G52CON:
Concepts of Concurrency
Lecture 15: Message Passing

Gabriela Ochoa
School of Computer Science & IT
gxo@cs.nott.ac.uk
Content

• Introduction and transition
  – Recapitulation on hardware architectures
  – Overview of shared memory synchronisation
  – The need for synchronisation mechanisms that are less centralised (suitable for distributed computing)

• Message passing (one-way communication) using channels
  – Asynchronous communications
    • Example: Producers and consumers paradigm: Filter processes
  – Synchronous communications
Hardware

Single processor

- Primary memory
  - Level 2 cache
    - Level 1 cache
      - CPU

Multiprocessor - shared memory

- Memory...
  - Interconnection network
    - Cache...
      - CPU...

Concurrent programming models exist as an abstraction above hardware and memory architectures

Figures from (Andrews, 2000) Chapter 1

Multicomputer - separate memories (Physically close to each other)

Network o or cluster of workstations
Overview of shared memory synchronisation

• Semaphores and monitors are the most important constructs found in concurrent programming languages
• They are the tools we need to use for shared memory synchronisation
• Both the semaphore and the monitor are highly centralised constructs. They maintain queues of blocked process and encapsulate data
• As multi-computers and distributed architectures become more popular, there is a need for synchronisation constructs that are less synchronised
Message passing

- These synchronisation constructs are based upon communication (message passing), rather than upon sharing.
- Synchronisation is achieved by using communication between sending processes and receiving processes.
- As before, we will be dealing with an abstraction, and not with the underlying implementation.
- The interleaving model will continue to be used.
Message passing

• In message-passing programs, processes interact by sending and receiving messages
• Process that interact by message passing, do not need access to shared memory
• Therefore, mutual exclusion is not an issue in message passing protocols
• Processes can be located in different computers connected by a communication network
• However, message passing is also used when processes are intended to run within a single computer
Models for communication

• Fundamental to message passing are the operations to **send** and **receive** a message

• Two basic models of message passing
  – *Synchronous*: the sender of a message waits until it has been received. **Example**: telephone call
  – *Asynchronous*: the sender does not wait and messages that have been set but not yet received are buffered. **Example**: text messages, or e-mail

• These are both *one-way* forms of communication: the messages are transmitted in one direction only, from sender to receiver

• Two-ways message protocol: rendezvous (next lecture)
Asynchronous message passing

- Provided by communication channels
- Channels are FIFO queues of pending messages
- Accessed by means of two primitives: send and receive
- To initiate a communication, a process send a message to a channel; another process acquires the message by receiving from the channel
- Channels are like semaphores that carry data.
  - send corresponds to $P$
  - receive corresponds to $V$
- Different notations have been proposed for asynchronous message passing, we follow here the notation used in Andrews (2000), Chapter 7
Asynchronous message passing

A channel declaration has the form:
- `chan name(id1: type1; ...; idN: typeN);`

Examples
- `chan input(char)`: used to transmit a single character
- `chan disk_access(int cylinder, int block, int count, char *buffer)`: four fields with names indicating their roles

Arrays of channels can be declared
- `chan result[n] (int);`
Message passing primitives

• **send name**(expr1, ..., exprN)
  – types and number of fields must match with channel declaration
  – **effect**: evaluate the expressions and produce a message \( M \), and atomically append \( M \) to the end of the named channel
  – send is *nonblocking* (asynchronous) (queue is unbounded)

• **receive name**(var1, ..., varN)
  – variables types and number must match with channel declaration
  – **effect**: wait for a message on the named channel, atomically remove first message (at the front of the queue) and put the fields of the message into the variables
  – Receive is a *blocking* primitive since it might cause delay
Asynchronous Message Passing – channel (Port)

- **send** the value of the expressions $e_1, .., e_n$ to channel **ch**. The process calling the send operation is not blocked. The message is queued at the channel if the receiver is not waiting.

- **Receive** a value into local variable $v_1, .., v_n$ from channel **ch**. The process calling the receive operation is blocked if there are no messages queued to the channel.
Asynchronous message passing

- Access to the content of each channel is atomic
- Message delivery is reliable and error-free
- Every message sent to the channel is eventually delivered
- Channels are FIFO queues. So, messages will be received in the order in which they were appended to the channel
- Example:

```c
chan ch(int)

// process A               // process B
send ch(1)                 receive ch(x)
send ch(2)                 receive ch(y)
```

`x` will contain 1 and `y` will contain 2

Order of messages from **SAME** source is the order of the sends
Another simple example

chan ch1(int), ch2(int)

// process A           // process B
send ch1(1)            receive ch1(x)
send ch2(2)            receive ch1(y)

// process C           // process D
send ch1(3)            receive ch2(u)
send ch2(4)            receive ch2(v)

• what is received now? x will get 1 or 3 and y will get 3 or 1 u will get 2 or 4 and v will get 4 or 2
Example 1: filter process to assemble line of characters

- A Filter is a process that receives messages from one or more input channels and send messages to one or more output channels.

- Consider a simple filter process that
  - receives a stream of characters from channel \texttt{input},
  - Assembles the characters into lines, and
  - Sends the resulting lines to channel \texttt{output}.

