Module Folder submitted by Abhik Roychoudhury

CS3211 Parallel and Concurrent Programming

Email: abhik@comp.nus.edu.sg
http://www.comp.nus.edu.sg/~abhik

Course Contents:

In this folder, I will present my teaching of

CS 3211 Parallel and Concurrent Programming

A brief description of the course is given as follows.

A concurrent system consists of a set of processes that executes simultaneously and that may collaborate by communicating and synchronizing with one another. Examples of concurrent systems are parallel programs that describe sets of collaborating processes. This module introduces the design, development and debugging of parallel programs. It will build on the concurrency concepts gained from the Operating Systems module. It covers concepts and modeling tools for specifying and reasoning (about the properties of) concurrent systems and parallel programs. It also covers principles of performance analysis, asynchronous and asynchronous parallel programming, and engineering concurrent systems and parallel programs.

Teaching Style:

The focus of this module is to expose the students to the fundamental notions of concurrency and parallelism. Prior to this module, they are exposed to concurrency via an Operating Systems module. However, CS 3211 is the first time where the students get to study concurrency as the main focus of the module. This is a substantial deviation in mind-set for students who are exposed and used to sequential programming.

Many first courses focusing on concurrency tend to become overly theoretical – zooming into discussions on concurrent system behaviors, behavioral equivalence, and formal verification of concurrent systems. As a result, the attention completely shifts away from programming. When I took over the module, I changed its contents to make sure that the focus firmly remains on concurrent and parallel programming, while the students get a good grasp of the fundamental concepts in concurrency and parallelism.

Parallel/concurrent programming usually allows for one of the two most common modes of communication between processes/threads --- namely shared memory and message passing. I have made sure that the students get exposed to both of these styles via CS 3211.

I have also encouraged the students to look beyond concurrent programming stereo-types such as equating locks with mutual exclusion. Instead the students are encouraged to *understand* the role of each concurrency construct – such as understanding locks as a construct which (i) can be used to enforce mutual exclusion only if a consistent locking discipline is followed, (ii) allows for atomic execution of a block of code, and (iii) allows for visibility of updates when the lock is relinquished.

Student Learning Objectives:

The learning objectives of CS 3211 were as follows:

(a) Exposure to fundamental concepts in concurrency and parallelism – threads and processes, inter-leavings, locks, barriers, deadlocks, live-locks.
(b) Clear hands-on training on the fundamental concepts via popular programming languages and environments such multi-threaded Java for concurrent programming, and Message Passing Interface (MPI) for parallel programming

Assessment Plan:

The assessment plan was geared towards a healthy mix of continuous evaluation and examinations. Apart from *one* midterms and a final examination, the students did hands-on work. The hands-on work comes in the form of three different labs or programming assignments.

- Lab1 – Assignment on concurrency and modeling
- Lab2 – Assignment on multi-threaded programming with shared variable and locks
- Lab3 – Parallel programming using MPI.

The overall division is thus as follows – Midterm 20%, Final 50%, Labs 30% (three labs, each 10%)

Student Profile and Student feedback:

The students in CS3211 come from Computing program, as well as Computer Engineering program. Following are my feedback ratings, as well as some qualitative feedback received in the module.

Academic Year	CS3211 Enrolment	Abhik's feedback rating in CS3211
2009-10	81	4.19
2012-13	45	4.07

In between my two offerings of the module, another lecturer taught the module, and his feedback ratings are also provided in the following.

Academic Year	CS3211 Enrolment	Other lecturer's feedback rating in CS3211
2010-11	54	3.82
2011-12	39	4.00

Overall, the students have appreciated the dedication and effort I have put in to encourage and inspire them to delve into difficult concepts on concurrency and parallelism. Following are some of the feedback.

2012-13 semester 2 feedback for CS3211

"Very very sincere and responsible. Very student focused approach which makes one develop an instant liking for the Professor and the subject. Explains in an amazing manner, humorous and very very good. On the whole, great job!"

"Serious overdosage of awesomeness inside him :D Ready to answer queries, however insignificant they might be to my class mates. Holds Saturday sessions for our sake (how cool is a Professor who is ready to spend time away from his family for our sake?). Swift e-mail replies. Understanding. Knowledgeible. You, sir, are a legend!"

2009-10 semester 2 feedback for CS3211

"Very nice person who is willing to go beyong to pass on knowledge to students. I will always remember the day when his father passed away but still came to lecture with a stable mind and a strong heart to complete the lecture and only releasing the news of his father at the end. I trully appreciate this dedicated nature of this lecturer. Not only that but he is always willing to answer students quesions with a smile and an open-mind. He welcomes student feedback every lecture and doesn't mind going over some sections of the lecture just in order to make students understand. I wish I could've performed better in his term test give the fact that he is such a dedicated and enthusiastic lecturer."

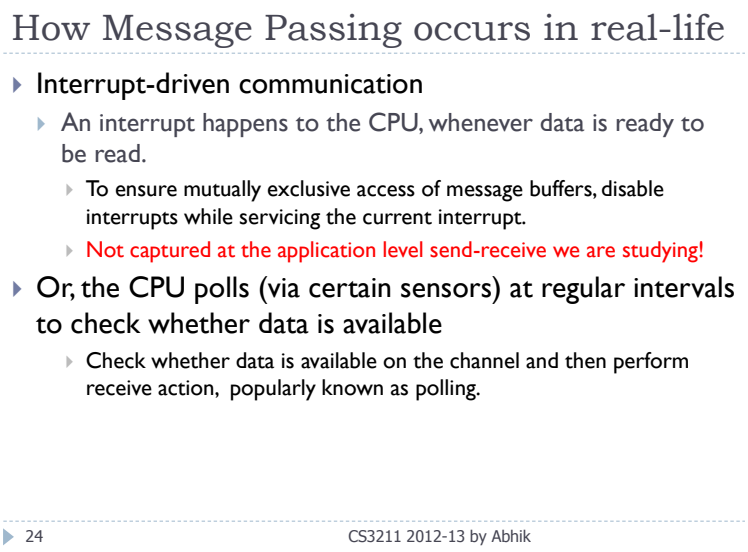
"One of the best professors I have had the opportunity to learn from. Is very knowledgeable in his area of expertise. Always positive and encouraging.. has a great sense of humour. Is very dedicated and sincere. Can spot and encourage the better students as well as help weaker students. Can be firm as well as caring. REally liked the module"

Teaching strategy and innovative practices

In the following, I highlight some teaching practices I have adopted to improve the students' learning experience in CS3211.

Additional Reflection Sessions on Saturdays Over the years, I have taken several steps to improve on my teaching both in terms of the content, delivery, exposition and inspiration. In the previous semester, I was teaching a third year undergraduate module - Parallel and Concurrent Programming. This is the students' first serious foray into concurrency – they get an initial understanding earlier in a module on Operating Systems. As a result, I felt the need for reflection sessions – beyond the lecture and tutorials. Thus, the lectures were used to introduce the teaching materials, the tutorials were conducted as problem solving sessions, and the additional reflection sessions were conducted as open-ended discussion sessions to clarify the students' doubts on concurrency and parallelism. Note that the reflection sessions were very different from a group consultation session – where only the students ask questions and the

instructor answers them, possibly by sharing the answer with all the students present. Instead, the reflection sessions served two purposes. First of all, exercises were given out impromptu during the reflection sessions (the students did not know these exercises in advance). From the working-out of these exercises during the session, the students were reminded of important concepts. Secondly, the students were encouraged and convinced to go beyond stereotypical understanding of concepts. In the context of concurrency and parallelism, a typical example of such stereotypical understanding would be --- the lack of understanding of how message passing happens in real-life, and the inability to separate out the message passing abstraction from its implementation.



As an illustration of how such stereotypical understanding may be stamped out, we would delve into how the message passing abstraction may be implemented. They also realize that implementing the message passing abstraction may also involve managing shared variables – such as the shared message buffers. This separation between the abstraction and the implementation (and acquiring an understanding of both), brings a level of comfort in the students' minds about the topics – as opposed to understanding only the programming abstraction.

