communication via shared variables.
- Standard data structures within a process.
- Assignment, Assert, Guards.
- NOW …
  - Guarded IF and DO statements
    - Within a process, if several statements are enabled, one of them executed non-deterministically!
  - Channel Communication between processes



## Non-deterministic choice

- Choice of statements within a process
  - if
  - :: condition$_1$ -> … ; … ; …
  - …
  - :: condition$_k$ -> … ; … ; …
  - fi;
- If several conditions hold, select and execute any one (more behaviors for verification).
- If none hold, the statement blocks.

## Loops

- Similar to the if-fi statement, we have a do-od statement.
- Repeat the choice selection forever.
  - Useful for modeling infinite loops pre-dominant in control software.
- Control can transfer out of the loop via a break statement in the flavor of the C language.



## A loop which may terminate

```
byte count;

        proctype counter()
        {
                do
                :: count = count + 1
                :: count = count - 1
                :: (count == 0) -> break
                od;
        }
```

Enumerate the reasons for non-termination in this example



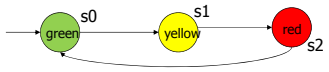
## Concepts in previous example

- Non-determinism within a process
  - Not normal for threads in programs!!
  - A model is often, less detailed than a program.
- Possibility of programming non-terminating control software
  - See next example too.



## A loop which will not terminate

```
active proctype TrafficLightController() {
    byte color = green;
    do
     :: (color == green) -> color = yellow;
     :: (color == yellow) -> color = red;
     :: (color == red) -> color = green;
     od;
}
```






## Synchronization

- Processes implicitly communicate via shared variables.
- However, for other reasons
  - Processes may need to explicitly synchronize.
- What reasons?
  - e.g. Mutually exclusive access to shared variables.
- How to synchronize?
  - Acquiring and releasing locks.



## Locking

```
byte sem = 1;

active proctype P() {
    do
    :: printf("Noncritical section P\n");
       atomic{ sem == 1; sem--; }
       printf("Critical section P\n");
       sem++;
    od
}

init{
    atomic{ run P(); run P(); }
}
```

atomic{ sem == 1; sem--; } Implementation of lock acquire
sem++ Implementation of lock release

When we program in Java, we do not program in the protocols for ensuring mutual exclusion. Instead, we assume a locking mechanism and program the non-critical / critical sections.

## So far …

- Process creation and interleaving.
- Process communication via **shared variables**.
- Standard data structures within a process.
- Assignment, Assert, Guards.
- Guarded IF and DO statements
- NOW …
  - Channel Communication between processes

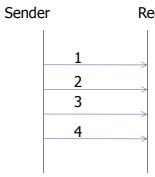


## Channels

- Processes in our modeling language can communicate by exchanging messages across channels.
- Channels are typed.
- Any channel is a FIFO buffer.
- Handshakes supported when buffer is null.
- chan ch = [2] of bit;
  - A buffer of length 2, each element is a bit.
- Array of channels also possible.
  - Talking to different processes via dedicated channels.



## Handshake or not?




Handshake communication
<!1, ?1>, <!2, ?2>, <!3, ?3>, …
(only possible interleaving)
Buffer of length 2
!1, !2, ?1, !3, ?2, …
(also possible)

What is the minimum sized buffer needed to allow this interleaving?

!1, !2, ?1, ?2, !3, !4, ?3, ?4, …



## Value-passing

```
chan ch =  [0] of bit;
active proctype sender()
{
    ch!1;
}
```

```
active proctype receiver()
{  bit  x;
   ch?x;
   printf("%d", x);
}
```

The value 1 is passed into local var. x via message passing.

In this example, the message passing was via a handshake
**! is output, ? is input**

Receiving is always blocked if the corresponding channel buffer is empty. Similarly for sending.



## An example with channels

```
chan name = [??] of byte;
proctype A() {
name!124;
name!121;
}

proctype B() {
    byte state;
    name?state
}
```

```
init { atomic { run A(); run B() } }
```

Enumerate the behaviors when:

?? is 0

?? is 1

?? is > 1



## Another (more famous) example

```
#define p 0
#define v 1
chan sema = [0] of { bit };
proctype dijkstra_semaphore() {
byte count = 1;
do
:: (count == 1) -> sema!p; count = 0
:: (count == 0) -> sema?v; count = 1
od
}
```

```
proctype user()
{
 do
 :: sema?p;  /* critical section */
    sema!v; /* non-critical section */
od
}

init {
    run dijkstra_semaphore();
    run user(); run user(); run user()
}
```

## Any comments?

Mutual Exclusion

- Can more than one process enter the critical section?
  [note that the guard and body of a guarded command can be executed non-atomically]

Non-starvation

- Can one process completely hog the critical section, and starve out other processes?