- Symbolic constants
  - CR: carriage-return character
  - \texttt{MAXLINE}: maximum length of a line
  - EOL: appended to the output to indicate the end of a line
Example 1: filter process to assemble line of characters

chan input(char), output(char [MAXLINE]);

process Char_to_Line {
    char line[MAXLINE]; int i = 0;
    while (true) {
        receive input(line[i]);
        while (line[i] != CR and i < MAXLINE) {
            // line[0:i-1] contains the last i input characters
            i = i+1;
            receive input(line[i]);
        }
        line[i] = EOL;
        send output(line);
        i = 0;
    }
}
More on Channels

• Common terminology
  – Mailbox: a channel that have several process sending and several process receiving;
  – Port: a channel that has exactly one receiver, it may have several senders
  – Link: a channel with just one sender and one receiver

• To determine whether a channel’s queue is currently empty, a process can call the Boolean-valued function
  – empty(ch) : This function is true if a channel ch contains no messages, otherwise, it is false
Example 2: A sorting Network

• A Filter is a process that receives messages from one or more input channels and send messages to one or more output channels
• Consider the problem of sorting a list of $n$ numbers into ascending order
• There are many kinds of sorting networks, just as there are many different internal sorting algorithms
• Merge network: repeatedly and in parallel, merge two sorted lists into a longer sorted list
• The network is constructed out of Merge filters
Example 2: A sorting Network

- Each Merge filter receives values from two ordered input streams \texttt{in1} and \texttt{in2}, and produces and ordered output \texttt{out}.
- The ends of the input stream are marked by a sentinel \texttt{EOS}
- \texttt{Merge} appends \texttt{EOS} and the end of the output stream
- \texttt{Merge} is implemented by repeatedly comparing the next two values received from \texttt{in1} and \texttt{in2} (stored in \texttt{v1} and \texttt{v2}), and sending the smaller to \texttt{out}
- The next slide shows the implementation of the filter process
chan in1(int), in2(int), out(int);

process Merge {
  int v1, v2;
  receive in1(v1);  // get first two input values
  receive in2(v2);
  // send smaller value to output channel and repeat
  while (v1 != EOS and v2 != EOS) {
    if (v1 <= v2)
      { send out(v1); receive in1(v1); }  
    else  // (v2 < v1)
      { send out(v2); receive in2(v2); }  
  }
  // consume the rest of the non-empty input channel
  if (v1 == EOS)
    while (v2 != EOS)
      { send out(v2); receive in2(v2); }  
  else  // (v2 == EOS)
    while (v1 != EOS)
      { send out(v1); receive in1(v1); }  
  // append a sentinel to the output channel
  send out(EOS);
}

A filter process that merges two input streams
A sorting network of Merge processes

- To form a sorting network, we employ a collection of Merge processes and arrays of input and output channels.
- If the number of values $n$ is a power of 2, the resulting communication pattern forms a tree.
- Information in the sorting network flows from left to right.
- Each node at the left is given two input values, which it merges to form a stream of two sorted values. The next node forms streams of four sorted values, and so on.
- The rightmost node produces the final sorted stream.
- The sorting network contains $n - 1$ processes, the width of the network is $\log_2 n$. 
Other uses of asynchronous channels
(Examples discussed in Andrews (2000))

• We covered a **producers and consumers** example (filters): each process is a *filter* that consumes the output of its predecessors and produces outputs for its successors.

• **Clients and servers**: dominant interaction patterns in distributed systems. A client process request a service, and waits for a reply. A server waits for requests from clients, then acts upon them.
  – **Examples**: resource managers, self-scheduling disk servers, file servers.

• **Interacting peers**: It occurs in distributed programs when there are several processes that execute the same code and exchange messages to accomplish a task.
  – **Examples**: scientific computing, matrix multiplication.
SYNCHRONOUS MESSAGE PASSING
Synchronous Message Passing - channel

Sender
sync_send ch(e₁, ..,eₙ)

Channel ch

Receiver
Receive ch(𝑣₁,..,𝑣ₙ)

one-to-one

- sync_end ch(e₁, ..,eₙ) - send the value of the expressions e₁, ..,eₙ to channel ch. The process calling the send operation is blocked until the message is received from the channel.

- Receive ch(𝑣₁,..,𝑣ₙ) - receive a value into local variable v₁,..,vₙ from channel ch. The process calling the receive operation is blocked waiting until a message is sent to the channel.
Synchronous Message Passing in one-way communication channels

Advantages

• There is a bound on the size of communications channels (buffer space)
• A process can have at most one message a time queued up on any channel
• Not until the message is received, can the sending process continue and send another message

Disadvantages

• Concurrency is reduced. When two processes communicate, at least one of them will have to block
• Programs are more prone to deadlock. The programmer has to be careful that all send and receive statements match up.
Asynchronous vs. Synchronous channels

- `send` and `sync_send`, are often interchangeable.
- The main difference between asynchronous and synchronous message passing is the trade-off between having possibly more concurrency, and having bounded communication buffers.
- Since memory is plentiful, and asynchronous send is less prone to deadlock, most programmers prefer it.
Distributed programming and Java

• Java supports concurrent programming by mean of threads, shared variables and synchronized methods
• Java can also be used to write distributed programs
• Java does not contain built-in primitives for message passing
• But it contains a standard package java.net
• Classes in the java.net package support
  – Low-level communications using datagrams
  – Higher-level communication using sockets
  – Internet communication using URLs (Uniform Resource Locations)
Overview and transition

• Message passing is ideally suited for programming filters and interacting peers, because these kinds of processes send information in one direction through communication channels.

• Message passing can also be used to program clients and servers. However, since client-server communications is two ways, this lead to a large number of channels.

• There are other additional programming constructs
  – Remote procedure call (RPC)
  – Rendezvous

• These are two-way communication protocols, ideally suited to programming client/server interactions. (next lecture)