Gaining feedback in each lecture via post-it notes as an additional measure I have put in measures to gain more and more technical feedback on which topics the students may find difficult. This is not only done for my modules, but for every single lecture. Prior to every single lecture, I go around and place one post-it note in each of the student's seat. This is done before the lecture starts, so that none of the lecture time is wasted. Subsequently, the students can put in any questions they may have in the post-it note and simply leave it at my desk, or at the door of the lecture theater.

It should be clarified that this measure is not put in to address any lack of interactivity in my lectures. As is evident in many of my student feedback over the semester, the students find my lectures to be very interactive with lot of discussions. However, at the end of the lecture, sometimes there are students who line up to ask additional questions, and sometimes when I am answering the questions for one student – the other students may have to wait which they may not prefer. For this reason, the post-it notes are provided. At the end of every lecture – I collect the post-it notes and send out the answers to the questions in the post-it notes to the whole class via email. This also gives the students an additional avenue for asking questions, which are answered immediately after the lecture.

Appendices in CS 3211 module folder:

1. *Lesson Plan*
2. *Sample Lecture notes*
3. *Sample Tutorial, with solutions*
4. *Sample Lab*
5. *Sample midterm examination, with solutions*
6. *Sample final examination, with solutions*

Module

Description
Text & Readings
Class Roster
Guest Roster
Groups
Timetable
Announcement
Discussion Forum
Lesson Plan
Survey
Workbin
Library E-Reserves

Lesson Plan : PARALLEL AND CONCURRENT PROGRAMMING

Updated: 28-Dec-2012 **Display All Weeks**

Print : Print All Weeks

Search

Weeks		
1 14 Jan-18 Jan	**Introductory Lecture- Concurrency:** We start with concurrency and concurrent execution as a general concept. Notions of threads/processes, interleaving and thread communication are introduced. These concepts are illustrated via a concurrent modeling language. Apart from the readings given, lot of online material on SPIN is available at http://spinroot.com/spin/Man/index.html if you are interested. I have also provided the installation instructions for SPIN - see the file **install.pdf** You can try out the tool if interested. We are mostly using the modeling features of the tool to illustrate concurrency concepts. **Since the concepts covered in this week's lecture are quite important, an additional session will be held on Sat 10 am - 12 noon at Meeting Room 1 (MR1) at COM1-03-19.**	Library E-Reserves Abhik Roychoudhury's Reading List, 2012/2013 Sem 2, Jan an overview of promela pt. 2.pdf - Holzmann, Gerald J. (2004) The spin... an overview of promela pt. 1.pdf - Holzmann, Gerald J. (2004) The spin... Workbins install.pdf - Installation instructions for SPIN.... Lec1.ppt Week1-Saturdaysession.ppt - Here are (some of) the discussions ... extra-lec1-dataraces.ppt - Explaining data races - quick expos...
2 21 Jan-25 Jan	**Multi-threaded Programming constructs in Java:** A blending of concurrency concepts and multi-threaded programming practice is the goal of this lecture. In particular, multi-threaded programming constructs using Java will be introduced. http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html	Workbins Lec2-followup-questions.ppt Lec2.ppt - Java's concurrency - re-uploaded af... extra-lec2-volatiles.ppt
3 28 Jan-01 Feb	**Java Threads & Concurrent Execution:** We continue with Java threads and their concurrent execution in this lecture. There will be less focus on shared objects and ensuring of mutual exclusion in this lecture. **Assignment 1 is posted (see the file Assignment1.pdf) - it is due on Feb 18 at 9:59 PM.** **Submissions should be uploaded to IVLE Workbin folder Lab1.**	Text & Readings Concurrency State Models and Java Programming, Chapters 2,3 Workbins 3211W3Tut-noans.docx - Concurrency modeling [No answer... 3211W3Tut-withans.docx - Week 3 tutorial - with answers Assignment1.pdf - First assignment posted - due 18 Fe... Lec3-followup.ppt - After lecture 3 (follow-up question... Lec3.ppt - Here are the finalized slides for t... Warmup-at-beginning-of-lec3.ppt - At the start of Lecture 3 (warm-up ...
4 04 Feb-08 Feb	**Lecture Cancelled, Tutorials continue as usual.**	Workbins w4-tutorial.zip - Week 4 tutorial (after a fix for in...
5 11 Feb-15 Feb	**Shared objects and Mutual Exclusion:** We will finish the discussion on concurrent execution (Lec4-2.ppt) and discuss examples of multi-threaded programming with shared objects using locks (Lec5.ppt). **Monday is a Public Holiday for CNY, so tutorials are affected.** **A make-up tutorial will be held on Saturday 16 Feb 10 am at Executive Classroom COM2-04-02.** ***The make-up will be followed by an optional additional session for all of us to discuss the concepts covered so far.***	Workbins 3211W5TutNoans.docx - Week 5 tutorial 3211W5TutWithans.docx - Week 5 tutorial with answers ThreadDemo.java - Java file for example program ThreadPanel.java - Java file for example program Lec5.ppt week5-saturdaysession.ppt - Saturday extra session on Feb 16. Lec5-followup.txt
6 18 Feb-22 Feb	**Monitors and condition synchronization:** Having discussed locks, we will discuss monitors and condition synchronization in this lecture. **Assignment 1 is due.** **Assignment 2 is posted.**	Workbins 3211W6TutNoans.docx - Week 6 tutorial, based on the lectu... CS3211-Tut-Week6.docx - Week 6 tutorial with answers cs3211-lab2.pdf - Assignment 2 description Lec6-followup.ppt Lec6.ppt handout.zip - Assignment 2 handout, containing co...
Recess 23 Feb-03 Mar		
7 04 Mar-08 Mar	**Midterm Examination** (*first hour*) **Deadlocks** in Concurrent Programming (second hour) ***Solution for Lab1 has been posted.***	Workbins CS3211-Tut-Week7-noans.doc CS3211-Tut-Week7-withans.doc SolHW1-new.pdf - Sample solution for Lab 1 is being ... Lec7.ppt - Deadlocks
8 11 Mar-15 Mar	**Safety and Liveness properties in Concurrent Programming** : So far we have studied specific properties to think of while writing concurrent programs - such as mutual exclusion and no deadlocks. In this lecture, we put them in proper perspective by presenting two general classes of properties, namely safety and progress.	Workbins CS3211-Tut-Week8-noans.doc - Week 8 tutorial CS3211-Tut-Week8.doc Lec8.ppt - Safety and Liveness Lec8followup.ppt - Advanced optional material on safet... lec8-somequestions-asked.txt
9 18 Mar-22 Mar		Workbins CS3211-Tut-Week9-noans.doc - Week 9 tutorial CS3211-Tut-Week9.doc

Message Passing: In this lecture, we discuss how to implement synchronous and asychnronous message passing in Java. This also provides a gentle glide for all of us from concurrent programming to parallel programming, since we will be learning parallel programming using the Message Passing Interface (MPI) specification from the next lecture.

Execution models of Parallel Programming: In the second hour of the lecture, we will discuss the basic concepts of parallel programming - in particular the execution models. This will be useful to understand before starting on hands-on parallel programming using MPI.

There will be an optional additional session for all of us to discuss the concepts on concurrent programming. This will be on Saturday March 23rd at 10 am - 12 noon in Executive Classroom (COM2 04-02).

- Sol-mid-2013.doc - Sample solutions for midterm - disc...
- Concurrency questions.docx
- Lec9-part1.pptx - Message passing using Java

Feed

10
25 Mar-29 Mar

Basics of Parallel Programming (30 - 40 minutes)
[continued from the end of previous lecture]

Basics of MPI, discussion of blocking point-to-point communication (remaining time)

Assignment 2 is due.

