So far in this class…

- We have assumed a single address space
  - Threads have communicated via shared variables
    - E.g., bounded buffer and readers/writers problems
  - Synchronization objects (locks, barriers, semaphores, monitors) have been placed in shared memory
    - Accessible by all threads
- What if we want threads in two different address spaces to communicate?
  - Cannot simply use a synchronization object to communicate, because they do **not** share memory

Message Passing

- Thread T_1 in address space A_1 sends a message to thread T_2 in address space A_2
  - Where are these threads/address spaces located?
    - Same machine? Different machine? Does it matter?
    - A_1 and A_2 certainly *could* be on different machines; if so, the message must travel over a network; how does it get there?
    - Does the programmer have to worry about this?
  - What is in the message?
- Because a thread plus an address space is a process, sometimes we will refer to processes P_1 and P_2 in the above

Message Passing

- In this class:
  - Message passing can occur only between threads in different address spaces
    - Again, we will use the term *processes* often
  - Message passing cannot occur between threads in the same address space. Why?

Message Passing

- In this class:
  - Message passing can occur only between threads in different address spaces
    - Again, we will use the term *processes* often
  - Message passing cannot occur between threads in the same address space. Why?
    - Should use shared variables for communication between threads in the same address space
    - Could also be thought of as a process sending to and receiving from itself

Physical Reality of Networks

- Networks are unreliable
  - messages are divided into packets
  - packets can get lost
  - packets can arrive out of order
  - receiver can get overloaded
    - cannot handle rate of packet arrival

Define a new abstraction: Channels

- Analogous to abstractions in OS's
  - process -- abstraction of a processor
  - virtual memory -- abstraction of unlimited memory
  - files -- abstraction of disk
- Want to abstract communication network
  - don't want to worry about lost packets, out-of-order packet arrival, overflowing buffers, etc.
  - **Channel** -- abstraction of point to point, reliable communication link

Send and Receive

- Send(channel, expr1, expr2, …, exprN)
  - Sends a message on the channel indicated
  - Expressions *expr1*, *expr2*, …, *exprN* can be l-vals or r-vals
- Receive(channel, arg1, arg2, …, argN)
  - Receive a message from the channel indicated into *arg1, arg2, …, argN*
  - Variables *arg1*, *arg2,* …, *argN* must be l-vals (must provide a storage location)
  - Receive blocks until data is available on channel
    - Can be relaxed in real implementations

“Ping-Pong” example
chan PingPong[2](int x) is shared

Process 0 code	Process 1 code
int a = 10, b	int a
Send(PingPong[1], a)	Receive(PingPong[1], a)
Receive(PingPong[0], b)	Send (PingPong[0], a)
print a, b	print a

Notes:

- Process 0 sends to Process 1; Process 1 receives and sends back
- Output is 10, 10 for Process 0; 10 for Process 1
- Need two channels to avoid self-send/self-receive
- Note that Receive takes its variable as an l-val

Ping-Pong code, shorthand Will be the format on quizzes

Process 0 code	Process 1 code
int a = 10, b	int a
Send(1, a)	Receive(0, a)
Receive(1, b)	Send (0, a)
print a, b	print a

Notes:

- Exact same idea as the version with channels
- Difference is Send names the process to which message is sent, and Receive names the process from which message is received
- Send/Receive match if each names the other and number and types of arguments match

Exchange example
chan Exchange[2](int x) is shared

Process 0 code	Process 1 code
int a = 10, b	int a, b = 8
Send(Exchange[1], a)	Send(Exchange[0], b)
Receive(Exchange[0], b)	Receive(Exchange[1], a)
print a, b	print a, b

Notes:

- Both processes send to the other and then receive from the other
- Output is 10, 8 for both processes
- Need two channels to avoid self-send/self-receive
- Note that Receive takes its variable as an l-val

Exchange code, shorthand
Will be the format on quizzes

Process 0 code	Process 1 code
int a = 10, b	int a, b = 8
Send(1, a)	Send(0, b)
Receive(1, b)	Receive(0, a)
print a, b	print a, b

Notes:

- Exact same idea as the version with channels
- Difference is Send names the process to which message is sent, and Receive names the process from which message is received

Example Quiz Question

Assume **each process** has two integers, a and b, initialized to 0

What are the final values of a and b in each process?

