G52CON: Concepts of Concurrency

Lecture 13 Message Passing

Brian Logan
School of Computer Science & IT
bsl@cs.nott.ac.uk

Outline of this lecture

- distributed processing implementations of concurrency
- message passing
  - one way vs two way communication
  - naming schemes
  - asynchronous vs synchronous communication
- client-server examples:
  - active monitors
  - self-scheduling disk driver

Implementations of concurrency

We can distinguish three types of implementations of concurrency:

- **multiprogramming**: execution of concurrent processes by timesharing them on a single processor;
- **multiprocessing**: the execution of concurrent processes by running them on separate processors which all access a shared memory; and
- **distributed processing**: the execution of concurrent processes by running them on separate processors which communicate by message passing.

Distributed processing

- *network architectures*, in which processors share only a communication network are becoming increasingly common: e.g., networks of workstations or multicomputers with distributed memory.
- the processes don't share a common address space, so they can't communicate via shared variables
- instead they communicate by *sending and receiving messages*

Message passing

Processes communicate by sending and receiving messages using special message passing primitives which include synchronisation:

- **`send`** *`destination message:`* sends *`message`* to another process *`destination`*
- **`receive`** *`source message:`* indicates that a process is ready to receive a message *`message`* from another process *`source`*

No special mechanisms are required for mutual exclusion, but new techniques are needed for condition synchronisation.

Approaches to message passing

Many different approaches to message passing have been proposed which differ in:

- whether communication is one way or two way;
- whether the naming of sources and destinations is direct or indirect
- whether the naming of sources and destinations is symmetrical or asymmetrical
- how **`send`** and **`receive`** operations are synchronised.

One way communication

If communication is *one way*

- process can only **`send`** or **`receive`** on a given channel in a single operation
- we need to use two channels to establish two way communication between processes

All the message passing primitives we will look at in this lecture use *one way communication*.

Naming sources and destinations

The naming of the source and destination of messages can be either:

- *direct:* using the names of processes; or
- *indirect:* using the names of channels; and

and

- *symmetrical*: both **`send`** and **`receive`** name processes or channels; or
- *asymmetrical*: only **`send`** names processes or channels and **`receive`** receives from any process or channel

Direct naming

In *direct naming***,** unique names are given to all processes comprising a program

- in *symmetrical direct naming* both the sender and receiver name the corresponding process
- in *asymmetrical direct naming* the receiver can receive messages from any process

Indirect naming

Indirect naming uses intermediaries called channels or mailboxes

- in *symmetrical indirect naming* both the sender and receiver name the corresponding channel:
- in *asymmetrical indirect naming* the receiver can receive messages from any channel:

In this lecture, I will assume we are using *asymmetrical indirect naming.*

Processes and channels

Direct naming	**Indirect naming**
symmetrical direct naming: **`send`** *`process message`* **`receive`** *`process message`*	symmetrical indirect naming: **`send`** *`channel message`* **`receive`** *`channel message`*
asymmetrical direct naming: **`send`** *`process message`* **`receive`** *`message`*	asymmetrical indirect naming: **`send`** *`channel message`* **`receive`** *`message`*

Synchronising **`send`** and **`receive`**

- if a process tries to **`receive`** a message before one has been sent by another process, it will block until there is a message for it to read.
- the differences are mainly in the behaviour of the sending process:
  - asynchronous **`send`** : e.g., Unix sockets, `Java.net`
  - synchronous **`send`** : e.g., CSP, occam
  - remote invocation : e.g., extended rendezvous, RPC

Asynchronous Message Passing

If a process sends a message and continues executing without waiting for the message to be received, then the communication is termed *asynchronous*

- **send** operations are non-blocking:
- a sending process can get arbitrarily far ahead of a receiving process;
- message delivery is not guaranteed if failures can occur; and
- since channels can contain an unbounded number of messages messages have to be buffered.

Problems with asynchronous message passing

- the receiving process cannot know anything about the current state of the sending process
- the sending process has no way of knowing if the message was ever received unless the receiving process sends a reply
- it is hard to detect when failures have occurred
- buffer space is finite–if too many messages are sent either the program will crash, the buffer will overflow with loss of messages, or **send** operation will block

Synchronous message passing

If the sending process is delayed until the corresponding receive is executed, the the message passing is *synchronous*

- both the **send** and **receive** operations are blocking
- a process **send**ing to a channel delays until another process is ready to receive from that channel;
- messages don't need to be buffered.

Problems with synchronous message passing

Synchronous message passing can result in reduced concurrency:

- whenever two processes communicate, at least one of them will have to block (whichever one tries to communicate first).
- concurrency is also reduced in some client-server interactions:
  - when a client is releasing a resource, there is usually no reason for it to wait until the server has received the release message
  - similarly, when a client writes to a device (e.g., a file or graphics display) it can usually continue without waiting for the sever to receive the message.

