# Message Passing

Reading: Chapter 10 of Magee and Kramer



## The big picture

- Concurrency and Parallelism
  - Requires communication between threads / processes
  - Common modes of communication
    - Shared memory
    - Message passing
- In this module
  - Concurrent programming using Java
    - Shared memory communication across threads
  - Parallel programming using MPI
    - Message passing communication across processes



## The lingering question

- Does this mean?
  - Concurrent programming necessarily involves shared memory communication across threads?
- In this lecture
  - Bridge between concurrent and parallel programming
  - Showing the possible programming of message passing abstractions on top of Java's shared memory concurrency.
- Remaining question
  - Shared memory parallel programming is also possible e.g. just do a Google search on OpenMP.



## The big difference

**Shared Memory**

- Shared variables accessed by diff. threads / processes.
- Synchronization mechanisms (locks) needed for safe access of the shared address space.

**Message Passing**

- Clean separation of address space between threads / processes.
- No such mechanisms are needed per –se
  - The passing of messages can be to a shared message buffer. However, there exist other mechanisms to ensure safe access.



## The different styles of message passing


Called synchronous message passing, studied in Promela modeling e.g.

Sender and receiver processes are fixed.
Point – to – point communication.

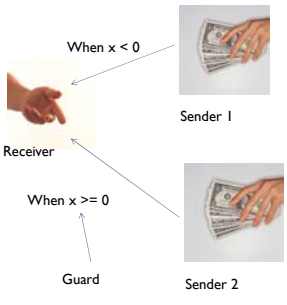
No message buffer is needed.

Sender must block if receiver is not ready.

*Question for class*: Are the sender and receiver symmetric in this case?



## The different styles of message passing




**Selective Message Passing**

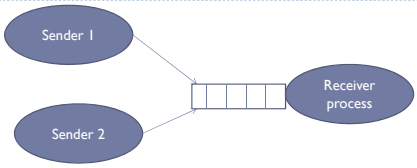
Each point-to-point communication can still be synchronous.

Easily modeled e.g. in Promela – we saw guarded if statements before.

How to achieve this in a programming language e.g. in Java?

Involves a selective synchronization which may not be easy to implement - Guards can after all be empty !!

## The different styles of message passing




**Asynchronous message passing** - common in real-life.

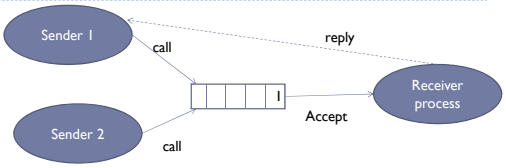
Sends are non-blocking (as long as there is space in the buffer!)

Each process maintains such a buffer to which messages sent from all other processes can be stored.

*Question for class:* Any idea for a selective style receive in asynchronous comm. ?



## The different styles of message passing




Called **Rendezvous.**

Senders send requests which get stored, meanwhile sender blocks.

These are eventually accepted at receiver end.

Once processing of request is completed – receiver sends a reply to sender in question.



## Message Passing

Concepts: synchronous message passing - channel
asynchronous message passing - port
- send and receive / selective receive
rendezvous bidirectional comms - entry
- call and accept ... reply

Models: channel : relabelling, choice & guards
port : message queue, choice & guards
entry : port & channel

Practice: distributed computing (disjoint memory)
threads and monitors (shared memory)



## 1. Synchronous message passing


Called synchronous message passing, studied in Promela modeling e.g.

Sender and receiver processes are fixed.
Point – to – point communication.

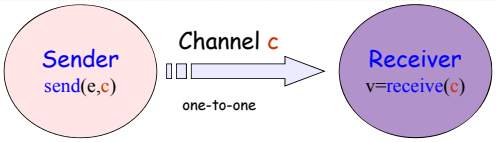
No message buffer is needed.

Sender must block if receiver is not ready.

*Question for class:* Are the sender and receiver symmetric in this case?



## Synchronous message passing - channel




♦ send(e,c) - send the value of the expression e to channel c. The process calling the send operation is blocked until the message is received from the channel.

