Announcements

Wednesday, 27 July

• Final Exam
• In-class
• You can bring with you one piece of paper with notes
• No electronic help of any kind is allowed

Thursday, 28 July

• Last lab (Message passing)
Q: What is the difference between a heavyweight and lightweight process?
From last time

Heavyweight processes

Lightweight processes

Thread 1

Thread 2

Thread 3

get()
From last time

Synchronous versus Asynchronous message passing

Q: Is the above an example of synchronous or asynchronous message passing?
From last time

Synchronous versus Asynchronous message passing

P1
- Send()
- Suspend()
- Transfer()

P2
- Receive()
- Suspend()

request
ack
message

P1
- gather()
Today

MPI
Rendezvous
Scenario: There are 5 processes, each of which has a locally scoped variable. The goal is for EVERY process to learn of the smallest and largest of the locally scoped variables among the 5 processes.
Scenario: There are 5 processes, each of which has a locally scoped variable. The goal is for EVERY process to learn of the smallest and largest of the locally scoped variables among the 5 processes.

```
<table>
<thead>
<tr>
<th>Process</th>
<th>myVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>P3</td>
<td>21</td>
</tr>
<tr>
<td>P5</td>
<td>-4</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

Shopping Cart

Q: What is the fewest number of messages that must be sent among the 5 processes (and in which order), to achieve the goal?

(on the board explanation)
Q: Which approach is best?

**Centralized**

**Symmetric**
Topologies

Which of these two allows for the **fastest** run time (requires least amount of time from start to finish)?

Assume dedicated paths, and all processes are capable of running concurrently

**Centralized**

- Time to send: 2s
- Time to sort: 10s

**Symmetric**

- Time to send: 10s
- Time to sort: 2s
Topologies

How much work (sum of all work done by individual processes) is required?

Assume dedicated paths, and all processes are capable of running concurrently.

Centralized

Symmetric

Time to send : 2s    Time to sort : 10s
Time to send : 10s   Time to sort : 2s
Can we assume that messages are sent and received in parallel?
Topologies

Can we assume that messages are sent and received in parallel?

As $n$ becomes large, dedicated paths cannot be assumed. How does that affect your choice of topology in terms of # of messages that are sent, and time needed?
Can we assume that messages are sent and received in parallel?

As $n$ becomes large, dedicated paths cannot be assumed. How does that affect your choice of topology in terms of # of messages that are sent, and time needed?
MPI = Message passing Interface

Lab 8
MPI = Message passing Interface

**MPI_Init** : initialize the MPI library

**MPI_COMM_WORLD** : The “world”

**MPI_Comm_size** : the number of processes started

**MPI_Comm_rank** : Determine your rank (unique ID)

**MPI_Send** : send another message. Includes the message, as well as recipient

**MPI_Receive** : receive a message from another process. Specifies into which buffer the data should be received

**MPI_Finalize** : terminate (knowledge of IDs is needed to terminate all processes)
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){

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

    // Parallel Region

    MPI_Finalize();
}
```
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    // Parallel Region
    MPI_Finalize();
}
```

Every instance of the program (every Processor is running this code) gets a copy of the arguments with which the program was started.
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    // Parallel Region
    MPI_Finalize();
}
```

Determine the number of processes working on “this” job, and what is “my” ID
MPI

MPI = Message passing Interface

#include<mpi.h>
main(int argc, char** argv){

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

  // Parallel Region

  MPI_Finalize();
}

After the parallel regions are “done,” then everything is cleaned up.

Q: How does MPI enable all processes to coordinate with each other?
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){

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

    // Parallel Region
    MPI_Finalize();
}
```

After the parallel regions are “done,” then everything is cleaned up. There are message that enable: barriers, bcast, scatter, gather, reduce, etc.
MPI = Message passing Interface

```c
#include<mpi.h>
main(int argc, char** argv){

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

  // Parallel Region

  MPI_Finalize();
}
```

MPI is a heavyweight process system, so ALL processes have the entirety of this program. Thus if you want a process to be a coordinator ...
The parallel regions usually has code for a coordinator and a worker, and the process ID (rank) is used to distinguish between the coordinator and worker.
MPI = Message passing Interface