Example: active monitors 1

```
// globally available names for n channels
channel request;
channel[] reply = new channel[n];

// Resource allocation process
process ResourceAllocator {
    list units = new list[MAXUNITS];
    queue pending;
    integer avail = MAXUNITS;
    integer clientID, unitID;
```


Active monitors 2

```
while (true) {
    receive request <clientID, kind, unitID>;
    if (kind == ACQUIRE) {
        if (avail > 0) {
            avail--;
            remove(units, unitID);
            send reply[clientID] <unitID>
        } else {
            insert(pending, clientID);
    } else { // kind == RELEASE
        // free a unit of resource …
```

Active monitors 3

```
        // continued ...
        } else { // kind == RELEASE
            if (empty(pending)) {
                avail++;
                insert(units, unitID);
            } else {
                remove(pending, clientID);
                send reply[clientID] <unitID>;
            }
        }
    }
}
```


Active monitors 4

```
// ith Client process of n ...
process Client {
    integer unitID;
    send request <i, ACQUIRE, null>;
    receive reply[i] <unitID>;
    // use the resource unitID, then release it …
    send request <i, RELEASE, unitID>;
}
```


Active monitors 5

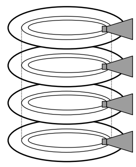
This is one way to program a simple monitor as an active process rather than a passive collection of procedures:

- we get mutual exclusion because only the server process can access its own local variables
- the monitor procedures typically get turned into the branches of an `if` or `switch` statement, so only one 'monitor procedure' can be active at a time, and the monitor procedures run with mutual exclusion.
- condition synchronisation is programmed with normal variables—conditions are re-evaluated on the arrival of a new message.

Example: scheduling disk access

- data is stored in *blocks* which are arranged in concentric *tracks*
- tracks in the same relative position on different platters form a *cylinder*
- the address of a block consists of a cylinder, track and offset from a start point
- data is accessed by positioning a read/write head over the appropriate track and waiting until the required block passes under the head.

A moving head disk



Disk access time

- disk access time depends on:
  - the seek time to move the heads to the appropriate cylinder
  - the rotational delay waiting for the block to pass under the heads
  - the data transmission time
- the time required to move the heads depends on the distance between the two cylinders–seeking even one cylinder takes much longer than the maximum rotational delay
- to reduce the *average* disk access time, we need to minimise head motion and hence seek time.

Scheduling disk requests

The key idea is to service the disk requests *out of order* when two or more clients want to access the disk at about the same time.

Suppose we have a sequence of requests for cylinders:

100, 1, 101, 2, ...

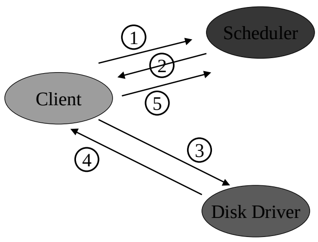
with the heads initially at cylinder 100. If we process these in the order of arrival, the heads travel a total of 300 cylinders.

If we process them in the order:

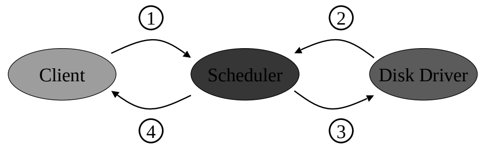
100, 101, 2, 1 ...

the heads travel a total of 101 cylinders. Furthermore the client requesting cylinder 1 only has to wait for the heads to move an extra two cylinders.

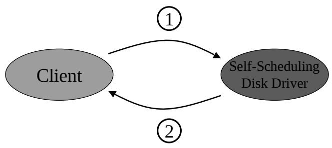
Separate scheduler



Scheduler as intermediary



Self-scheduling disk driver



Message passing solution 1

```
// shared channels
channel request;
channel[] reply = new channel[n];

// ith Client process of n ...
process Client_i {
    disk_block dblock;
    integer track, offset;
    // request a block of data from the disk
    send request <i, track, offset>;
    receive reply[i] <dblock>;
    // use the data from the disk …
}
```


Message passing solution 2

```
// Self-Scheduling Disk Driver process
process DiskDriver {
    queue saved;
    disk_block dblock;
    integer clientID, track, offset, nsaved = 0;
    while(true) {
        while(!empty(request) or nsaved == 0) {
            // wait for first request or receive another
            receive request <clientID, track, offset>;
            insert(saved, <clientID, track, offset>);
            nsaved++;
        }
    // select the pending request closest to the ...
```


Message passing solution 3

```
        // select the saved request closest to the
        // current head position and access the disk
        remove(saved, <clientID, track, offset>);
        nsaved--;
        dblock = read(track, offset);
        send reply[clientID] <dblock>;
    }
}
```

Summary

- on a shared memory machine, procedure calls and operations on condition variables are more efficient than message passing primitives
- most distributed systems are based on message passing since it is more natural and more efficient than simulating shared memory on a distributed memory machine
- neither asynchronous nor synchronous message passing have yet found their way into a widely accepted general purpose programming language
- message passing in concurrent programs remains at the level of OS or library calls.

The next lecture

Remote invocation

Suggested reading:

- Andrews (2000), chapter 8;
- Ben-Ari (1982), chapter 6;
- Burns & Davies (1993), chapter 5;
- Andrews (1991), chapters 9.