Example of using Thread Class

```
    public static void main(String[ ] args)
    {
        System.out.println("Simple Thread Demonstration");
        System.out.println(" Extending Thread");
        for (int i = 0; i < 4; i++)
        {
            MyThread newThread = new MyThread( i );
            newThread.start( );
        }
    }
}
```

Synchronization of Java Threads

- The previous examples shows independent, asynchronous threads (i.e., run at their own pace)
- Independent threads can run asynchronously if they only use local (private) data
- In many cases, threads i) share data, and/or ii) rely on the activities of other threads
- They must synchronize to share data
- Consider the following producer-consumer example

Producer

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;

    public Producer (CubbyHole c) {
        cubbyhole = c;
    }

    public void run ( ) {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            try {
                sleep ((int) (Math.random( ) * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

- Generates *0 <= i < 9*
- Stores it in a Cubbyhole
- Sleeps for a while
- Generates the next *I*


```
public class CubbyHole {
    private int contents;
    public int get (int number)
    { … }
    public void put (int number, int value)
    { … }
```

Consumer

```
public class Consumer extends Thread
{
   private CubbyHole cubbyhole;
   public Consumer (CubbyHole c) {
      cubbyhole = c;
   }
   public void run ( ) {
      int value = 0;
      for (int i = 0; i < 10; i++) {
         value = cubbyhole.get();
      }
   }
}
```

- Consumes the integers from the Cubbyhole Object
- As quickly as they become available

Producer / Consumer Example

- CubbyHole is the shared data
- Ideally Consumer will get each value only once produced by Producer
- If Producer runs faster than Consumer,


```
Consumer #1 got: 3
Producer #1 put: 4
Producer #1 put: 5
Consumer #1 got: 5
```

- **Problem arises if Producer is quicker than Consumer**
- **Producer generates two numbers before Consumer consumes the first**
- **E.g. misses the number 4**

Producer / Consumer Example

Producer #1 put: 4
Consumer #1 got: 4
Consumer #1 got: 4
Producer #1 put: 5

- If Producer is slower than Consumer we also have a problem
- Consumer gets two same numbers before Producer generates a new number
- E.g. gets the number 4 twice

- Example of a race condition – reading and writing shared data concurrently; success depends on timing

- Example of critical section - Need synchronization

Synchronization of Java Threads

- The **put** and **get** in the CubbyHole code are the so called *critical section (monitor region)*
- These should be marked as **synchronized**


```
public class CubbyHole {
  private int contents;
  private boolean available = false;

  public synchronized int get () { … }
  public synchronized void put (int value) { … }
```


- **Running synchronized method locks class**
- **Unlocked when method terminates**

Synchronization of Java Threads

- So now we have mutual exclusion, but how do Producer and Consumer cooperate to address the race condition?
- Consumer needs to wait until Producer has put new data, and Producer needs to notify Consumer after the data is put– and vice versa