♦ v = receive(c) - receive a value into local variable v from channel c. The process calling the receive operation is blocked waiting until a message is sent to the channel.

cf. distributed assignment v = e



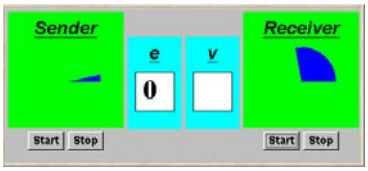
## Basic Idea

- Implement a separate channel object
  - Simulates a buffer of zero capacity between sender and receiver.
  - If we simply implement the channel simply as a monitor – it will not work!
    - Send and receive will not happen in one step.
    - Need to make the sender and receiver handshake, even while calling channel.send and channel.receive methods.
    - How to do so?

## Synchronous message passing - applet

A sender communicates with a receiver using a single channel.

The sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.




```
Channel<Integer> chan = new Channel<Integer>();
tx.start(new Sender(chan,senddisp));
rx.start(new Receiver(chan,recvdisp));
```

Instances of ThreadPanel

Instances of SlotCanvas



## Java implementation - channel

```
public class Channel<T>  extends Selectable {
T chan_ = null;

  public synchronized void send(T v)
                  throws InterruptedException {
        chan_ = v;
   notify();
   while (chan_ != null) wait();
  }

  public synchronized T receive()
                  throws InterruptedException {
        block(); clearReady();     //part of Selectable
        T tmp = chan_; chan_ = null;
        notify();
        return(tmp);
  }
}
```

*This is a simplification of the actual code.*

The implementation of Channel is a monitor that has synchronized access methods for send and receive.

Selectable is described later.



## Java implementation - sender

```
class Sender implements Runnable {
  private Channel<Integer> chan;
  private SlotCanvas display;
  Sender(Channel<Integer> c, SlotCanvas d)
    {chan=c; display=d;}

  public void run() {
    try { int ei = 0;
                  while(true) {
                    display.enter(String.valueOf(ei));
                    ThreadPanel.rotate(12);
                    chan.send(new Integer(ei));
                    display.leave(String.valueOf(ei));
                    ei=(ei+1)%10;ThreadPanel.rotate(348);
                  }
    } catch (InterruptedException e){}
  }
}
```



## Java implementation - receiver

```
class Receiver implements Runnable {
  private Channel<Integer> chan;
  private SlotCanvas display;
  Receiver(Channel<Integer> c, SlotCanvas d)
    {chan=c; display=d;}

  public void run() {
    try { Integer v=null;
          while(true) {
            ThreadPanel.rotate(180);
            if (v!=null) display.leave(v.toString());
            v = chan.receive();
            display.enter(v.toString());
            ThreadPanel.rotate(180);
          }
    } catch (InterruptedException e){}
  }
}
```



## Important issue in message passing

- Message passing assumes that the memory space of the different processes are disjoint.
  - If channels become shared objects accessed by sender and receiver – there still exists the possibility of two processes modifying a shared memory location – data races etc !
  - To avoid data races ---
    - Adopt the practice that the receivers never modify a shared channel object received – they only read from them.
      - Q. to class: How is the read-write data race avoided here?
    - Sender should either not access the sent object, or if it needs it, create a copy of the object before sending it.



## A Model

```
range M = 0..9                  // messages with values up to 9

SENDER = SENDER[0], // shared channel chan
SENDER[e:M] = (chan.send[e]-> SENDER[(e+1)%10]).

RECEIVER = (chan.receive[v:M]-> RECEIVER).

                                // relabeling to model synchronization
||SyncMsg = (SENDER || RECEIVER)
                        /{chan/chan.{send,receive}}.
```

LTS?