[This week onwards we discuss MPI programming. You will program using MPI on the tembusu cluster.
Please refer to MPI programming instructions on tembusu - see *tembusu-MPI-access.docx*]

- Library E-Reserves
  - Abhik Roychoudhury's Reading List, 2012/2013 Sem 2, Jan
    - MPI & Other Local View Languages pt.2.pdf - Lin, Calvin (2009) Principles of
    - MPI & Other Local View Languages pt.1.pdf - Lin, Calvin (2009) Principles of
- Text & Readings
  - Principles of Parallel Programming
- Workbins
  - CS3211-Tut-Week10-noans.doc - Week 10 tutorial
  - CS3211-Tut-Week10.doc
  - tembusu-MPI-access.docx - MPI access instructions on tembusu ...
  - Lec10-MPI-1.ppt
  - Lec10-part1-Parallelism-basics.pptx

Feed

11
01 Apr-05 Apr

MPI Communication styles : Non-blocking point to point communication, Collective communication in parallel programming

Assignment 3 is posted - see uploaded file *cs3211-lab3.pdf*

READING: A copy of the chapter on MPI from Lin and Snyder's book will be placed on E-reserves (1 download only per student). In addition, you can look at the following online tutorial

https://computing.llnl.gov/tutorials/mpi/

- Library E-Reserves
  - Abhik Roychoudhury's Reading List, 2012/2013 Sem 2, Jan
    - MPI & Other Local View Languages pt.2.pdf - Lin, Calvin (2009) Principles of
    - MPI & Other Local View Languages pt.1.pdf - Lin, Calvin (2009) Principles of
- Text & Readings
  - Principles of Parallel Programming
- Workbins
  - CS3211-Tut-Week11-noans.doc - Week 11 tutorial
  - CS3211-Tut-Week11.doc
  - cs3211-lab3.pdf - Lab 3 - MPI programming
  - Lec10-MPI-1.ppt
  - Lec11-MPI-2.ppt

Feed

12
08 Apr-12 Apr

Communicators in MPI: We conclude the discussion on collective communication in MPI, and finish with the management of communicators/groups in MPI.

READING: A copy of the chapter on MPI from Lin and Snyder's book will be placed on E-reserves (1 download only per student). In addition, you can look at the following online tutorial

https://computing.llnl.gov/tutorials/mpi/

- Library E-Reserves
  - Abhik Roychoudhury's Reading List, 2012/2013 Sem 2, Jan
    - MPI & Other Local View Languages pt.1.pdf - Lin, Calvin (2009) Principles of
    - MPI & Other Local View Languages pt.2.pdf - Lin, Calvin (2009) Principles of
- Text & Readings
  - Principles of Parallel Programming
- Workbins
  - CS3211-Tut-Week12-noans.doc - Week 12 tutorial
  - CS3211-Tut-Week12.doc
  - Lec11-MPI-2.ppt
  - Lec12-MPI-3.ppt - I added a few slides to explain the...

Feed

13
15 Apr-19 Apr

Revision of all materials: This lecture will constitute a revision of all materials. Some examples/questions will be discussed. The notes for this lecture will be posted *after* the lecture.

- Workbins
  - CS3211-Tut-Week13-noans.doc - Week 13 tutorial. Since it has more...
  - CS3211-Tut-Week13.doc
  - Lec13.ppt - Week 13 - revision

Feed

Reading
20 Apr-26 Apr

Assignment 3 is due.

Consolidated zip of all tutorials with solutions - alltut.zip - posted here.

Consolidated zip of all lecture slides all_lec.zip - posted here.

- Workbins
  - all_lec.zip - Consolidated zip of all lectures
  - alltut.zip - Consolidated zip of all tutorials w...

Examination
27 Apr-11 May

Vacation
12 May-04 Aug

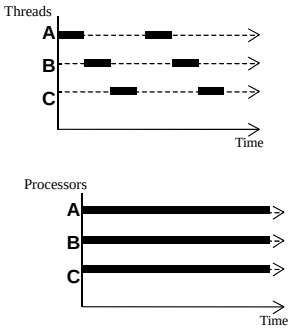
CS 3211 Sample Lecture Notes

Parallel Programming and MPI- Lecture 1

Abhik Roychoudhury
CS 3211
National University of Singapore

Sample material: Parallel Programming by Lin and Snyder, Chapter 7.
Made available via IVLE reading list, accessible from Lesson Plan.

Concurrency and Parallelism



Why parallel programming?

- Performance, performance, performance!
- Increasing advent of multi-core machines!!
  - Homogeneous multi-processing architectures.
  - Discussed further in a later lecture.
- Parallelizing compilers never worked!
  - Automatically extracting parallelism from app. is very hard
- Better for the programmer to indicate which parts of the program to execute in parallel and how.

How to program for parallel machines?

- Use a parallelizing compiler
  - Programmer does nothing, too ambitious !
- Extend a sequential programming language
  - Libraries for creation, termination, synchronization and communication between parallel processes.
  - The base language and its compiler can be used.
  - Message Passing Interface (MPI) is one example.
- Design a parallel programming language
  - Develop a new language – Occam.
    - Or add parallel constructs to a base language – High Perf. Fortran.
  - Must beat programmer resistance, and develop new compilers.

Parallel Programming Models

- Message Passing
  - MPI: Message Passing Interface
  - PVM: Parallel Virtual Machine
  - HPF: High Performance Fortran
- Shared Memory
  - Automatic Parallelization
  - POSIX Threads (Pthreads)
  - OpenMP: Compiler directives

The Message-Passing Model

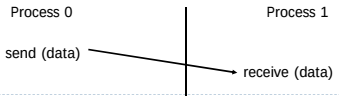
- A process is (traditionally) a program counter and address space
- Processes may have multiple threads (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's

The programming model in MPI

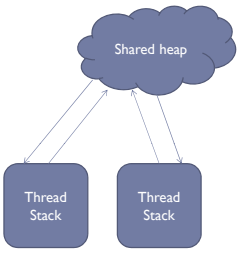
- Communicating Sequential Processes
  - Each process runs in its local address space.
  - Processes exchange data and synchronize by message passing
  - Typically, but not always, the same code may be executed by all processes.

Cooperative Operations for Communication

- Message-passing approach makes the exchange of data cooperative
- Data is explicitly sent by one process and received by another
- Advantage:
  - Any change in the receiving process's memory is made with the receiver's active participation.
- Communication and synchronization are combined.



Shared Memory communication in Java



Java program compiled into bytecodes. Bytecodes are interpreted by the Java Virtual Machine.

Bytecodes are the assembly language of the Java Virtual Machine (a machine implemented in software).

Bytecode execution returns in movements between thread local stack and the shared heap (which is shared across threads).

Program to Bytecode

```
3: public int foo(int j){
4: int ret;
5: if ( j % 2 == 1 )
6: ret= 2;
7: else
8: ret= 5;
9: return ret;
10: }
```

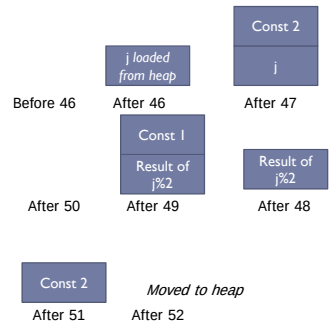
```
public int foo(int);
46: iload_1
47: iconst_2
48: irem
49: iconst_1
50: if_icmpne 54
51: iconst_2
52: istore_2
53: goto 56
54: iconst_5
55: istore_2
56: iload_2
57: ireturn
```

Simplified Bytecode format

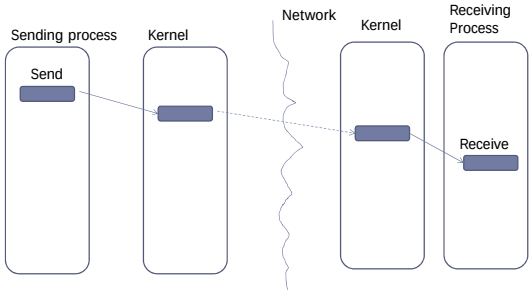
Stack ↔ Heap movements in Java

```
public int foo(int);
46: iload_1
47: iconst_2
48: irem
49: iconst_1
50: if_icmpne 54
51: iconst_2
52: istore_2
53: goto 56
54: iconst_5
55: istore_2
56: iload_2
57: ireturn
```

j%2 == 1
ret = 2

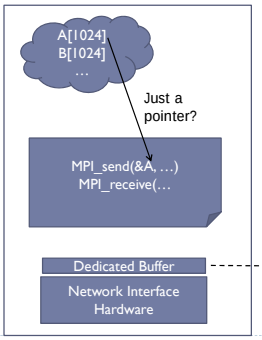


In comparison, communication in MPI is:



No notion of a shared address space across processes.

