Concurrency

10 - Message Passing

Concurrent communication

Alexandre David
adavid@cs.aau.dk

Credits for the slides:
Jeff Magee & Jeff Kramer
### Message Passing

**Concepts:**
- **synchronous** message passing (via channels)
- **asynchronous** message passing (via ports)
  - send and receive / selective receive
- **rendezvous** bidirectional communications - entry
  - call and accept ... reply

**Models:**
- **channel**: re-labelling, 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

- **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* \( v = e \)
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 chan = new Channel();
tx.start(new Sender(chan, senddisp));
rx.start(new Receiver(chan, recvdisp));

Instances of **ThreadPanel**

Instances of **SlotCanvas**

Concurrency: message passing
class Select {

    Vector list = new Vector(2);

    public void add(Selectable s) {
        // add s to list, set s select to this
    }

    public synchronized int choose() throws .. {
        // block until one selectable is
        // ready with a satisfied guard
    }
}
class Selectable {
    private boolean open = false;
    private int ready = 0;
    private Select inchoice = null;
    private boolean guard_ = true;

    void setSelect(Select s) { inchoice = s; }
    void guard(boolean g) { guard_ = g; }
    void clearReady() { --ready; }

    synchronized void block throws .. { 
        // set open, wait until ready, clear open
    }

    synchronized void signal() { 
        ++ready; if (open) inchoice.notifyAll();
    }
}
class Channel extends Selectable {
    Object chann = null;
    
    public synchronized void send(Object v)
        throws InterruptedException {
        chann = v;
        signal();
        while (chann != null) wait();
    }
    
    public synchronized Object receive()
        throws InterruptedException {
        block(); clearReady();  //part of Selectable
        Object tmp = chann; chann = null;
        notifyAll();  //could be notify()
        return(tmp);
    }
}
```
public class Sender implements Runnable {
    private Channel chan;
    private SlotCanvas display;
    Sender(Channel 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) {};
    }
}
```
class Receiver implements Runnable {
    private Channel chan;
    private SlotCanvas display;
    Receiver(Channel 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 = (Integer)chan.receive();
                display.enter(v.toString());
                ThreadPanel.rotate(180);
            }
        }
        catch (InterruptedException e){}
    }
}
Model

range \( M = 0..9 \)  // messages with values up to 9

\[
\text{SENDER} = \text{SENDER}[0],  \quad  // \text{shared channel chan} \\
\text{SENDER}[e:M] = (\text{chan}.\text{send}[e] \rightarrow \text{SENDER}[(e+1)\%10]).
\]

\[
\text{RECEIVER} = (\text{chan}.\text{receive}[v:M] \rightarrow \text{RECEIVER}).
\]

// relabeling to model synchronization

\[
\text{||SyncMsg} = (\text{SENDER} \parallel \text{RECEIVER})  \\
\quad \{\text{chan}/\text{chan}.\{\text{send, receive}\}\}.
\]

How can this be modelled 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>?</td>
</tr>
<tr>
<td>( v = \text{receive}(\text{chan}) )</td>
<td>?</td>
</tr>
</tbody>
</table>
Selective Receive

Sender[n] send(en, cn)

Channels

select
  when \( G_1 \) and \( v_1 = \text{receive}(chan_1) \) => \( S_1 \);
  or
  when \( G_2 \) and \( v_2 = \text{receive}(chan_2) \) => \( S_2 \);
  or
  when \( G_n \) and \( v_n = \text{receive}(chan_n) \) => \( S_n \);
end

How would we model this in FSP?

Concurrency: message passing
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).

Implementation using message passing?
Java Implementation - Selective Receive

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

    public MsgCarPark(Channel a, Channel 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{})
}

See Applet
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.
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

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

Concurrency: message passing
class Port extends Selectable {
    Vector queue = new Vector();

    public synchronized void send(Object v){
        queue.addElement(v);
        signal();
    }

    public synchronized Object receive() throws InterruptedException {
        block(); clearReady();
        Object tmp = queue.elementAt(0);
        queue.removeElementAt(0);
        return(tmp);
    }
}

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 //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]}.  

Concurreny: message passing
Model of Applet

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

- **Client**:
  - `res = call(entry, req)`
  - `suspended`

- **Server**:
  - `req = accept(entry)`
  - `perform service`
  - `reply(entry, res)`
Rendezvous

- $res = \text{call}(e, req)$ - send the value $req$ 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 $req$.

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

- $\text{reply}(e, res)$ - send the value $res$ as a reply message to entry $e$. 

Concurrency: message passing

©Magee/Kramer
Asynchronous Message Passing - Applet

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

Entry entry = new Entry();
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.
public class Entry extends Port {
    private CallMsg cm;

    public Object call(Object req) throws InterruptedException {
        Channel clientChan = new Channel();
        send(new CallMsg(req, clientChan));
        return clientChan.receive();
    }

    public Object accept() throws InterruptedException {
        cm = (CallMsg) receive();
        return cm.request;
    }

    public void reply(Object res) throws InterruptedException {
        cm.replychan.send(res);
    }

    private class CallMsg {
        Object request; Channel replychan;
        CallMsg(Object m, Channel c) {
            request = m; replychan = c;
        }
    }
}
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).
```

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 : re-labelling, choice & guards
- port : message queue, choice & guards
- entry : port & channel

◆ Practice
- distributed computing (disjoint memory)
- threads and monitors (shared memory)