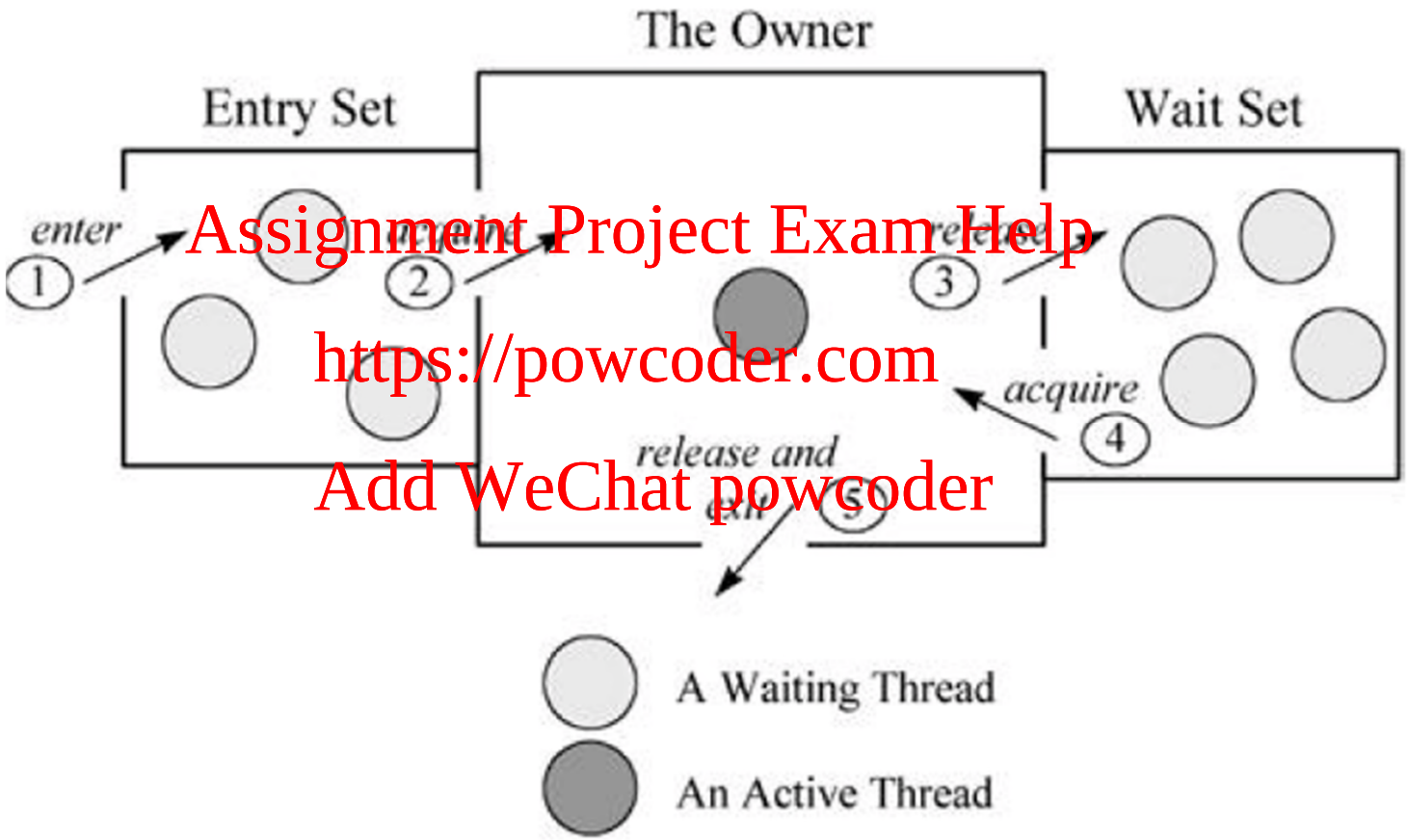
  - put and get need to wait on and notify each other of their activities


```
public synchronized int get {

  while (available == false) {
    try { // wait for producer to put value
       wait ( );
    } catch (InterruptedException e) { }
  }
  available = false;
 // Notify Producer value has been retrieved
  notify ( );
  return contents
}
```

  - available=false means the data has not been put
  - wait releases lock

Wait Releases Monitor



Synchronization of Java Threads

- So now we have mutual exclusion, but how do Producer and Consumer cooperate?
- Consumer needs to wait until Producer has put something new, and Producer needs to notify Consumer after the data is put– and vice versa
- i.e put and get need to wait on and notify each other of their activities


```
public synchronized int get {

  while (available == false) {
    try { // wait for producer to put value
       wait ( );
    } catch (InterruptedException e) { }
  }
  available = false;
 // Notify Producer value has been retrieved
  notify ( );
  return contents
}
```

- available=false means the data has not been put
- wait releases lock
- Loops until the new data is available
- When consumer gets data, it does “notify” and Producer comes out of wait state and puts the next new data

Synchronization of Java Threads

- So now we have locking, but how do Producer and Consumer cooperate?
- Consumer needs to wait until Producer has put something, at which point Producer needs to notify Consumer – and vice versa
- i.e put and get need to wait on and notify each other of their activities

