Message Passing

- Different from shared memory programming
  - no shared memory
  - can't use simple semaphores, condition vars
  - can't use shared buffers, producer/consumer
- Communication based on *message passing*
  - Process A on machine 1 sends message to process B on machine 2 (over the network)
  - How does it get there? [we will ignore this]

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 messages, wrong ordering, overflow, etc.
  - **Channel** -- abstraction of point to point, reliable communication link

Send and Receive

- Send(channel, exprs)
  - Send a message containing *exprs* on the channel indicated
  - The *exprs* can be r-vals
- Receive(channel, args)
  - Receive a message from the channel indicated into *args*
  - Blocks until data is available on channel (can be relaxed in real implementations)
  - Args must be l-vals (must provide a storage location)

Send and Receive

- Notes:
  - Channel handles reliability
    - Must be implemented by network protocols
  - Access to channel is atomic
  - Message has to be buffered if data 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)

Send and Receive

- Notes, continued:
  - Important special case: both *exprs* and *args* are the empty set; in this case:
    - Send is equivalent to V(s)
    - Receive is equivalent to P(s)
    - Number of pending messages is equivalent to the value of *s*

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

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 appropriate client
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( )**

Picture of Resource Allocation with Message Passing

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 {ACQUIRE, RELEASE);
```

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.
}
```