3C03 Concurrency: Message Passing

Mark Handley

Outline

- Motivation

- Synchronous Message Passing
  - Modelling Synchronous Message Passing in FSP
  - Selective Receive

- Asynchronous Message Passing
  - Modelling Asynchronous Message Passing in FSP

- Rendezvous in Java
Absence of Shared Memory

In previous lectures interaction between threads has been via shared memory
- In Java, we refer to shared objects.
- Usually encapsulate shared memory in Monitors.

In a distributed setting there is no shared memory.
- Communication is achieved via passing messages between concurrent threads.
- Same message passing abstraction can also be used in non-distributed settings.

Forms of Message Passing

- Principal Operations
  - send
  - receive

- Synchronization
  - Synchronous
  - Asynchronous
  - Rendezvous

- Multiplicity
  - one-one
  - many-one
  - many-many

- Anonymity
  - anonymous message passing
  - non-anonymous message passing

- Receipt of Messages
  - Unconditional
  - Selective
Synchronous Message Passing

- **send(e,c)**
  - Send e to channel c. Sending process is blocked until channel received e

- **v=receive(c)**
  - Receive a value into a local variable v from channel c. The calling process is blocked until a message is sent into channel

Channel has no buffering.

Synchronous Message Passing in Java

Java has no built in message passing primitives
- Unlike Occam or Ada.

Can still do message passing in Java, but it’s clunky:
- Encapsulate message passing abstractions in monitor

Channel:

```java
class Channel extends Selectable {
    public synchronized void send (Object v)
        throws InterruptedException{
        ...
    }
    public synchronized Object receive() {
        ...
    }
}
```

Demo
Modelling Sync. Message Passing

range M = 0..9
SENDER = SENDER[0],
SENDER[e:M] = (chan.send[e] -> SENDER[(e+1)%10]).
RECEIVER = (chan.receive[v:M] -> RECEIVER).
||SYNCMSG = (SENDER || RECEIVER)
/{chan/chan.(send,receive)}.

To avoid re-labelling:
range M = 0..9
SENDER = SENDER[0],
SENDER[e:M] = (chan[e] -> SENDER[(e+1)%10]).
RECEIVER = (chan[v:M] -> RECEIVER).
||SYNCMSG = (SENDER || RECEIVER).

Selective Receives

What happens if a process wants to receive from more than one channel?
- It blocks if it chooses the wrong channel

Selective receives (e.g. Occam or Ada):

select when G1 and v1=receive(chan1) => S1;
    or when G2 and v2=receive(chan2) => S2;
    ...
    or when G3 and vn=receive(chan) => S3;
end;

- Note similarity to FSP guarded actions
Modelling Selective Receives

Example: Car Park Control

CONTROL (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 | | DEPARTURES | | CONTROL(4) ).

How to implement CONTROL using Message Passing?

Classes for Selective Receive

Selectable
+ guard()

Channel
+ send(o : Object)
+ receive() : Object

Select
+ add(s : Selectable)
+ choose() : int

MsgGate
+ run()

MsgCarPark
+ run()

Runnable
<<implements>>

Demo
Implementing Selective Receives

class MsgGate implements Runnable {
    private Channel chan;
    private Object signal = new Object();
    public MsgGate(Channel c) {chan=c;}
    public void run() {
        try {
            while(true) {
                ThreadPanel.rotate(12);
                chan.send(signal);
                ThreadPanel.rotate(348);
            }
        } catch (InterruptedException e){}
    }
}

Implementing Selective Receives

class MsgCarPark implements Runnable {
    private Channel arrive, leave;
    private int spaces, N;
    public MsgCarPark(Channel a, Channel l, int capacity) {
        leave=l; arrive=a; N=spaces=capacity;
    }
    public void run() {
        try {
            Select sel = new Select();
            sel.add(leave);
            sel.add(arrive);
            while(true) {
                ThreadPanel.rotate(12);
                arrive.guard(spaces>0);
                leave.guard(spaces<N);
                switch (sel.choose()) {
                    case 1:leave.receive(); display(++spaces); break;
                    case 2:arrive.receive(); display(--spaces); break;
                }
            }
        } catch (InterruptedException e){}
    }
}
Asynchronous Message Passing

- **Send does not block**
- Messages are queued at the receiver
- We refer to these queues as ports
- Communication can be many-to-one

 Sender[1]
 send(e1,p)

 Sender[2]
 send(e2,p)

 Sender[n]
 send(en,p)

 Receiver
 v=receive(p)

 Port p

 Asynchronous Message Passing in Java

 Two operations:

 1. **send(e,p)**
    - send value e to port p. Calling process not blocked
 2. **v=receive(p)**
    - receive value into variable v from port p. Calling process is blocked if no value queued to port.

 Implementation of Ports in Java:

 ```java
class Port extends Selectable{
    Vector queue;
    public synchronized void send(Object v) {...}
    public synchronized Object receive() throws InterruptedException {...}
}
```

© Wolfgang Emmerich, Mark Handley 1998 - 2003
**Modelling Asynchronous Message Passing**

\[
\begin{align*}
\text{range } M & = 0..4 \\
\text{set } S & = \{ [M], [M][M] \} \\
\text{PORT} & = (\text{send}[x:M] \to \text{PORT}[x]), \\
\text{PORT}[h:M] & = (\text{send}[x:M] \to \text{PORT}[x][h] \\
& \quad | \text{receive}[h] \to \text{PORT}), \\
\text{PORT}[t:S][h:M] & = (\text{send}[x:M] \to \text{PORT}[x][t][h] \\
& \quad | \text{receive}[h] \to \text{PORT}[t]). \\
\text{ASENDER} & = \text{ASENDER}[0], \\
\text{ASENDER}[e:M] & = (\text{port.send}[e] \to \text{ASENDER}[\text{(e+1)}\%4]). \\
\text{ARECEIVER} & = (\text{port.receive}[v:M] \to \text{ARECEIVER}). \\
\text{ASYNCM} & = \{ \text{port.send[port.send]} \land \text{port:PORT} \land \text{ARECEIVER \land s[1..2].port.send} \}.
\end{align*}
\]

**Rendezvous Message Passing**

*Rendezvous is a Request-Reply form of message passing to support client-server interaction:*

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

- **Server**
  - req = accept(entry)
  - reply(entry, res)

\[ \text{ltsa} \]
Rendezvous

(1) \texttt{res=\texttt{call(e,req)} - send the value \texttt{req} as a request message which is queued to the entry \texttt{e}.}

(4) The calling process is \textit{blocked} until a reply message is received into the local variable \texttt{req}.

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

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

Entry in Java

- \textit{Implemented as a extension of a port.}
- \textit{call()} \textit{creates a channel} on which to receive the reply, and passes a reference to \textit{this} in the message to the server.
- \textit{Server keeps a reference to the reply channel, and later uses it to send the reply back to the client.}
Entry in Java

```java
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;
        }
    }
}
```

Summary

- **Synchronous Message Passing**
  - Modelling Synchronous Message Passing in FSP
  - Selective Receive
- **Asynchronous Message Passing**
  - Modelling Asynchronous Message Passing in FSP
- **Rendezvous**
  - Request/reply message passing.