How can this be modeled directly without the need for relabeling?

| message operation | Process Equation |
|---|---|
| send(e,chan) | chan.[e] |
| v = receive(chan) | chan.[v:M] |

## A slightly different Model

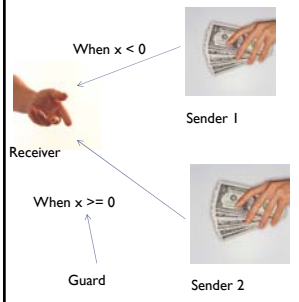
```
range M = 0..9                  // messages with values up to 9

SENDER = SENDER[0], // shared channel chan
SENDER[e:M] = (chan.[e]-> SENDER[(e+1)%10]).

RECEIVER = (chan.[v:M]-> RECEIVER).

                                // relabeling to model synchronization
||SyncMsg = (SENDER || RECEIVER)
```

The send-receive is being captured through shared actions.

The sender and the receiver are still not fully symmetric – value passing dictates this aymmetry.



## 2. Selective Message Passing




**Selective Message Passing**

Each point-to-point communication can still be synchronous.

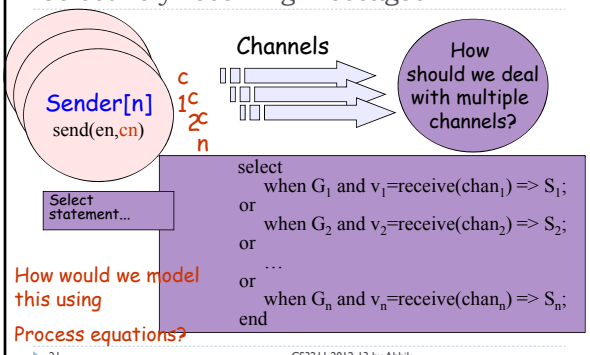
Easily modeled e.g. in Promela – we saw guarded if statements before.

How to achieve this in a programming language e.g. in Java?

Involves a selective synchronization which may not be easy to implement - Guards can after all be empty !!

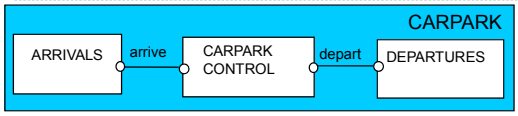


## Selectively receiving messages






## Selective Receive




```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]
                 ).
ARRIVALS   = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).
||CARPARK = (ARRIVALS||CARPARKCONTROL(4)
                      ||DEPARTURES).
```

Interpret as channels

Implementation using message passing?



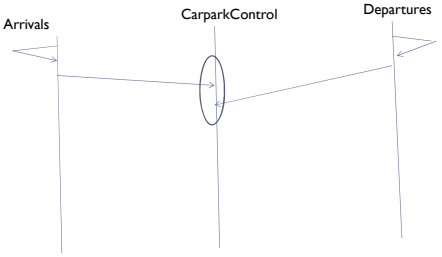
## Implementation

- Previously viewed Carparkcontrol as a Monitor.
- For message passing implementations
  - View it as a separate thread/process
    - It receives messages from arrival and departure processes.
- Schematic code
  - MsgCarPark
    - While (1){
    - **select**
      - when spaces>0 && receive(arrive) -> spaces++;
        when spaces < N && receive (depart) -> spaces--;

      end
    }



## Behaviors of the implementation




How to select among the incoming messages?

Suppose there is no message buffer to store the incoming messages.

## Java implementation – selective receive

```
class MsgCarPark implements Runnable {
  private Channel<Signal> arrive,depart;
  private int spaces,N;
  private StringCanvas disp;

  public MsgCarPark(Channel<Signal> a,
                    Channel<Signal> l,
                StringCanvas d,int capacity) {
    depart=l; arrive=a; N=spaces=capacity; disp=d;
  }
  …
  public void run() {…}
}
```

Implement CARPARKCONTROL as a thread MsgCarPark which receives signals from channels arrive and depart.



