Announcement

Schedule for remainder of term

• Friday, 4 March : last lab (Message passing)
• Monday, 7 March : lecture
• Tuesday, 8 March : final exam prep, course evals
• Wednesday, 9 March : no lecture – work on final project
• Friday, 10 March : no lab – work on final project
• Thursday, 17 March : final exam, 8-10am
  • If you score higher on the final exam than you did on the midterm exam, the final exam score will replace your midterm exam score.
From last time

A monitor is following one of these disciplines, but NOT both.

**Signal and continue (SC):** the signaler (process that called signal) continues, and the signaled (process that is awakened) executes at some later time.

**Signal and wait (SW):** the signaler waits until some later time and the signaled process executes immediately.
From last time

**Signal and wait (SW):** the signaler waits until some later time and the signaled process executes immediately.

If a process is executing, and it issues `signal`, what happens next, assuming SW?

- **entry queue**
- **cv queue**
- **Execution “in” monitor**
From last time

Signal and wait (SW) : the signaler waits until some later time and the signaled process executes immediately.

If a process is executing, and it issues signal, what happens next, assuming SW?

The signaler “waits” by being placed at the end of the entry queue.
From last time

Signal and wait (SW): the signaler waits until some later time and the signaled process executes immediately.

If a process is executing, and it issues signal, what happens next, assuming SW?

The signaled process (that is at the head of cv) executes immediately.
Message Passing
Processes and Threads

Up until this point, we’ve been talking about “threads”, in which a heap is a shared memory resource that is dynamically allocated.
Processes and Threads

Up until this point, we’ve been talking about “threads”, in which a heap is a shared memory resource that is dynamically allocated.

Threads “communicate” via shared memory resources (an array, for example).
Processes and Threads

Up until this point, we’ve been talking about “threads”, in which a heap is a shared memory resource that is dynamically allocated.

Threads “communicate” via shared memory resources (an array, for example)
Processes and Threads

Up until this point, we’ve been talking about “threads”, in which a heap is a shared memory resource that is dynamically allocated.

Threads “communicate” via shared memory resources (an array, for example) or via a `get()` method that retrieves the result of each thread’s calculation.

A threading approach is well suited for a multi-core, single processor architecture.
Processes and Threads

We’ve also talked about Processes in the context of locks and barriers, and critical regions.

Up until now, we haven’t distinguished between a **lightweight** process and a **heavyweight** process.
Processes and Threads

We’ve also talked about Processes in the context of locks and barriers, and critical regions.

Up until now, we haven’t distinguished between a lightweight process and a heavyweight process.

Each of these threads can be thought of as a lightweight process, because it is not a self-contained entire program.

Q: So what is a heavyweight process?
Processes and Threads

(heavyweight process)
Process Approach

In Contrast ...

These are **heavyweight processes** because each has their own heap AND stack

This is a distributed memory model, and the processes can be run on different CPUs on different racks, machines, etc.

Q: How do such processes communicate?
Processes and Threads

(heavyweight process)
Process Approach

In Contrast ...

These are heavyweight processes because each has their own heap AND stack

This is a distributed memory model, and the processes can be run on different CPUs on different racks, machines, etc.

Q: How do such processes communicate?

First, there needs to be a physical connection

Q: Is having a “connection” enough to facilitate communication?
In Contrast ...

These are **heavyweight processes** because each has their own heap AND stack.

This is a distributed memory model, and the processes can be run on different CPUs on different racks, machines, etc.

**Q: How do such processes communicate?**

**First, there needs to be a physical connection**

There also needs to be a communication protocol.
Message passing is concerned with sending messages either asynchronously or synchronously among heavyweight processes on a distributed memory architecture.

Task: Be able to explain the difference between synchronous and asynchronous message passing. Be able to discuss the pros and cons of each.
Synchronous Message Passing

Synchronous message passing: when a sender and receiver synchronize HOW they communicate ...