## Communication among processes

- Shared variables
  - (as in concurrent programming in Java)
- Message Passing
  - (we will later use MPI for parallel programming)
  - At the application level, the issue of locking does not arise.
    - Seemingly, no shared variables !
    - So, we do not need to worry about this now!
  - However, in reality, the message buffers or channels are shared global variables and the programmer will need some mechanism to mutually ensure exclusive access
    - Two processes cannot read/write to the channel at the same time.

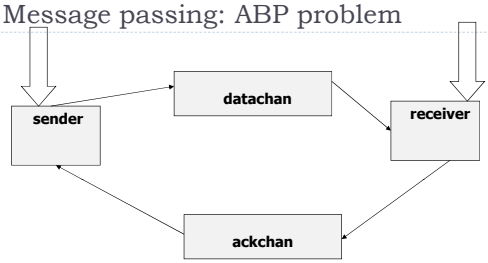


## How Message Passing occurs in real-life

- Interrupt-driven communication
  - An interrupt happens to the CPU, whenever data is ready to be read.
    - To ensure mutually exclusive access of message buffers, disable interrupts while servicing the current interrupt.
    - Not captured at the application level send-receive we are studying!
- Or, the CPU polls (via certain sensors) at regular intervals to check whether data is available
  - Check whether data is available on the channel and then perform receive action, popularly known as polling.
  - Instead of being blocked at request?client as if the server checks periodically if the client has sent its data.



## Message passing: ABP problem




Alternating Bit Protocol
Reliable channel communication between sender and receiver.
Exchanging msg and ack. Channels are lossy .
Attach a bit with each msg/ack. Proceed with next message if the received bit matches your expectation.



## ABP modeling

```
active proctype Sender()
{   bit out, in;
    do
    :: datachan!out ->
            ackchan?in;
            if
            :: in == out
                 -> out = 1- out;
            :: else fi
     od
}
```

```
active proctype Receiver()
{   bit in ;
    do
    :: datachan?in -> ackchan!in
    :: timeout -> ackchan!in
    od
}
```

```
chan datachan = [2] of { bit };
chan ackchan = [2] of { bit };
```



## Let us finish with a real-life situation

- July 4, 1997
  - NASA's Pathfinder landed on Mars.
  - Tremendous engineering feat.
  - Hard to design the control software with concurrency and priority driven scheduling of threads.
  - The SpaceRover would lose contact with earth in unpredictable moments.

## Mars PathFinder Problem

"But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once"." …



## Essence of the problem in our modeling language

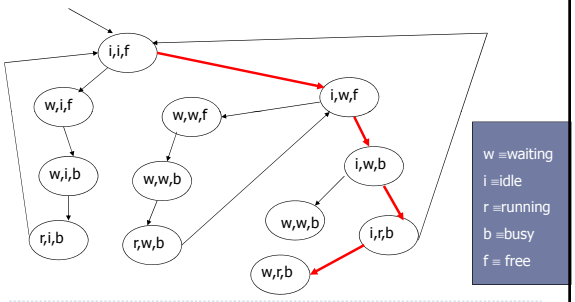
```
mtype = { free, busy, idle, waiting, running };
mtype H = idle;   mtype L = idle; mtype mutex = free;
```

```
active  proctype high();
{end:  do
       :: H = waiting;
          atomic { mutex == free ->
                   mutex = busy };
          H = running;
          atomic{ H=idle; mutex=free }
       od
}
```

```
active proctype low() provided (H == idle)
{ end: do
       :: L = waiting;
          atomic{ mutex== free->
                   mutex = busy};
          L = running;
          atomic{ L=idle; mutex = free }
    od
}
```



## State Space Graph






## Deadlock

- Counterexample
  - Low priority thread acquires lock
  - High priority thread starts
  - Low priority process cannot be scheduled
  - High priority thread blocked on lock
- Actual error was a bit more complex with three threads of three different priorities
  - Timer went off with such a deadlock resulting in a system reset and loss of transmitted data.



## Readings for today's lecture

- Basic SPIN manual
  - http://spinroot.com/spin/Man/Manual.html
  - Promela is the front end of the SPIN tool, a model checker. We will concern ourselves mostly about the modeling in cs3211.
- Chapter 3 of "The SPIN Model Checker" by Gerard J. Holzmann.
  - Scanned version made available from IVLE Lesson Plan (E-reserves).
- Lot of other material available online at
  - http://spinroot.com/spin/Man/index.html



## Until the …

- Additional session on Saturday 19 Jan, 10 am – 12 noon.
  - Question for us to ponder about --
    - *Locks ensure mutual exclusion. How to ensure mutual exclusion of the lock variable / object?*
    - *A common concept in concurrency is a "deadlock" where in its simple form, two processes wait for each other. Think of some scenarios where lock usage leads to deadlock – we just saw one!*