More elaborate view of a MPI process



Program's memory – stack or heap.

Address space of MPI libaries
May contain buffers.

Data travels over the network

Message Passing Interface (MPI)

- A message-passing library specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Designed to provide access to parallel hardware for
  - End users
  - Library writes
  - Tool developers
- Provides a powerful, efficient, and portable way to express parallel programs

MPI (Contd.)

- The processes in a parallel program are written in a sequential language (e.g., C or Fortran)
- Processes communicate and synchronize by calling functions in MPI library
- Single Program, Multiple Data (SPMD) style
  - Processors execute copies of the same program
  - Each instance determines its identity and takes different actions

MPI History

- Message Passing Interface Forum
  - Representative from over 40 organizations
- Goal
  - Develop a single library that could be implemented efficiently on the variety of multiprocessors
- MPI-1 accepted in 1994
- MPI-2 accepted in 1997
- MPI is a standard
- Several implementations exist

Some Basic Concepts

- Processes can be collected into groups
  - An ordered set of processes.
- A group and context together form a communicator
  - A scoping mechanism to define a group of processes.
  - For example define separate communicators for application level and library level routines.
- A process is identified by its rank in the group associated with a communicator
- There exists a default communicator whose group contains all initial processes, called MPI_COMM_WORLD

MPI Datatypes

- Data in a message is described by a triple
  - <address, count, datatype> where
- MPI datatype is recursively defined as
  - Predefined corresponding to a data type from the language (MPI_INT, MPI_DOUBLE)
  - A contiguous array of MPI datatypes
  - A strided block of datatypes
  - An indexed array of blocks of datatypes
  - An arbitrary structure of datatypes
- MPI functions can be used to construct custom datatypes

Why datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication)
- Specifying application-oriented layout of data in memory
  - Reduces memory-to-memory copies in the implementation
  - Allows the use of special hardware (scatter/gather) when available

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a tag or not screened by specifying MPI_ANY_TAG as the tag in a receive

Basic MPI Functions

- MPI_Init(int *argc, char ***argv)
  - Initializes MPI
  - Must be called before any other MPI functions
- MPI_Comm_rank(MPI_Comm comm,
  int *rank)
  - Find my rank within specified communicator
- MPI_Comm_size (MPI_Comm comm,
  int *size)
  - Find number of group members within specified communicator
- MPI_Finalize ()
  - Called at the end to clean up

Getting started

```
#include "mpi.h"
#include <stdio.h>
int main( argc, argv )
int argc;
char **argv; {
   MPI_Init( &argc, &argv );
   printf( "Hello world\n" );   /* run on each process */
   MPI_Finalize();
   return 0;
}
```


MPI_Comm_size and MPI_comm_rank

- Two of the first questions asked in a parallel program are:
  - How many processes are there? and
  - Who am I?
- How many is answered with
  - MPI_Comm_size
- Who am I is answered with
  - MPI_Comm_rank.
  - The rank is a number between zero and size-1.

What does this program do?

```
#include "mpi.h"
#include <stdio.h>

int main( argc, argv )
int argc;
char **argv; {
int rank, size;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   printf( "Hello world! I'm %d of %d\n", rank, size );
   MPI_Finalize();
   return 0;
}
```

Embarrassingly simple MPI program

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, id, p;
    void unit_task( int, int); // no return value

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    for (i=id; i < 65536; i+=p) unit_task(id, i);
    printf("Process %d is done\n", id);
    fflush(stdout); MPI_Finalize();
    return 0;
}
```

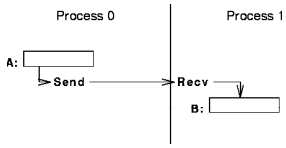
Compile: mpicc –o simple simple.c

Run: mpirun –np 2 simple (creating 2 processes)

Organization

- So Far
  - What is MPI
  - Entering and Exiting MPI
  - Creating multiple processes
- Now
  - Message Passing

Inter-process comunication



- Via point-to-point message passing.
- Messages are stored in message buffers.

Basic Blocking Communication

- int MPI_Send (void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - Send contents of a variable (single or array) to specified PE within specified communicator
- When this function returns, the data has been delivered and the buffer can be reused. The message may not have been received by the target process
- [Blocking here means]
  - Sender blocks until the send action is completed, not recv.
  - Receiver blocks until the recv. is completed.

More on blocking send

- int MPI_Send (void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - The address of data to be sent
  - # of data elements to be sent
  - Type of data elements to be sent
  - ID of processes that should receive the message
  - A message tag to distinguish the message from other messages which may be sent to the same process.
    - Wild cards allowed, we can say MPI_ANY_TAG
  - A communication context capturing groups of processes working on the same sub-problem
    - By default MPI_COMM_WORLD captures the group of all processes.

Basic Blocking Communication (contd.)

- int MPI_Recv(void *buff, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - Receive contents of a variable (single or array) from specified PE within specified communicator
- Waits until a matching (on source and tag) message is received
- Source is rank in communicator specified by comm or MPI_ANY_SOURCE
- Receiving fewer than count occurrences of datatype is OK, but receiving more is an error
- The status field captures information about
  - Source , Tag, How many elements were actually received

Simple Sample Program

```
#include <mpi.h>
main( int argc, char *argv[]) {
   ……..
   MPI_Init (&argc, &argv);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   if (myid == 0)
     { otherid = 1; myvalue = 14;}
   else
     { otherid = 0; myvalue = 25;}
   MPI_Send (&myvalue, 1,MPI_INT,otherid, 1, tag, MPI_COMM_WORLD);
   MPI_Recv (&othervalue, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
     MPI_COMM_WORLD, &status);
   printf(" process %d received %d\n", myid, othervalue);
   MPI_Finalize();
}
```


Another example

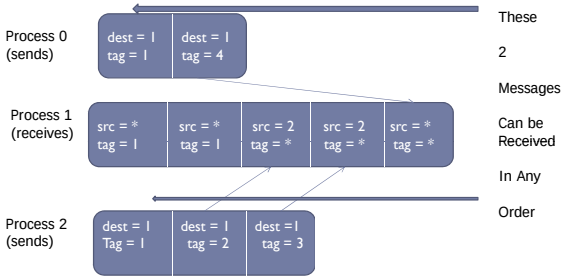
```
char msg[20];  int myrank, tag =99;
MPI_status status;
…
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0){
   strcpy(msg, "Hello there");
   MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else if (myrank == 1){
   MPI_Recv(msg, 20, MPI_CHAR,0, tag, MPI_COMM_WORLD, status);
}