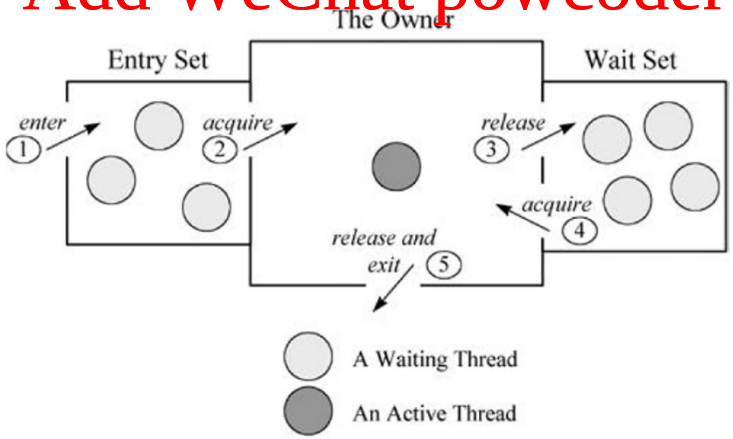

```
public synchronized void put(int value){

   while (available == true) {
      try { // wait for consumer to get value
         wait ( );
      } catch (InterruptedException e) {   }
   }
   contents = values;
   available = true;
   // Notify Consumer that value has been sent
   notify ( );
}
```


- available=true means the data has been put
- Loops until Producer ready with new value
- wait relinquishes lock
- When Producer put the data, it does notify and consumer comes out of wait state and get() can collect value

Synchronization of Java Threads

- *notify* will allow one (rather than all) thread to be woken
- *notifyAll* wakes up all threads waiting on the monitor in question
- Awakened threads compete for lock (ownership of the monitor)
- Those threads that don't get the lock continue to wait



Synchronization in Java (Monitor)

- Java provide built-in thread synchronization ability (through monitors), which is part of the programming language
- The compiler generate correct code to generate the monitor for each class/object and implement the monitor locks

For more information, see:
http://java.sun.com/docs/books/tutorial/essential/threads

Synchronisation in C

```
pthread_mutex_t my_mutex;  /* declared in global area*/


void *Calc(void *);
main(){
        pthread_mutex_init(&my_mutex, NULL) /* before pthread_create*/
        for(ith=0; ith<NUM_THREADS; ith++) pthread_create( , Calc, );
}
void *Calc(void *param){
        pthread_mutex_lock(&my_mutex);
        critical section;
        pthread_mutex_unlock(&my_mutex);
}
```

Synchronization in Java (Monitor)

- Java provide built-in thread synchronization ability (through monitors), which is part of the programming language
- The compiler generate correct code to generate the monitor for each class/object and implement the monitor locks
- Higher level and less error prone than forcing programmers to explicitly handle locks

For more information, see:
http://java.sun.com/docs/books/tutorial/essential/threads

THE UNIVERSITY OF WARWICK

High Performance Computing *Course Notes*

Message Passing Programming I

Dr Ligang He

Message Passing Programming

- Message Passing is the most widely used parallel programming model
- Message passing works by creating a number of processes, uniquely named, that interact by sending and receiving messages to and from one another (hence the message passing)
  - Generally, processes communicate through sending the data from the address space of one process to that of another
    - Communication of processes (via message, socket, pipe, files)
    - Communication of threads within a process (via global data area)
- Message passing programs can be based on standard sequential language programs (C/C++, Fortran), augmented with calls to library functions for sending and receiving messages

Message Passing Interface (MPI)

- There are different **message passing models**; MPI is the most popular models (PVM is another one);
- MPI is a **specification**, not a particular implementation
  - Specify a number of routines as well as the parameters that the routines should have
  - Does not specify how these routines are implemented, nor process startup, error codes, amount of system buffer, etc
- MPI **implementation** is a library, not a language
  - There are different MPI implementations: MPICH, LAM/MPI, OpenMPI
- In terms of the scope, Message passing model > MPI specification > MPI implementation

OpenMP vs MPI

- MPI is used on distributed-memory systems
- OpenMP is used on shared-memory systems
- Both are explicit parallelism
- OpenMP is higher-level control
  - Compiler can automatically parallelise the codes when instructed
- MPI is lower-level control
  - Data partition, allocation and communication are conducted by programmers

A brief history of MPI

- Message-passing libraries developed for a number of early distributed memory computers
- By 1993 there were many vendor specific implementations
- By 1994 MPI-1 was proposed
  - Emphasize message passing
  - Has a static runtime environment
- By 1996 MPI-2 was finalized
  - includes new features such as
    - parallel I/O,
    - dynamic process management
    - remote memory operations
- 2012 MPI-3 is formalized

The MPI programming model

- MPI standards -
  - MPI-1, MPI-2, MPI-3
- Standard bindings - for C, C++ and Fortran.
- There are MPI bindings for Python, Java etc (non-standard)
- We will stick to the C/C++ binding, for the lectures and coursework.

