Chapter 10

Message Passing

Concurrent/Message passing

Concepts:
- Synchronous message passing - channel
  - send and receive / selective receive
  - send and receive / rendezvous
  - bidirectional communications
  - entry
- Asynchronous message passing - port
  - send and receive
  - selective receive
- Model:
  - 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

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

Synchronous message passing - applet

Instances of ThreadPanel

Instances of SlotCanvas

cf. distributed assignment $v = e$
Java implementation - channel

```java
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);
    }
}
```

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

Selectable is described later.

Java implementation - sender

```java
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

```java
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){}
    }
}
```

model

```java
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).
```

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>send(e,chan)</td>
<td>chan[e]</td>
</tr>
<tr>
<td>v = receive(chan)</td>
<td>chan[v:M]</td>
</tr>
</tbody>
</table>

Concurrence: message passing
Concurrent message passing

Channels

Sender[n] send(en, cn)

How should we deal with multiple channels?

Select statement...

Arrivals

Carpark

Departures

Select statement...

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.
    }
}
```

Java implementation - selective receive

```java
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 e) {
    }
}
```

Interpret as channels

Implementation using message passing?

See Applet
10.2 Asynchronous Message Passing - 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 ThreadPanel

Instances of SlotCanvas

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();
    }
}
```

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]),
PORT = (send[x:M] -> PORT[x][h],
receive[h] -> PORT )
PORT = (send[x:M] -> PORT[x][t][h],
receive[h] -> PORT[t] )

// minimise to see result of abstracting from data values

\[ APORT = PORT/\{send/send[M],receive/receive[M]\} \]
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.

\[ res = \text{call}(entry, req) \]

\[ req = \text{accept}(e) \] - 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?
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.

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

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 ).

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