G52CON: Concepts of Concurrency

Lecture 13 Message Passing

Natasha Alechina
School of Computer Science
nza@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 example: active monitors

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

- processors share only a communication network, 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*

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 *symmetrical 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 difference is 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 and there is only one server process
- 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.

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.

Any questions about coursework?