MPI functions

- MPI is a complex system comprising of numerous
- functions with various parameters and variants

- Six of them are indispensable, but can write a large number of useful programs already

- Other functions add other features, including flexibility (datatype), robustness (non-blocking send/receive), efficiency (ready-mode communication), modularity (communicators, groups) or convenience (collective operations, topology).

- In the lectures, we are going to cover most commonly encountered functions

Intuitive Interfaces for sending and receiving messages

- Send(data, destination), Receive(data_location, source)
  - minimal interface
- Not enough in some situations, we also need
  - Message matching – sender may send multiple messages to the same receiver, add message id at both send and receive interfaces
  - they become Send(data, destination, msg_id), receive(data, source, msg_id)
  - Message_id
    - Is expressed using an integer, termed as *message tag*
    - Can differentiate the messages from the same process
    - Enable the messages to be processed in an ordered fashion

How to express the data in the send/receive interfaces

- Early stages:
  - (address, length) for the send interface
  - (address, max_length) for the receive interface
- They are not always good
  - the data may not occupy contiguous memory locations
  - Storing format for data may not be the same in heterogeneous platform
- Enventually, a triple (address, count, datatype) is used to express the data to be sent and (address, max_count, datatype) for the data to be received
  - Reflecting the fact that a message contains much more structures than just a string of bits, For example, (vector_A, 300, MPI_REAL)
  - Programmers can construct their own datatype
- Now, the interfaces become send(address, count, datatype, destination, msg_tag) and receive(address, max_count, datatype, source, msg_tag)

How to distinguish messages

- **Message tag is necessary, but not sufficient**

- **So, communicator is introduced ...**

Communicators

- Messages are put into communication contexts
  - Contexts are allocated at run time by the MPI system
  - Each generated context is unique
- The processes in a MPI system are divided into groups
- The notions of context and process group are combined in a single object, which is called a *communicator*
  - A communicator consists of a group of processes and a communication context
  - The MPI library defines a initial communicator, MPI_COMM_WORLD, which contains all the processes running in the system
- Messages from different communicators can have the same tag
- So the send interface becomes send(address, count, datatype, destination, tag, comm)

Status of the received messages

- **The structure of the message status is added to the receive interface**
- **Status holds the information about source, tag and actual message size**
  - In the C language, source can be retrieved by accessing status.MPI_SOURCE,
  - tag can be retrieved by status.MPI_TAG and
  - actual message size can be retrieved by calling the function MPI_Get_count(&status, datatype, &count)
- **The receive interface becomes receive(address, maxcount, datatype, source, tag, communicator, status)**

How to express source and destination

- The processes in a communicator (group) are identified by ranks
- If a communicator contains n processes, process ranks are integers from 0 to n-1
- Source and destination processes in the send/receive interface are the ranks

Some other issues

In the receive interface, tag can be a wildcard (MPI_ANY_TAG), which means any message will be received

In the receive interface, source can also be a wildcard (MPI_ANY_SOURCE), which match any source

MPI basics

First six functions (C bindings)

MPI_Send (buf, count, datatype, dest, tag, comm)

Send a message

buf	*address of send buffer*
count	*no. of elements to send (>=0)*
datatype	*of elements*
dest	*process id of destination*
tag	*message tag*
comm	*communicator (handle)*

MPI basics

First six functions (C bindings)

MPI_Send (buf, count, datatype, dest, tag, comm)

Send a message

buf	*address of send buffer*
count	*no. of elements to send (>=0)*
datatype	*of elements*
dest	*process id of destination*
tag	*message tag*
comm	*communicator (handle)*

