Chapter 10

Message Passing
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)
10.1 Synchronous Message Passing - channel

- **Sender**
  - `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.

- **Channel c**

- **Receiver**
  - `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`
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.

```java
Channel<Integer> chan = new Channel<Integer>();
Tx.start(new Sender(chan, senddisp));
Rx.start(new Receiver(chan, recvdisp));
```

Instances of **SlotCanvas**

Instances of **ThreadPanel**

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

```
public class Channel<T> extends Selectable {

    T chan_ = null;

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

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

Selectable is described later.
class Sender implements Runnable {
    private Channel<Integer> chan;
    private SlotCanvas display;
    Sender(Channel<Integer> c, SlotCanvas d)
        {chan=c; display=d;}

    public void run() {
        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);
        }
    }
}

Java implementation - sender
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) {}
How can this be modeled directly without the need for relabeling?

<table>
<thead>
<tr>
<th>message operation</th>
<th>FSP model</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{send(e,chan)}</td>
<td>\texttt{chan.[e]}</td>
</tr>
<tr>
<td>\texttt{v = receive(chan)}</td>
<td>\texttt{chan.[v:M]}</td>
</tr>
</tbody>
</table>

Concurrency: message passing
How should we deal with multiple channels?

Select statement...

```
select
  when G_1 and v_1 = receive(chan_1) => S_1;
  or
  when G_2 and v_2 = receive(chan_2) => S_2;
  or
  ...
  or
  when G_n and v_n = receive(chan_n) => S_n;
end
```
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).

Concurrency: message passing
Java implementation - selective receive

```java
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.
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){}
}
10.2 Asynchronous Message Passing - *port*

- **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.

Instances of SlotCanvas

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

Concurrency: message passing
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 model

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

PORT
= (send[x:M]→PORT[x]), //empty state, only send permitted
PORT[h:M]
= (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]}.
model of applet

\[
\begin{align*}
\text{ASENDER} &= \text{ASENDER}[0], \\
\text{ASENDER}[e:M] &= (\text{port}.\text{send}[e] \rightarrow \text{ASENDER}[(e+1)\%10]). \\
\text{ARECEIVER} &= (\text{port}.\text{receive}[v:M] \rightarrow \text{ARECEIVER}). \\\n\mid | \text{AsyncMsg} &= (s[1..2]:\text{ASENDER} \mid \mid \text{ARECEIVER} \mid \text{port}:\text{PORT}) \\
\text{ASYNCH_MSG} &= \{s[1..2].\text{port}.\text{send}/\text{port}.\text{send}\}. \\
\end{align*}
\]

Safety?
10.3 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.

![Diagram showing Rendezvous: entry](image)
Rendezvous

- \( \text{res}=\text{call}(e, \text{req}) \) - send the value \( \text{req} \) as a request message which is queued to the entry \( e \).
- The calling process is \textit{blocked} until a reply message is received into the local variable \( \text{req} \).
- \( \text{req}=\text{accept}(e) \) - receive the value of the request message from the entry \( e \) into local variable \( \text{req} \). The calling process is \textit{blocked} if there are no messages queued to the entry.
- \( \text{reply}(e, \text{res}) \) - send the value \( \text{res} \) 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?
Two clients call a server which services a request at a time.

```java
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`

Concurrency: message passing

Instances of `SlotCanvas`
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.
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?
model of entry and applet

We reuse the models for ports and channels ...

```plaintext
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 ).
```

Concurrency: message passing
rendezvous Vs monitor method invocation

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)?
Summary

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)
Course Outline

- Processes and Threads
- Concurrent Execution
- Shared Objects & Interference
- Monitors & Condition Synchronization
- Deadlock
- Safety and Liveness Properties
- Model-based Design

- Dynamic systems
- Concurrent Software Architectures
- Message Passing
- Timed Systems

Concurrency: message passing