## Java implementation – selective receive

```
public void run() {
  try {
    Select sel = new Select();
    sel.add(depart);
    sel.add(arrive);
    while(true) {
      ThreadPanel.rotate(12);
      arrive.guard(spaces>0);
      depart.guard(spaces<N);
      switch (sel.choose()) {
      case 1:depart.receive();display(++spaces);
           break;
      case 2:arrive.receive();display(--spaces);
           break;
      }
    }
  } catch InterruptedException{}
}
```



## Selective receive

```
switch (sel.choose()) {
case 1:depart.receive();display(++spaces);
      break;
case 2:arrive.receive();display(--spaces);
      break;
}
```

sel.add(depart);
sel.add(arrive);

*Depart and arrive are selectable channel objects*

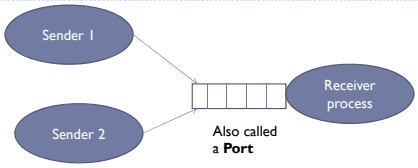
A selectable channel object is ready when a send has been performed.

When the receive on the chosen selectable object is executed – it must go ahead since send has already been performed.

If no send operations have been performed (among the choices) --- the entire choose operation will block, until a matching send has been performed on either the depart or the arrive channels (the two selectable objects in the preceding code).



## 3. Asynchronous message passing




**Many to one communication** - common in real-life.

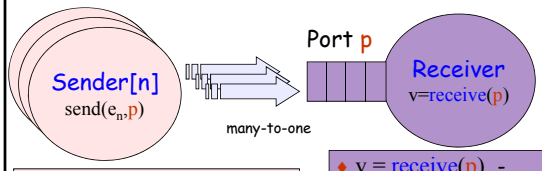
Sends are non-blocking (as long as there is space in the buffer!)

Each process maintains such a buffer to which messages sent from all other processes can be stored.

*Question for class:* Any idea for a selective style receive in asynchronous comm. ?



## Asynchronous Message passing - ports




♦ send(e,p) - send the value of the expression e to port p. The process calling the send operation is not blocked. The message is queued at the port if the receiver is not waiting.

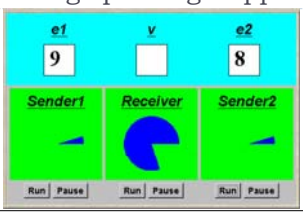
♦ v = receive(p) - receive a value into local variable v from port p. The process calling the receive operation is blocked if there are no messages queued to the port.



## Asynchronous message passing - applet

Two senders communicate with a receiver via an "unbounded" port.

Each sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.




Receiver gets a merger of the 2 data streams.

```
Port<Integer> port = new Port<Integer> ();
tx1.start(new Asender(port,send1disp));
tx2.start(new Asender(port,send2disp));
rx.start(new Areceiver(port,recvdisp));
```

Instances of ThreadPanel

## Java implementation - port

```
class Port<T> extends Selectable {

 Queue<T> queue = new LinkedList<T>();

  public synchronized void send(T v){
        queue.add(v);
    signal();
  }
  public synchronized T receive()
                throws InterruptedException {
        block(); clearReady();
        return queue.remove();
  }
}
```

The implementation of Port is a monitor that has synchronized access methods for send and receive.

```
port.send(new integer(ei));
```
*class Asender*

```
v = port.receive();
```
*class Areceiver*



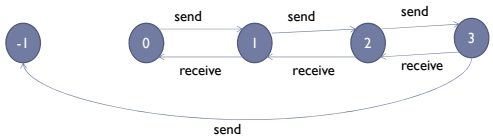
## Model for Ports

```
range M = 0..9            // messages with values up to 9
set   S = {[M],[M][M]}    // queue of up to three messages

PORT                      //empty state, only send permitted
  = (send[x:M]->PORT[x]),
PORT[h:M]                 //one message queued to port
  = (send[x:M]->PORT[x][h]
    |receive[h]->PORT
    ),
PORT[t:S][h:M]                  //two or more  messages queued to port
  = (send[x:M]->PORT[x][t][h]
    |receive[h]->PORT[t]
    ).

// minimise to see result of abstracting from data values
||APORT = PORT/{send/send[M],receive/receive[M]}.
```

LTS?
What happens if send 4 values?



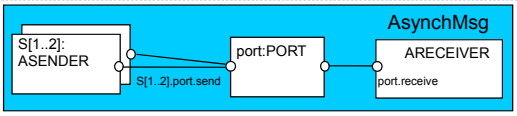
## (Simplified) State machine for Port




Values 0..9 being stored in the Port are abstracted away to reduce the number of states in the state machine.