MPI basics

First six functions (C bindings)

MPI_Send (buf, count, datatype, dest, tag, comm)

Send a message

buf	*address of send buffer*
count	*no. of elements to send (>=0)*
datatype	*of elements*
dest	*process id of destination*
tag	*message tag*
comm	*communicator (handle)*

MPI basics

First six functions (C bindings)

MPI_Send (buf, count, datatype, dest, tag, comm)

Calculating the size of the data to be send …

buf *address of send buffer*

*count * sizeof(datatype)* *bytes of data*

MPI basics

First six functions (C bindings)

MPI_Send (buf, count, datatype, dest, tag, comm)

Send a message

buf — *address of send buffer*

count — *no. of elements to send (>=0)*

datatype — *process id of destination*

dest — *process id of destination*

tag — *message tag*

comm — *communicator (handle)*

MPI basics

First six functions (C bindings)

MPI_Send (buf, count, datatype, dest, tag, comm)

Send a message

buf	*address of send buffer*
count	*no. of elements to send (>=0)*
datatype	*process id of destination*
dest	*process id of destination*
tag	*message tag*
comm	*communicator (handle)*

MPI basics

First six functions (C bindings)

MPI_Send (buf, count, datatype, dest, tag, comm)

Send a message

buf — *address of send buffer*

count — *no. of elements to send (>=0)*

datatype — *process id of destination*

dest — *process id of destination*

tag — *message tag*

comm — *communicator (handle)*

MPI basics

First six functions (C bindings)

MPI_Recv (buf, count, datatype, source, tag, comm, status)

Receive a message

buf	*address of receive buffer*
count	*max no. of elements in receive buffer (>=0)*
datatype	*of receive buffer elements*
source	*process id of source process, or MPI_ANY_SOURCE*
tag	*message tag, or MPI_ANY_TAG*
comm	*communicator*
status	*status object*

MPI basics

First six functions (C bindings)

MPI_Init (int *argc, charargv)**

Initiate a MPI computation

argc(number of arguments) and argv(argument vector) hold main program's arguments

Must be called first, and once per process

MPI_Finalize()

Shut down a computation

The last thing that happens

MPI basics

First six functions (C bindings)

MPI_Comm_size (MPI_Comm comm, int* size)

Determine number of processes in comm

comm is communicator handle,

*MPI_COMM_WORLD is the default (including **all** MPI processes)*

***size** holds number of processes in group*

MPI_Comm_rank (MPI_Comm comm, int* pid)

Determine id of current (or calling) process

***pid** holds id of current process*

MPI basics – a basic example

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
int rank, nprocs;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
printf("Hello, world. I am %d of %d\n", rank, nprocs);
MPI_Finalize();
}
```



```
mpirun –np 4 myprog
Hello, world. I am 1 of 4
Hello, world. I am 3 of 4
Hello, world. I am 0 of 4
Hello, world. I am 2 of 4
```

MPI basics – send and recv example (1)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
int rank, size, i;
int buffer[10];
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (size < 2) {
        printf("Please run with two processes.\n");
        MPI_Finalize();
        return 0;
}
if (rank == 0) {
        for (i=0; i<10; i++)
            buffer[i] = i;
        MPI_Send(buffer, 10, MPI_INT, 1, 123, MPI_COMM_WORLD);
}
```

MPI basics – send and recv example (2)

```
if (rank == 1) {
        for (i=0; i<10; i++)
            buffer[i] = -1;
        MPI_Recv(buffer, 10, MPI_INT, 0, 123, MPI_COMM_WORLD,
&status);
        for (i=0; i<10; i++) {
            if (buffer[i] != i)
      printf("Error: buffer[%d] = %d but is expected to be %d\n", i, buffer[i], i);
        }
}
MPI_Finalize();
}
```

MPI language bindings

- Standard (accepted) bindings for Fortran, C and C++
- Two types of Java bindings (work in progress)
  - native MPI bindings: native MPI library is called by Java programs through Java wrappers
    - JavaMPI
    - mpiJava
  - pure Java implementations: the whole MPI library is rewritten in Java
    - jmpi
    - MPIJ
- Java Grande Forum trying to sort it all out
- We will use the C bindings