…
```

status tells us how many elements were actually received!

Message ordering

- MPI_Send and MPI_Recv are blocking
  - MPI_Send blocks until send buffer can be reclaimed.
  - MPI_Recv blocks until receive is completed.
  - When MPI_Send returns we cannot guarantee that the receive has even started.
- If the sender sends 2 messages to same destination which match the same receive, the receive cannot match the 2nd msg, if the 1st msg is still pending.
- If a receiver posts 2 receives, and both match the same msg, the 2nd receive cannot get the msg, if the 1st receive is still pending.

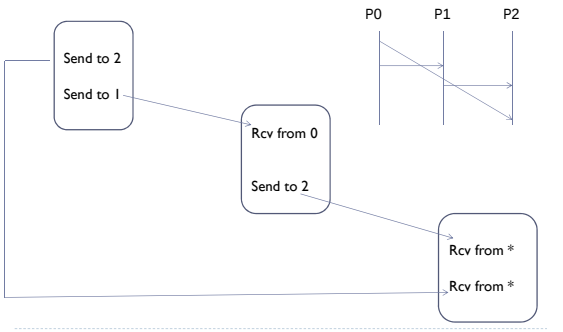
Order preservation in messages



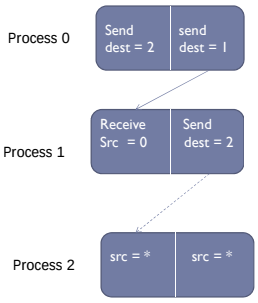
Order preservation in messages

- Messages are non-overtaking
  - Successive messages sent by a process p to another process q are ordered in sequence.
- Receives posted by a process are also ordered.
  - Each incoming message matches the first matching receive.
  - Matching defined by tags and source/destination.

Order preservation is not transitive



Order preservation is not transitive



Between any pair of processes, messages flow in order.
However, across pairs of processes we cannot guarantee a consistent total order on the comm. events.

Communication delays can be arbitrary.

Wrapping up

- Blocking sends and receives
  - A blocking send completes when the send buffer can be re-used
  - A blocking receive completes, when the data is available in the receive buffer.
  - Each incoming message matches the first matching receive.
  - Order is preserved between any pair of processes.
  - Order preservation is however, not transitive.

Organization

- So Far
  - What is MPI
  - Entering and Exiting MPI
  - Creating multiple processes
  - Blocking Message Passing (point-to-point)
- Now
  - Non-blocking point to point communication
  - Collective communication

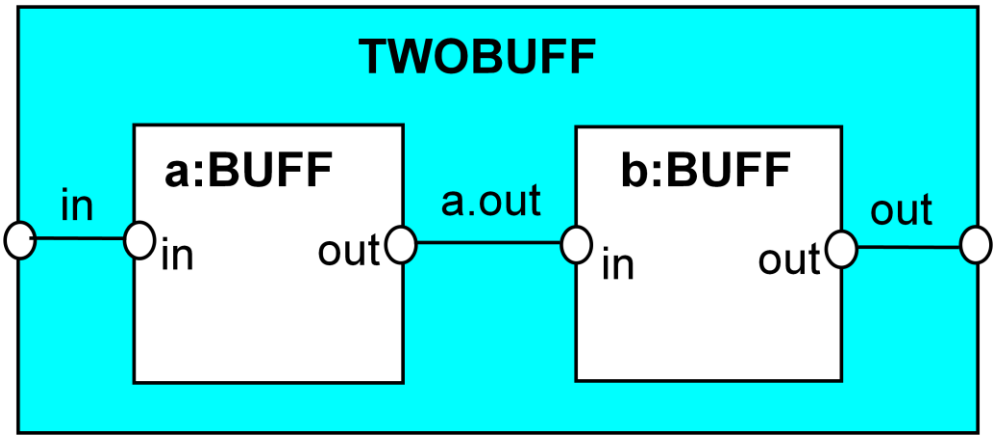
CS 3211 Sample Tutorial questions (with solutions)

Content: Lecture 5 (shared objects and mutual exclusion)

Sample Exercises:

[Please conduct these as an interactive discussion, rather than an evaluation. Please also make it clear to the students that they are not being evaluated for their performance in these exercises, so that they are not afraid to make mistakes while answering.]

1. Consider a one cell buffer
   BUFF = in(x) -> out(x) -> BUFF



Suppose we want to put the two cells together in the above fashion. What will be the process equation describing the TWOBUFF process.

Answer:

TWOBUFF = (a:BUFF/{in/a.in} || b:BUFF /{a.out/b.in, out/b.out})@{in,out}

Another solution is

TWOBUFF = (BUFF/{x/out} || BUFF/{x/in})@{in,out}

2. Consider the following process equations from lecture slides

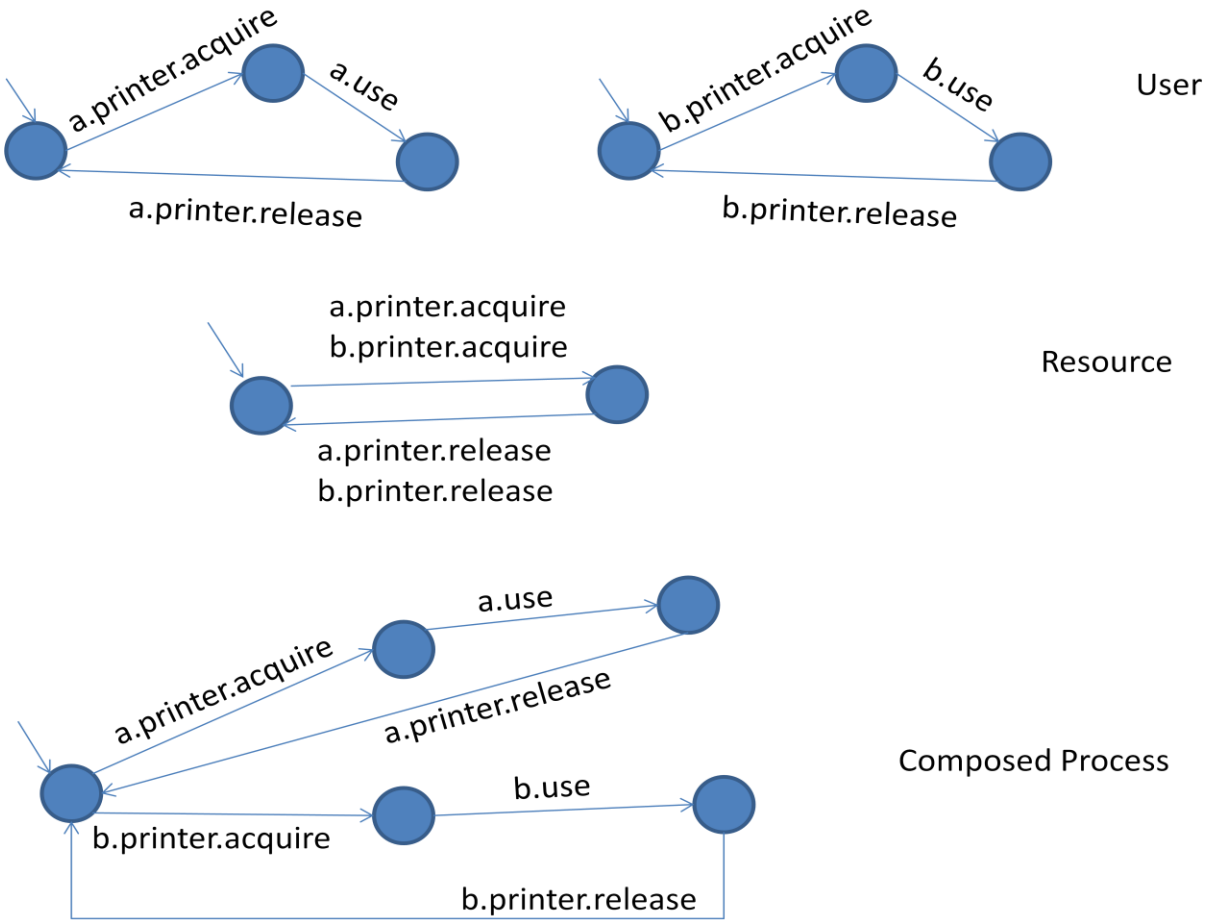
RESOURCE = acquire -> release -> RESOURCE.

USER = printer.acquire -> use -> printer.release -> USER.

PRINTER_SHARE = (a:USER || b:USER || {a,b}:PRINTER:RESOURCE).

Describe all behaviors of the composed process.

Answer: The state model compositions are shown below.



3. In class we discussed two methods of lock – to embed the lock inside the object, or to make every "user" of a shared object impose a locking discipline.

Choice 1 :