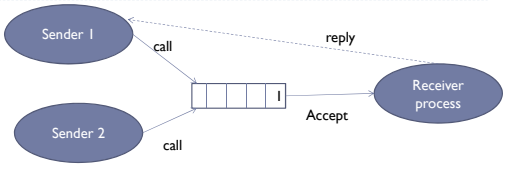
## Model for Applets




```
ASENDER = ASENDER[0],
ASENDER[e:M] = (port.send[e]->ASENDER[(e+1)%10]).

ARECEIVER = (port.receive[v:M]->ARECEIVER).

||AsyncMsg = (s[1..2]:ASENDER || ARECEIVER||port:PORT)
          /{s[1..2].port.send/port.send}.
```



## 4. Rendezvous




Called **Rendezvous, popularized by Ada programming language.**

Senders send requests which get stored, meanwhile sender blocks.

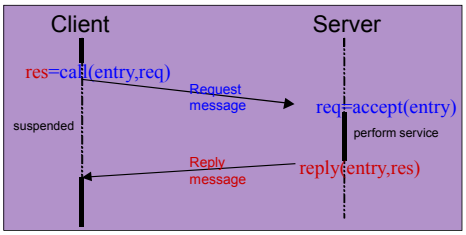
These are eventually accepted at receiver end.

Once processing of request is completed – receiver sends a reply to sender in question.



## Rendezvous - entry

Rendezvous is a form of request-reply to support client server communication. Many clients may request service, but only one is serviced at a time.



## Rendezvous

♦ res=call(e,req) - send the value req as a request message which is queued to the entry e.

♦ The calling process is blocked until a reply message is received into the local variable req.

♦ req=accept(e) - receive the value of the request message from the entry e into local variable req. The calling process is blocked if there are no messages queued to the entry.

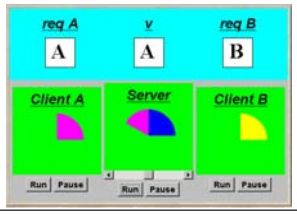
♦ reply(e,res) - send the value res as a reply message to entry e.

The model and implementation use a port for one direction and a channel for the other. Which is which?



## Rendezvous - applet

Two clients call a server which services a request at a time.




```
Entry<String,String> entry = new Entry<String,String> ();
clA.start(new Client(entry,clientAdisp,"A"));
clB.start(new Client(entry,clientBdisp,"B"));
sv.start(new Server(entry,serverdisp));
```

Instances of ThreadPanel

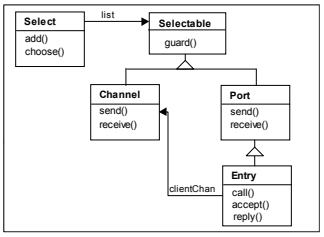
Instances of SlotCanvas



## Java implementation - entry

Entries are implemented as extensions of ports, thereby supporting queuing and selective receipt.

The call method creates a channel object on which to receive the reply message. It constructs and sends to the entry a message consisting of a reference to this channel and a reference to the req object. It then awaits the reply on the channel.




The accept method keeps a copy of the channel reference; the reply method sends the reply message to this channel.



## Java implementation - entry

```
class Entry<R,P> extends Port<R> {
 private CallMsg<R,P> cm;
 private Port<CallMsg<R,P>> cp = new Port<CallMsg<R,P>>();
 public P call(R req) throws InterruptedException {
  Channel<P> clientChan = new Channel<P>();
  cp.send(new CallMsg<R,P>(req,clientChan));
  return clientChan.receive();
 }
 public R accept() throws InterruptedException {
  cm = cp.receive();
  return cm.request;
 }
 public void reply(P res) throws InterruptedException {
  cm.replychan.send(res);
 }
 private class CallMsg<R,P> {
  R request;
  Channel<P> replychan;
  CallMsg(R m, Channel<P> c)
   {request=m; replychan=c;}
 } }
```

Do call, accept and reply need to be synchronized methods?