```c
if (my_rank == 0){
    // receive
    // calculate

} else{
    // calculate
    // send
}
```
Motivation for RPC and Rendezvous

Message passing is great for interacting Peers

Q: But when is message passing a poor choice for “implementing” concurrency?
Motivation for RPC and Rendezvous

Message passing is great for interacting Peers

Q: But when is message passing a poor choice for “implementing” concurrency?

• Synchronous message passing includes both request and acknowledgement messages in ADDITION to the actual message that is sent

Q: Why is this “bad”? 
Motivation for RPC and Rendezvous

Message passing is great for interacting Peers

Q: But when is message passing a poor choice for “implementing” concurrency?

• Synchronous message passing includes both request and acknowledgement messages in ADDITION to the actual message that is sent

Many “administration” messages are needed

Q: What is another reason?
Motivation for RPC and Rendezvous

Message passing is great for interacting Peers

Q: But when is message passing a poor choice for “implementing” concurrency?

- Synchronous message passing includes both request and acknowledgement messages in ADDITION to the actual message that is sent
- Either ALL clients are listening on the same channel, or all client-client connections are along dedicated paths

Q: Why is this “bad”?

Many “administration” messages are needed
Motivation for RPC and Rendezvous

Message passing is great for interacting Peers

Q: But when is message passing a poor choice for “implementing” concurrency?

• Synchronous message passing includes both request and acknowledgement messages in ADDITION to the actual message that is sent

• Either ALL clients are listening on the same channel, or all client-client connections are along dedicated paths, which necessitates a large (impractical) number of channels as the count of interacting peers increases

Many “administration” messages are needed

Dedicated channel nightmare and/or shared channel congestion
Motivation for RPC and Rendezvous

Message passing is great for interacting Peers

Q: But when is message passing a poor choice for “implementing” concurrency?

- Synchronous message passing includes both request and acknowledgement messages in addition to the actual message that is sent.

- Either ALL clients are listening on the same channel, or all client-client connections are along dedicated paths, which necessitates a large (impractical) number of channels as the count of interacting peers increases.

Q: How did monitors “solve” these problems?

Many “administration” messages are needed.

Dedicated channel nightmare and/or shared channel congestion.
Motivation for RPC and Rendezvous

Message passing is great for interacting Peers

Q: But when is message passing a poor choice for “implementing” concurrency?

- Synchronous message passing includes both request and acknowledgement messages in ADDITION to the actual message that is sent
  
  Many “administration” messages are needed

- Either ALL clients are listening on the same channel, or all client-client connections are along dedicated paths, which necessitates a large (impractical) number of channels as the count of interacting peers increases
  
  Dedicated channel nightmare and/or shared channel congestion

Monitors are passive, and as many active processes as we want can access and use a SINGLE monitor, which maintains queues to orchestrate FIFO blocking and unblocking
Motivation for RPC and Rendezvous

Task: Let us try to “speed” this up ...

Q: What would you do?

- Use of monitors includes processes and a monitor
- Processes communicate and synchronize by calling procedures declared in a monitor
- Processes and monitors are all in the same address space
A Remote Procedure Call is a mechanism by which procedures are executed by CPUs/computers that have available resources.

- Use of monitors includes processes and a monitor
- Processes communicate and synchronize by calling procedures declared in a monitor
- Processes and monitors are all in the same address space
Use of monitors includes processes and a monitor
- Processes communicate and synchronize by calling procedures declared in a monitor
- Processes and monitors are all in the same address space

An RPC has only a single program component – the module
- Modules have both processes and procedures
- Modules may be scattered among different addresses spaces as for example in different nodes in a network
• Use of monitors includes processes and a monitor
• Processes communicate and synchronize by calling procedures declared in a monitor
• Processes and monitors are all in the same address space
• An RPC has only a single program component – the module
• Modules have both processes and procedures
• Modules may be scattered among different addresses spaces as for example in different nodes in a network
- Use of monitors includes processes and a monitor
- Processes communicate and synchronize by calling procedures declared in a monitor
- Processes and monitors are all in the same address space

- An **RPC** has only a single program component – the module
- Modules have both processes and procedures
- Modules may be scattered among different addresses spaces as for example in different nodes in a network

The calling process sends a message, which is a call to a needed function

The calling process then delays (self stalls)
Use of monitors includes processes and a monitor.
Processes communicate and synchronize by calling procedures declared in a monitor.
Processes and monitors are all in the same address space.

An **RPC** has only a single program component – the module.
Modules have both processes and procedures.
Modules may be scattered among different addresses spaces as for example in different nodes in a network.

A new process (not already running) services the call.
RPC

- Use of monitors includes processes and a monitor
- Processes communicate and synchronize by calling procedures declared in a monitor
- Processes and monitors are all in the same address space

- An RPC has only a single program component – the module
- Modules have both processes and procedures
- Modules may be scattered among different addresses spaces as for example in different nodes in a network

A new process (not already running) services the call
And then returns (sends a message), and terminates
Use of monitors includes processes and a monitor
Processes communicate and synchronize by calling procedures declared in a monitor
Processes and monitors are all in the same address space
An RPC has only a single program component – the module
Modules have both processes and procedures
Modules may be scattered among different addresses spaces as for example in different nodes in a network

When the calling process receives the results, and continues
Q: How does an RPC differ from an Rendezvous?
RPC and Rendezvous

RPC

Calling Process → Server Process

Call

Return
RPC and Rendezvous

**RPC**

- Calling Process
- Server Process
- Call
- Return

**Rendezvous**

- Calling Process
- Server Process
- Rendezvous
- Return
Sample Final Exam
Up Next

Final Exam