```
class SynchronizedCounter extends Counter {
…
   synchronized void increment() {
      Counter.increment();
   }
}
```

Choice 2: `synchronized(counter) {counter.increment();}`

Comment on the use of the choices in presence of the recursive locking scheme of Java.

Answer: Choice 2 is less safe, since it requires all user threads of shared objects to remember to impose a locking discipline. However, it is less dependent on the recursive locking scheme since the top level method call is locked. So, even if the increment method is recursive, the code from choice 2 can execute without running into a deadlock.

4. Let x be a shared integer variable. Consider two threads executing the following code

```
while (x< 1){                    while (x >= 1){
        x++;                             x--;
}                                }
```

Initially x is 1. What can you say about the termination of the left hand side loop? What are the possible number of times it can execute? Assume *each line of code* is executed atomically.

Also, comment on the possible termination / non-termination of the program as a whole.

Answer: Executing 0 times is possible, if the left thread executes at the very beginning and the loop is exited.

Executing once is also possible, if the right thread executes one iteration of its loop, exits the loop, and the then the left thread executes as follows

```
                                while(x >= 1)    // x is now 1
                                      x--   // x is now 0
                               while (x >= 1)   // exits
while (x < 1)  // x is now 0
      x++   // x is now 1
while (x < 1) // exits
```

So, in this scenario both loops terminate.

We could construct a scenario where both loops go on forever by alternating one iteration of each loop.

```
                                while  (x >= 1)  // x is now 1
                                      x--    // x is now 0
while (x < 1)  // x is now 0
     x++  // x is now 1
                                while (x >= 1) // x is now 1
                                      x--  //  x is now 0
while (x < 1) // x is now 0
      x++  // x is now 1
…….
```

5. Consider the following multi-threaded program. flag is a boolean variable initialized to false. x is an integer variable initialized to zero. Assume that all condition evaluations and assignments are executed atomically.

```
Thread 1            |  Thread 2
-------------------------------------------
while (!flag){      |  while (x == 0){
   x = 1 - x;}      |           x = x; }
                    |   flag = true;
```

(i) Construct an interleaving where the program does not terminate.
(ii) Construct an interleaving where the program terminates.
(iii) What are the possible values of x when the program terminates? Justify your answer.

(i) If the first thread is not scheduled at all, only the second thread is scheduled and the program does not terminate.

(ii) The following interleaving leads to termination.

```
while (!flag){
x = 1 - x // x == 1
                                  while (x == 0) // exits loop
                                  flag = true
while (!flag) // exits loop
```

(iii) In the above interleaving we have x == 1 when the program terminates.

We can also have x == 0 as follows.

```
while (!flag){
x = 1 - x // x == 1
                                   while (x == 0) // exits loop
while (!flag){
x = 1 - x // x == 0
                                   flag = true;
while (!flag) // exits loop
```

CS 3211 Sample Lab Assignment

Lab 3 of CS 3211, 2013, Total 10 marks

Please submit in IVLE workbin folder **Lab3** by **Monday 22 April, 9:59 PM.**

Kindly note that there will be no extensions. If you are not finished by the deadline, please submit whatever partial answer you may have - this is better than not submitting at all. Only submissions in the IVLE Workbin will be graded. Submissions sent by e-mail, unfortunately, cannot be considered.

Upload one single zip file containing all the files including the programs (.c files). Also include a README.txt file in the zip which will say what each file contains.

*Please submit your zip file to the IVLE Workbin folder **Lab3***

MPI usage instructions

Beginning with MPI: *refer to the file* ***tembusu-MPI-access.docx*** *in*

Workbin\Assignment Descriptions.

Please only use access node 0- 4, do not use any other access node or computing node.

Let your program be cpi.c

MPI program running over Ethernet (MPICH)
[user@access0]$ /opt/mpich/bin/mpicc -c cpi.c
[user@access0]$ /opt/mpich/bin/mpicc -o cpi cpi.o

For MPICH, create a machine file that looks like this. Call it "mynodes" for example

access0
access1
access2
access3
access4

Run binary MPI program (MPICH), for example by executing …
[user@access0]$ /opt/mpich/bin/mpirun -machinefile mynodes -np 4 ./cpi

Question 1 [6 marks]

A frequently used operation is parallel computing is prefix sum. Given a sequence of numbers x0, x1, …, xn, prefix sum computes all the partial sums as follows:

s0 = x0
s1 = x0 + x1
s2 = x0 + x1 + x2
: : :
sn = x0 + x1 + … + xn

Write an MPI program to compute parallel prefix sum. Use the algorithm described in Figure 2 of the paper Data Parallel Algorithms" by Hills and Steele, see

http://cva.stanford.edu/classes/cs99s/papers/hillis-steele-data-parallel-algorithms.pdf

For simplicity, you can assume the length of sequence and number of processes (specified by –np) are the same. Submit the C source code to compute prefix sums, as well as sample output.

Question 2 [4 marks]

Consider a collection of processes, where each process has an array of 10 integers. For each of the 10 locations, compute the smallest value, and rank of the process containing the smallest value. Submit the C source code as well as the sample program output.

CS 3211 Sample Midterm Examination

NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

EXAMINATION FOR
Semester 2 AY2012/2013

CS3211 – PARALLEL AND CONCURRENT PROGRAMMING

March 2013 **Time Allowed: 1 Hour**

INSTRUCTIONS TO CANDIDATES

1. This examination contains **four (4)** questions and comprises **SEVEN (7)** printed pages, including this page.
2. Answer **ALL** questions within the space in this booklet
3. This is an Open Book examination.
4. Please write your Matriculation Number below.

MATRICULATION NO: _____

This portion is for examiner's use only

Question	Marks	Remarks
1	**/ 6**	
2	**/ 4**	
3	**/ 5**	
4	**/ 5**	
Total	**/ 20**	

Question 1 [6 marks]

Consider the following multi-threaded program where x is an integer variable initialized to zero. You may assume that any assignment statement is executed atomically. Any condition evaluation is also executed atomically. The printf statement is also executed atomically.

```
Thread 1                    |          Thread 2
-------------------------------------------------------------
                            |
while (x < 2){              |        x = x + 1;
   printf("%d", x);}        |        x = x + 1;
                            |
```

Which of the following output sequences may be printed? For each, if it may be printed by the program, construct an interleaving that can print it. Also, if any sequence cannot be printed – give a reason why it cannot be printed.
(i) 012 (ii) 021 (iii) 12

Answer:

(i) This is possible, as shown by the following interleaving

```
while (x < 2)
     print x   // print 0
                                    x = x + 1     // x== 1
while (x <2)
     print x // print 1
while (x < 2)
                                    x = x + 1 //  x == 2
     print x // print 2
```

(ii) Not possible since the value of x is monotonically increasing with any execution of this program. Thus, prints in the left hand thread of the program should be printing higher values in later iterations of the loop (as compared to the earlier iterations).

(iii) This is possible, as shown by the following interleaving.

```
                                    x = x + 1 //   x == 1
while (x < 2)
     print  x // print 1
while (x < 2)
                                    x = x + 1 // x == 2
     print x // print 2
```

Question 2 [4 marks]

Consider the following encoding in Promela for the critical section problem. Processes are trying to access critical section, and we should ensure mutual exclusion of access, no deadlock, and eventual entry to critical section for each process. Comment on the following solution. You may assume that a false statement always blocks.