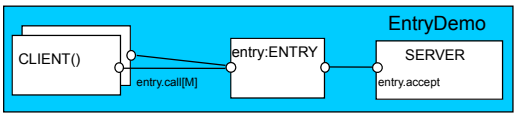
## Answer

- call, accept and reply are not synchronized methods.
  - Client and Server do not share any variables in Entry.
    - cm is only accessed by server for example.
  - Communication among client and server via Port and Channel
    - These communication are thread safe, because
      - Accesses by different processes are time separated, inherently by the rendezvous communication scheme.



## Model of entry and applet

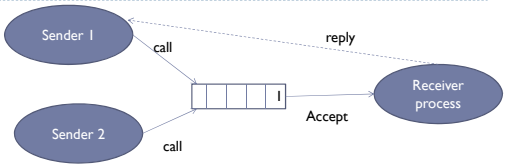
We reuse the models for ports and channels ...




```
set M = {replyA,replyB}          // reply channels
||ENTRY = PORT/{call/send, accept/receive}.
CLIENT(CH='reply) = (entry.call[CH]->[CH]->CLIENT).
SERVER = (entry.accept[ch:M]->[ch]->SERVER).
||EntryDemo = (CLIENT('replyA)||CLIENT('replyB)
          || entry:ENTRY || SERVER ).
```

Action labels used in expressions or as parameter values are prefixed with a single quote.

## Reflections: Rendezvous




Essentially seeking service from a single server process, which manages the shared data structure. Any problem encoded as monitor could be modeled this way too.

Producer Consumer Problem
- Monitor implementation
  - Processes: Producer, Consumer  Monitor: Buffer
- Rendezvous implementation
  - Senders/Clients: Producer, Consumer
  - Receiver: The server process which encapsulates buffer state.



## Rendezvous vs. Invoking Monitor Method

**What is the difference?**
- … from the point of view of the client?
- … from the point of view of the server?
- … mutual exclusion?

**Which implementation is more efficient?**
- … in a local context (client and server in same computer)?
- … in a distributed context (in different computers)?



## Rendezvous vs. Monitor method call

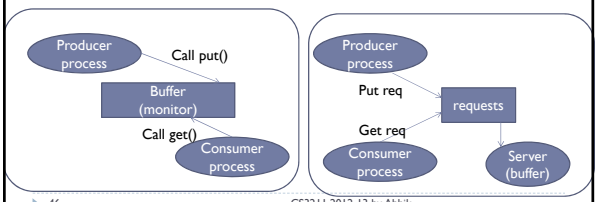
- Bounded buffer was earlier implemented as Monitor.
- As Rendezvous – see pseudocode

```
BoundedBuffer
        entry put, get;
        int count = 0;

        while (1){
            select
                when (count<N)  && obj = accept(put)
                        ->  count++;  // insert obj here
                            reply(put, …)
                 when (count > 0) && accept(get)
                        -> count --;  // retrieve obj here
                            reply(get, …)
            end
    }
```



## Rendezvous vs. Monitor : Correctness

- Server code is shown in previous slide.
  - Issue of mutual exclusion is neatly avoided.
  - Producers and consumer send their requests to server.
  - Buffer state is encapsulated inside server, and modified by server process only!






## Rendezvous vs. Monitor: Performance

- Client and server in same computer
  - Rendezvous involves 2 context switches.
  - Monitor may involve no context switch [can be more efficient]
    - e.g. get from non-empty buffer
- Client and server in different computers [distributed]
  - Rendezvous may be more efficient.
  - Implementing monitors then involves
    - Transfer client's monitor method invocation via Remote Method Invocation.
    - Create new thread in the server computer to call the monitor method on behalf of the client [inefficient].



## Summary of the discussions

- Concepts
  - synchronous message passing - channel
  - asynchronous message passing - port
    - send and receive / selective receive
  - rendezvous bidirectional comms - entry
    - call and accept … reply
- Models
  - channel : relabelling, choice & guards
  - port : message queue, choice & guards
  - entry : port & channel
- Practice
  - distributed computing (disjoint memory)
  - threads and monitors (shared memory)