If you have 2 processes sending messages back and forth to each other, HOW do they communicate?

Hint: Think of how humans communicate
Q: Is there a protocol?
Synchronous Message Passing

Synchronous message passing: when a sender and receiver synchronize HOW they communicate ...

If Process 1 wants to communicate with Process 2, how is that achieved? (if you’ve taken Networks, this communication process should be familiar)
Synchronous message passing: when a sender and receiver synchronize HOW they communicate ...

P1 wants to communicate with P2, but before it can “send” data to P2, it must first ask “hey are you there.” After the request is sent, P1 suspends itself.
Synchronous message passing: when a sender and receiver synchronize HOW they communicate...

P2 receives a request, and invokes the Receive() method, which sends an acknowledgement message to P1 to the effect of “okay I’m listening.” P2 then suspends itself.
Synchronous message passing: when a sender and receiver synchronize HOW they communicate ...

Once P1 receives the acknowledge message from P2 ...
Synchronous message passing: when a sender and receiver synchronize HOW they communicate ...

It sends the message (data) via the network, to P2.
Synchronous Message Passing

Synchronous message passing: when a sender and receiver synchronize HOW they communicate...

Which is received and processed by P2
Synchronous message passing: when a sender and receiver synchronize \textbf{HOW} they communicate ...

In this scenario, the \texttt{Send()} routine is a \textbf{blocking routine}, because after a process sends a message, it self blocks.
Synchronous Message Passing

Synchronous message passing: when a sender and receiver synchronize HOW they communicate...

Q: What are the disadvantages of synchronous message passing?
Synchronous message passing: when a sender and receiver synchronize HOW they communicate...

Q: What are the disadvantages of synchronous message passing?

1. Concurrency is vastly reduced, because there is always one process that is blocked and waiting
Synchronous Message Passing

Synchronous message passing: when a sender and receiver synchronize HOW they communicate...

Q: What are the disadvantages of synchronous message passing?

1. Concurrency is vastly reduced, because there is always one process that is blocked and waiting.
2. There is the possibility of deadlock when two processes are “coded” incorrect.

Q: Why will these processes deadlock?

Q: How might you fix the deadlock?
Synchronous Message Passing

Synchronous message passing: when a sender and receiver synchronize HOW they communicate ... 

Q: What are the disadvantages of synchronous message passing?

1. Concurrency is vastly reduced, because there is always one process that is blocked and waiting.
2. There is the possibility of deadlock when two processes are “coded” incorrect.
3. The method of making the connection is error prone.
Message Passing

Q: How does asynchronous message passing differ from synchronous message passing?
Asynchronous Message Passing

Q: What is the human communication “analogy” to asynchronous message passing?

Hint: Think of how humans communicate in a large group setting
Q: Is there a protocol?
Q: Do people wait until it is their turn to speak?
Asynchronous Message Passing

Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.
Asynchronous Message Passing

Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.
Asynchronous Message Passing

Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the disadvantages of an asynchronous broadcast?
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the disadvantages of an asynchronous broadcast?

1. Flooding of the network, which might cause congestion
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the disadvantages of an asynchronous broadcast?

1. Flooding of the network, which might cause congestion
2. Communication overhead diminishes a program’s performance, thus sending too many messages is detrimental to a program’s performance
Asynchronous Message Passing

Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the disadvantages of an asynchronous broadcast?

1. Flooding of the network, which might cause congestion
2. Communication overhead diminishes a program’s performance, thus sending too many messages is detrimental to a program’s performance
3. P1 has an excessively high computational burden (it makes and sends all the messages)
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

The “human” analogy: When a manager wants to delegate tasks to subordinates, it...
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

The “human” analogy: When a manager wants to delegate tasks to subordinates, it ...
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the drawbacks of using scatter?
Asynchronous Message Passing

Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the drawbacks of using scatter?

1. Flooding of the network, which might cause congestion if $n$ is large
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the drawbacks of using scatter?