```
byte turn = 1;
active proctype P(){
    do
    ::  if
        ::  true
        ::  true -> false
        fi
        turn == 1;
       // critical section
        turn = 2;
    od
}
```

```
active proctype Q(){
      do
       ::  turn == 2;
           // critical section
           turn = 1;
       od
 }
```

Answer:

The only challenge comes from the following structure

```
if
 :: true
 :: true -> false
fi
```

Otherwise – it is simply a round-robin scheme which satisfies all the three properties.

Due to this if structure – the process P may block.
This will prevent process Q's attempt to enter critical section, since it waits forever for turn == 2 to be true.

Question 3 [5 marks]

Consider an atomic operation flip, such that

```
int flip (int lock){ lock =(lock +1)%3; return lock}
```

This is a variation of an example we discussed in class, where we had `lock =(lock +1)%2`. Suppose 2 processes are executing the following code, with lock initialized to 0. Will the solution work – i.e. it ensures mutual exclusion and no starvation? Give detailed comments.

```
/* Lock acquisition */
while (flip(lock) != 1)
      while (lock!= 0) {};
CRITICAL SECTION  /* Does not alter the value of lock */
/* Lock release */
lock = 0;
```

Answer:

For the class example, mutual exclusion was violated with `lock =(lock +1)%2`

Mutual exclusion is now preserved with `lock =(lock +1)%3`. Initially lock is 0, and any arbitrary process, say process 1, executes flip(lock) which returns 1, gaining entry to critical section. Since subsequent flip(lock) executions return 2, process 2 enters the inner loop where it is stuck until process 1 exits from the critical section and sets lock to 0.

No starvation is not guaranteed as shown by the following execution

Process 1	Process 2

flip(lock) returns 1 // exit outer loop
CRITICAL SECTION	

	flip(lock) returns 2 // enter outer loop
	lock == 2 // enter inner loop and stuck

lock = 0	
	lock == 0 // exit inner loop

flip(lock) returns 1 // exit outer loop
CRITICAL SECTION	

	flip(lock) returns 2// enter outer loop

< The pattern above may repeat forever>

EMPTY PAGE

Question 4 [5 marks]

The readers-writers problem for concurrently accessing a shared database was discussed in class. In this problem, several reader and writer threads try to access a shared database. At any time only one writer or several readers (but not both) should be allowed to access the database.

A. Following is a Java solution of the problem using monitors. Comment on the solution in terms of progress of readers/writers in eventually accessing the database. Give detailed comments.
B. Comment in general about the ability of monitors in Java in ensuring that a thread trying to enter a monitor will eventually (and quickly) enter the monitor.

```
class RWmonitor{
   private int readers = 0; private boolean writing = false;

  public synchronized void StartRead(){
           while (writing){
              try{ wait();
                 } catch(InterruptedException e){}
           }
           readers++; notifyAll();
  }

  public synchronized void StartWrite(){
           while (writing || (readers != 0)){
                try{ wait();
                   } catch(InterruptedException e){}
           }
           writing = true;
   }
}
```

```
public synchronized void EndRead(){
        notifyAll();
        readers--;
}

public synchronized void EndWrite(){
         notifyAll();
         writing = false;
}
```

Answer

A. A writer will wait whenever *readers > 0*. Hence readers can starve out a writer if more and more readers continue to acquire access to the database by executing StartRead. Even if there are fixed number of readers, and they are forced to exit reading after bounded time --- we can have reader i+1 acquire access to the database immediately after reader i relinquishes access. This can go on forever, starving the writer.

   The readers also wait whenever *writing* == *true*. Thus, if there are several hungry writers, they will also continue to access the database, starving out the readers.

B. In Java, the process executing notify (the signaling process) has to release the lock. Even after executing notify/notifyAll --- it continues to hold the lock until it returns from a synchronized method or encounters a wait itself. The notified process (which was waiting) therefore has to re-check the condition on which it was waiting, and the condition may no longer be true. This allows for starvation in monitor entry.

To avoid such starvation – one could allow for the signaling process to immediately pass control to the chosen waiting process. However, this is not done in Java implementations.

In addition, notifyAll notifies all waiting processes – processes waiting on the object, and only one of them is chosen. So, a process may keep on getting ignored.

END OF PAPER

CS 3211 Sample Final Examination

NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

EXAMINATION FOR
Semester 2 AY2012/2013

CS3211 – PARALLEL AND CONCURRENT PROGRAMMING

May 2013 **Time Allowed: 2 Hours**

INSTRUCTIONS TO CANDIDATES

1. This examination paper contains **five (5)** questions and comprises **TEN (10)** printed pages, including this page.
2. Answer **ALL** questions within the space in this booklet
3. This is an Open Book examination.
4. Please write your Matriculation Number below.

MATRICULATION NO: _____

This portion is for examiner's use only

Question	Marks	Remarks
1	**/ 10**	
2	**/ 10**	
3	**/ 5**	
4	**/ 10**	
5	**/ 10**	
Total	**/ 45**	

Question 1 [10 marks]

Consider an array X storing the names of people working in A*STAR, another array Y storing the names of alumni of NUS School of Computing, and another array Z of people living in the Clementi area of Singapore. All three arrays are sorted in alphabetical order. There exists at least one person whose name exists in all the three arrays (there may be more).

a) Write a Promela process to find out the alphabetically first such person, whose name appears in all the three arrays.
b) Develop a concurrent solution in Promela with several processes co-operating to solve the problem.

Answer:

a) The solution can be described concisely as single Promela do-loop with non-determinism

```
i = 0; j = 0;   k = 0;
do
::  X[i]  < Y[j]   -> i  = i + 1;
::  Y[j]  < Z[k]  ->  j = j + 1;
::  Z[k]  < X[i]  ->  k = k +1;
:: else -> break;
od
printf(i, j, k);
```

b) We should have three plus one, that is four threads. The schematic pseudo-code is as follows

```
flag1 = flag2 = flag3 = 1;

do                              do                              do
  :: X[i] <  Y[j] -> i = i++;     :: Y[j] < Z[k] -> j++;          :: Z[k] < X[i] -> k++;
  :: else -> flag1 = 0;           :: else ->flag2 = 0;            :: else -> flag3 = 0
od                              od                              od


do
::  flag1 + flag2 + flag3 == 0 -> print(i,j,k); break;
::  else -> print("not yet\n");
od
```

-BLANK PAGE-

Question 2 [10 marks]

Suppose there are n passenger threads and a car thread. The passengers repeatedly wait to take roller coaster rides in the car, which can hold C passengers, where C < n. The car can go round the tracks only when it is full. Moreover:

- Passengers should invoke `board` and `unboard`.
- The car should invoke `load`, `go_around` and `unload`.
- Passengers cannot `board` until the car has invoked `load`.
- The car cannot depart until C passengers have boarded (also stated in the text above).
- Passengers cannot `unboard` until the car has invoked `unload`.

Write a multi-threaded Java program for the passengers and car that enforces these constraints.

Answer:

As discussed in class, semaphores can be encoded as Java monitors. Below is the initialization code.

```
mutex = Semaphore(1)
mutex2 = Semaphore(1)
boarders = 0
unboarders = 0
boardQueue = Semaphore(0)
unboardQueue = Semaphore(0)
allAboard = Semaphore(0)
allAshore = Semaphore(0)
```

`mutex` protects passengers, which counts the number of passengers that have invoked `boardCar`.
Passengers wait on `boardQueue` before boarding and `unboardQueue` before unboarding.
`allAboard` indicates that the car is full.

Following is the code for the car thread.

```
load()
boardQueue.signal(C)
allAboard.wait()
go_around()
unload()
unboardQueue.signal(C)
allAshore.wait()
```

When the car arrives, it signals C passengers, then waits for the last one to signal allAboard. After it departs, it allows C passengers to disembark, then waits for allAshore.

The code for the passenger thread is given in the next page.

```
boardQueue.wait()
board()

mutex.wait()
     boarders += 1
     if boarders == C{
            allAboard.signal();
            boarders = 0;
      }
mutex.signal()       // this portion is protected by mutex

unboardQueue.wait()
unboard()

mutex2.wait()
       unboarders += 1;
       if unboarders == C{
               allAshore.signal();
               unboarders = 0;
       }
mutex2.signal();    // this portion is protected by mutex2
```

Passengers wait for the roller coaster car before boarding, naturally, and wait for the car to stop before leaving. The last passenger to board signals the car and resets the passenger counter.

Note that mutex and mutex2 could have been defined as separate lock objects – allowing for fine grained locking. The passenger code could have been as follows.