Does the code terminate?

```
P0 code
  Send(1, a+1)
  Receive(1, b)
P1 code
  Receive(0, b)
  Send(2, b+1)
  Receive(2,  a)
  Send(0, a+1)
P2 code
  Receive(1, b)
  Send(1, b+1)
```

Another example
chan S(int x) is shared

Process 0 code

Send(S, 10)

Process 1 code

int a = 5; Receive(S, a); print “I am P1”, a

Process 2 code

int a = 5; Receive(S, a); print “I am P2”, a

This will deadlock. What happens if there is another send in P0, of a different value? Which process receives which value?

Send and Receive

- Notes:
  - Channel handles reliability
    - Must be implemented by network protocols
  - Access to channel is atomic
  - Message has to be buffered if it arrives but receiver has not yet invoked receive
    - In fact, receiver's view of channel is a FIFO queue of pending messages
  - Implementation requires synchronization
  - Send, Receive can be OS kernel primitives or can be library primitives (e.g., MPI, the library we will use for program 3)

Send and Receive

- Notes, continued:
  - Special case: both *exprs* and *args* are the empty set; in this case:
    - Send is analogous to V(s)
    - Receive is analogous to P(s)
    - Number of pending “messages” is analogous to the value of *s*
      - More precisely, number of pending invocations

Duality of Monitors/Message Passing

- First observed by Lauer and Needham in 1978
  - "On the Duality of Operating System Structures"
  - Observed that a monitor program can be translated mechanically into a message passing program (and vice versa)

Resource Allocation with Monitors

```
monitor ResourceAllocator
  int free = true; cond c
  acquire( ):      if (free) free = false
                   else wait(c)
  release( ):      if (empty(c)) free = true
                   else signal(c)
end ResourceAllocator
```

Client calls ResourceAllocator.acquire( )/ResourceAllocator.release( )

Resource Allocation with Message Passing Server-side code

```
enum reqType :={ACQUIRE, RELEASE}
chan request(int clientId, reqType which)
chan reply[n]( )   // one entry per client, no params
Allocator ( )        // runs on server
   queue pending;  // initially empty
   int clientId;  bool free := true
   reqType which;
```

Resource Allocation with Message Passing
Server-side code, continued

```
while (1) {
    receive(request, clientId, which)
    switch(which) {
     ACQUIRE:
        if (free)
          free := false; send(reply[clientId])
        else
          pending.insert(clientId)
     RELEASE:
        if notempty(pending)  send(reply[pending.remove( )])
        else free := true
    }
}
```

Resource Allocation with Message Passing Client-side code

```
Client (i) {
    send request (i, ACQUIRE)
    receive reply[i]( )   // blocks awaiting permission
    send request (i, RELEASE)  // no block here
}
```

Duality of Monitors/Message Passing

Monitors	Message Passing
Monitor variables	Local vars on server
Entry (implicit mutex)	**Blocking recv on server**
Procedures in monitor	Arms of switch stmt
Procedure call	**Client sends request to server; may block awaiting reply**
Procedure return	Server sends result to proper client if necessary
Wait	Insert request on server Q
Signal	Remove & process request from server queue

Resource Allocation with Monitors

```
monitor ResourceAllocator
  int free = true; cond c
  acquire( ):       if (free) free = false
                    else wait(c)
  release( ):       if (empty(c)) free = true
                    else signal(c)
end ResourceAllocator
```

Client calls **ResourceAllocator.acquire( )/ResourceAllocator.release( )**

Resource Allocation with Message Passing Server-side code

```
enum reqType := {ACQUIRE, RELEASE}
chan request(int clientId, reqType which)
chan reply[n]( )   // one entry per client
Allocator ( )        // runs on server
   queue pending;  // initially empty
   int clientId;  bool free := true
   reqType which;
```

Resource Allocation with Message Passing
Server-side code, continued

```
while (1) {
    receive(request, clientId, which)
    switch(which) {
     ACQUIRE:
        if (free)
         free := false; send(reply[clientId])
        else
          pending.insert(clientId)
     RELEASE:
        if notempty(pending)  send(reply[pending.remove( )])
        else free := true
     }
}
```

Resource Allocation with Message Passing
Client-side code

```
Client (i) {
    send request (i, ACQUIRE)
    receive reply[i]( )   // blocks
    send request (i, RELEASE)  // no block here
}
```

Programming Client/Server Applications (General Outline)

Outline of Client code

```
while (1)   {
  build request
  send(request, server)
  receive(reply)
  do something
}
```

Outline of Server code

```
while (1)  {
  receive(request)
  switch(request)
  case type1:
     send(client, reply1)
  case type2:
     send(client, reply2)
  etc.
}
```