1. Flooding of the network, which might cause congestion if $n$ is large
2. P1 must create $n-1$ unique “messages,” whereas in broadcast P1 makes 1 message and sends it out multiple copies
Asynchronous Message Passing

Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Once portions of the problem have been sent out (and computations performed), what is the manager’s role?
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Collect the “work done”
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the drawbacks of using gather?
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the drawbacks of using gather?

1. P1 is now the location of a major bottleneck, because it receives upwards of \( n-1 \) messages.
Asynchronous Message Passing

Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the drawbacks of using gather?

1. P1 is now the location of a major bottleneck, because it receives upwards of n-1 messages
2. If each of P2 through Pn finish “at the same time,” network congestion will be large
Humans do not always wait their turn to talk. Some just start talking. Processes can do the same.

Q: What are the drawbacks of using gather?

1. P1 is now the location of a major bottleneck, because it receives upwards of n-1 messages
2. If each of P2 through Pn finish “at the same time,” network congestion will be large
3. Assuming P1 receives all of the messages from P2 through Pn, it must synthesize (combine) them into a coherent whole
Q: Regardless whether broadcast, scatter, or gather are used, their ultimate efficiency is based on what 2 factors?
Q: Regardless whether broadcast, scatter, or gather are used, their ultimate efficiency is based on what 2 factors?

a) The nature of the problem
b) The Topology of the network
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.

Q: What kind of topology is best?
Q: Which CPU(s) should perform the calculations?

(on the board discussion)
In a centralized approach, there is a single CPU (1 in this case) that is designated as the coordinator.
In a centralized approach, there is a single CPU (1 in this case) that is designated as the coordinator.
### Topologies

<table>
<thead>
<tr>
<th>Dissemination Approach</th>
<th>Fewest Number of messages sent</th>
<th>Pros/Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a centralized approach, there is a single CPU (1 in this case) that is designated as the coordinator.

(on the board discussion)
In a symmetric approach, all processes (nodes) send and receive messages from all others.
### Topologies

<table>
<thead>
<tr>
<th>Dissemination Approach</th>
<th>Fewest Number of messages sent</th>
<th>Pros/Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td></td>
<td>In class exercise</td>
</tr>
<tr>
<td>Symmetric</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a symmetric approach, all processes (nodes) send and receive message from all others.
### Topologies

<table>
<thead>
<tr>
<th>Dissemination Approach</th>
<th>Fewest Number of messages sent</th>
<th>Pros/Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symmetric</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a symmetric approach, all processes (nodes) send and receive message from all others.
### Topologies

<table>
<thead>
<tr>
<th>Dissemination Approach</th>
<th>Fewest Number of messages sent</th>
<th>Pros/Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symmetric</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shopping Cart</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How many messages must be passed as a function of \( n \), the number of processes? Assume messages are sent and received in parallel (each path is dedicated).

(centralized system diagram)

(oned diagram)

(on the board discussion)
Topologies

Which of these two allows for the **fastest** approach (requires least amount of time)? Assume messages are sent and received in parallel (each path is dedicated), and that processing (sorting) and sending a message takes the same amount of time.

Centralized

Symmetric

(on the board discussion)
How much work is done and how is it distributed?
Assume messages are sent and received in parallel (each path is dedicated), and that sorting takes 10 units of time, and sending a message takes 2 units of time.

Centralized

Symmetric

(on the board discussion)
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.

What if we cannot 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?
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.

What if we cannot 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

MPI = Message passing Interface
MPI

MPI = Message passing Interface

**MPI_Init**: initialize the MPI library
**MPI_COMM_WORLD**: number of processes available
**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)
#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 Processors 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 = 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.

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();
}
```

Recall that 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 ...
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();
}
```

The parallel regions usually has code for a coordinator and a worker, and the process ID (rank) is used to distinguish the coordinator.
MPI = Message passing Interface

```c
if (my_rank == 0){
    // receive
    // calculate
} else{
    // calculate
    // send
}
```
Up Next

Rendezvous