```
private Object mutex = new Object();
private Object mutex2 = new Object();
boardQueue.wait()
board()

synchronized(mutex){
     boarders += 1
     if boarders == C{
            allAboard.signal();
            boarders = 0;
      }
}
unboardQueue.wait()
unboard()
synchronized(mutex2){
       unboarders += 1;
       if unboarders == C{
               allAshore.signal();
               unboarders = 0;
       }
}
```

BLANK PAGE

Question 3 [5 marks]

Following is the code executed by process id, for $1 \leq$ id $\leq N$. Note that x is a shared variable where $0 \leq x \leq N$. Any atomic operation is enclosed in angle brackets. The `skip` statement does nothing. Assume that each process is executing on a separate processor, so we have truly parallel execution of the processes. Atomic execution of an operation here means that its effect is visible to all the processes on all processors. The variable x is simply stored in a shared memory which is accessible by all the processors.

```
repeat
     while <x != 0> <skip>;
     < x = id>; < delay >
until < x == id>;
CRITICAL SECTION
< x = 0 >;
```

Is mutual exclusion preserved when there is no `delay` operation, that is, delay == 0? Does the preservation of mutual exclusion get altered for higher values of delay? Give detailed comments.

Answer:

Mutual exclusion may not be preserved when delay == 0. The different <x= id> assignments are executed, and the mutual exclusion here depends on the entry to critical section being given to the jth process – where <x = j> was executed the last. So, two processes could execute as follows. Even though we have parallel execution the second process could execute x = j later (say due to lower processor speed).

<x = i>
Check < x = i>
Enter Critical section

<x = j>
Check <x == j>
Enter Critical section.

Therefore the <x == id> check in the repeat-until loop should wait until all the <x = id> assignments have been executed and their effect has propagated to the shared memory. Only then mutual exclusion is preserved. So, the delay should be long enough for N-1 assignments to be executed, since all the N processes might be trying to execute the assignment <x=id> at the same time.

Question 4 [10 marks]

A propositional logic formula is made out of atomic propositions, binary operators $\wedge$, $\vee$ (for logical AND, OR) and the unary operator $\neg$ (logical NOT). The Sharp-SAT or #SAT problem for propositional logic is the problem of counting the number of satisfying assignments of a given propositional logic formula. Write an MPI program which efficiently solve the #SAT problem for a given propositional logic formula, by taking advantage of parallelism. Give detailed comments on how your program is exploiting parallelism.

Answer:

We can simply try the different proposition assignments, and parallelize this generate-and-test method. MPI_Reduce can be used to count the number of solutions.

Below, I show the code for a formula with 10 propositions and 2^10 = 1024 assignments.
In general N can be taken in as input.

The procedure check_assignment(id, i) is executed by process numbered id, and it checks whether the assignment represented by the binary representation of satisfies the given formula

```
#include “mpi.h”
#include <stdio.h>

void  main(int argc, char *argv[]){

     int all_solutions;      /* the  value returned */
     int i, id, procnum, solutions;

     int check_assignment(int, int);

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &id);
     MPI_Comm_size(MPI_COMM_WORLD, &p);

     solutions = 0;
     for (i = id;   i  < 1024;  i += procnum);
          solutions  +=   check_assignment(id,  i);

     MPI_Reduce(&solutions, &all_solutions, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

     MPI_Finalize();
     if  (id == 0)  printf(“The #SAT problem for the given formula returns %d solutions\n”, all_solutions);

}
```

EMPTY PAGE

Question 5 [5 + 5 = 10 marks]

A. Two students are concurrently taking the CS3211 final exam. The pseudo-code executed by the two students are as follows. The number N is a non-negative integer. Note that s1 and s2 capture whether the two students are in/out of the exam hall. Initially, s1 == in, s2 == in, and N == 1. For each student process, each iteration of the `do-forever` loop is executed atomically.
Can you draw a global finite state machine describing the behavior of the concurrent system? Use the finite state machine to reason that two students are not out of the exam hall at the same time.

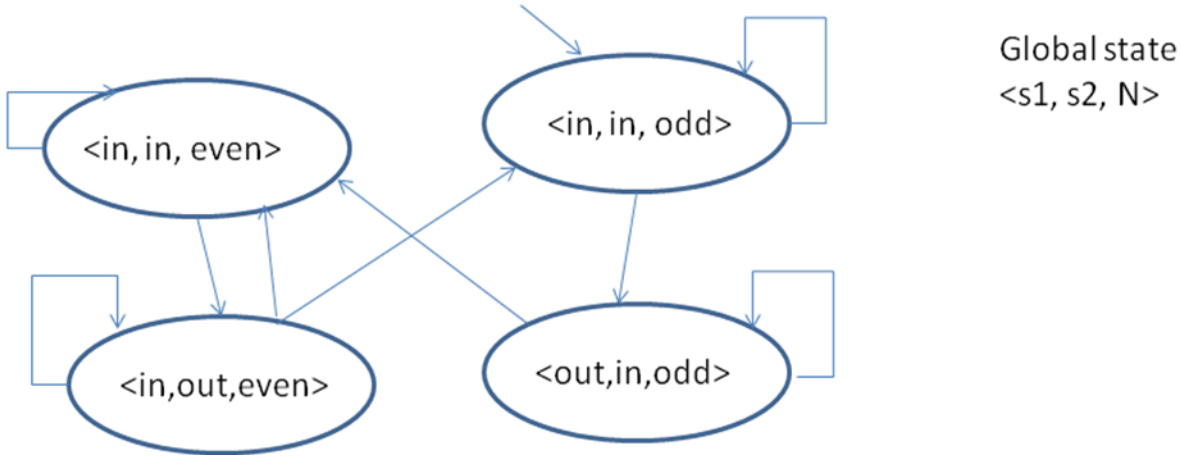
```
do forever{
   if s1 = in and N is odd
      {s1 := out; }
   else if s1 = out
      { s1 := in; N := 3*N+1}
   else {do nothing }
}
```

```
do forever{
      if s2 = in and N is even
            {s2 := out; }
      else if s2 = out and N is even
            { s2 := in ; N := N/2 }
      else { do nothing }
}
```

Answer:

Following is the global state machine. Each state is of the form <s1, s2, N>

We do not capture the exact value of N, but only whether N is an odd or even number, in order to produce a *finite* state machine. From the states of the state machine we can see that the two students can never be out of the exam hall at the same time since no state of the form <out, out, _ > is reachable from initial state.



B. For the concurrent system in previous page, develop process equations for the individual processes. Use the process equation notation used throughout the semester in our class. What are the processes you need, and how would you show their interactions? Explain each process, and each action label you use.

Answer:

We have the variable N protected as a monitor. It is like a passive object whose state changes but not on its own – it is owing to the actions of the active objects – Student1 and Student2.

```
System =  (Student1[in]  ||  Student2[in]  || Variable[1] )

/*  Imposing a bound X. This need not be imposed but the number of states becomes unbounded */

Variable[v: 0..X]  =  (read[v] -> Variable[v]    |  write[u:0..X] -> Variable[u])

Student1[local] =   (     when (local == out)  read[n] -> write[3*n+1] -> Student1[in]
                        | when (local == in)     read[n] -> (   when (n%2 == 1)   Student1[out]
                                                              | when (n%2 == 0)  Student1[local]
                                                            )
                    )

Student2(local) =     (     when (n %2 == 1)  read[n] -> Student2(local)
                          |  when (n%2 == 0)  read[n] ->  (     when (local == in)    Student2[out]
                                                              |  when (local == out)  write[n/2] -> Student2[in]
                                                            )

                    )
```

The Student1 and Student2 processes will synchronize with the Variable process via the shared actions read and write. These actions correspond to reading and writing of the shared non negative integer variable.

END of PAPER