G52CON: Concepts of Concurrency

Lecture 13 Message Passing

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

Outline of this lecture

- exercise: Dining Philosophers Problem
- distributed processing implementations of concurrency
- message passing
  - one way vs two way communication
  - naming schemes
  - asynchronous vs synchronous communication
- client-server example: active monitors

© Brian Logan 2007

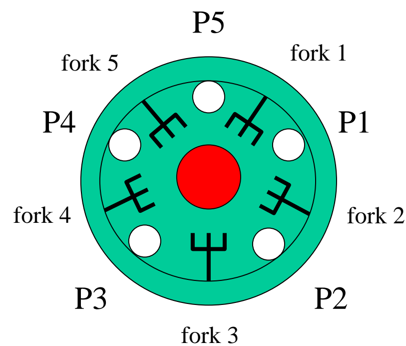
Dining Philosophers Problem

The *Dining Philosophers* problem illustrates mutual exclusion between processes which compete for overlapping sets of shared variables

- five philosophers sit around a circular table
- each philosopher alternately thinks and eats spaghetti from a dish in the middle of the table
- the philosophers can only afford five forks–one fork is placed between each pair of philosophers
- to eat, a philosopher needs to obtain mutually exclusive access to the fork on their left and right

The problem is to avoid *starvation*–e.g., each philosopher acquires one fork and refuses to give it up.

Dining Philosophers Problem



Deadlock in the Dining Philosophers

The key to the solution is to avoid *deadlock* caused by circular waiting:

- process 1 is waiting for a resource (fork) held by process 2
- process 2 is waiting for a resource held by process 3
- process 3 is waiting for a resource held by process 4
- process 4 is waiting for a resource held by process 5
- process 5 is waiting for a resource held by process 1.

No process can make progress and all processes remain deadlocked.

© Brian Logan 2007

Semaphore Solution

```
// Philosopher i, i == 1-4

while(true) {
    //get right fork then left
    P(fork[i]);
    P(fork[i+1]);
    // eat ...
    V(fork[i]);
    V(fork[i+1]);
    // think ...
}
```

```
// Philosopher 5

while(true) {
    //get left fork then right
    P(fork[1]);
    P(fork[5]);
    // eat ...
    V(fork[1]);
    V(fork[5]);
    // think ...
}
```

```
// Shared variables
binary semaphore fork[5] = {1, 1, 1, 1, 1};
```

© Brian Logan 2007

Deadlock

Although fully synchronised objects are always safe, threads using them are not always live

- some `synchronized` actions are multiparty – they acquire locks on multiple objects
- *deadlock* is possible when two or more objects are mutually accessible from two or more threads, and each thread holds one lock while trying to obtain another lock held by another thread

Example: `Cell`

```
class Cell { // Broken, do not use ...
    private long value;

    synchronized long getValue() { return value; }

    synchronized void setValue(long v) { value = v; }

    synchronized void swapValue(Cell other) {
        long t = getValue();
        long v = other.getValue();
        setValue(v);
        other.setValue(t);
    }
}
```

– see Lea (2000), p 87.

Example trace

Consider two threads, one of which invokes `a.swapValue(b)` while the other invokes `b.swapValue(a)`

Thread 1	Thread 2
acquire lock for `a` on invoking `a.swapValue(b)`	
pass the lock for `a` (since already held) on invoking `t = getValue()`	acquire lock for `b` on invoking `b.swapValue(a)`
block waiting for lock on b on invoking `v = other.getValue()`	pass lock for `b` (since already held) on invoking `t = getValue()`
	block waiting for lock on a on invoking `v = other.getValue()`

Resource ordering

One way to avoid this kind of deadlock is to use resource ordering:

- associate a numerical (or any other strictly orderable data type) tag with each object that can be an argument to a `synchronized` multiparty action
- if synchronization is always performed in tag order, then a situation can never arise in which a thread which has a lock on object `x` and is waiting for a lock on `y` while another thread has a lock on `y` and is waiting for a lock on `x`
- whichever thread locks the resource with the lowest tag first will acquire both locks while the other waits, and then the second thread will acquire both locks

Example: swapValue()

```
public void swapValue(Cell other) {
    if (System.identityHashCode(this) <
        System.identityHashCode(other))
        this.doSwapValue(other);
    else
        other.doSwapValue(this);
}
protected synchronized void doSwapValue(Cell other) {
    long t = getValue();
    long v = other.getValue();
    setValue(v);
    other.setValue(t);
}
```


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.

© Brian Logan 2007

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*.

© Brian Logan 2007

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

© Brian Logan 2007

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.

© Brian Logan 2007

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 …
```

© Brian Logan 2007

Active monitors 3

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

© Brian Logan 2007

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.