Concepts of Concurrency

Graham Hutton

Department of Computer Science
University of Nottingham

January 1997

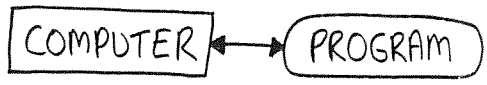
Concepts of Concurrency

Graham Hutton

Lecture 1 - Introduction

SEQUENTIAL PROGRAMMING

The most basic kind of computer can execute one program at a time:



Examples:

Sinclair Spectrum; Commodore 64; BBC Micro; DOS machines.

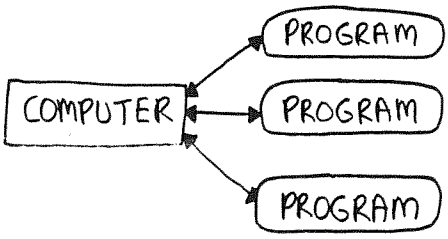
Since program instructions are executed in series, such computers are called sequential computers, and are programmed using sequential programming languages.

Examples:

Basic; Pascal.

MULTIPROGRAMMING

A more sophisticated kind of computer can give the illusion of more than one program executing at the same time, by periodically switching between programs:



Examples:

Apple Mac; Windows machines; Unix machines.

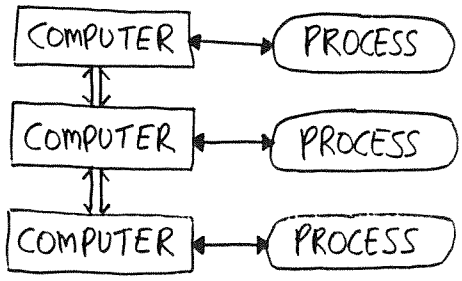
Such computers are called multiprogramming computers, and the programs themselves are usually called processes.

Example languages:

C

CONCURRENT PROGRAMMING

More than one program can really be executed at the same time by having more than one computer:



Examples:

Connection machine; Transputer machines; Networked computers.

Since many program instructions can be executed in parallel, such computers are called parallel computers, and are programmed using parallel (or concurrent) programming languages.

Examples:

Occam; Ada.

WHY CONCURRENT PROGRAMMING MATTERS

Concurrent programming is an interesting and useful topic to study for a number of reasons, including:

- New programming features

  Concurrent languages increase the programmers "power of expression", by providing new features for dealing with processes and communication between them.

- New problem-solving tools

  Even if parallelism is simulated by multiprogramming, many problems have a natural solution when one thinks in terms of concurrent processes.

- Increased performance

  On a truly parallel computer, being able to execute many processes at the same time can give significant performance gains over a purely sequential computer.

- Inherently concurrent applications

  Many modern computer applications involve concurrency (whether real or simulated) in an essential way.

  Examples:

  Operating systems;
  User-interfaces;
  Computer networks;
  Telecommunications software.

- Widely supported

  Most modern operating systems now support concurrent programming in some form or another.

  Examples:

  Unix;
  Windows 95;
  Macintosh;
  NeXTSTEP

SIMPLIFYING ASSUMPTIONS

To allow us to study the "essentials" of concurrent programming, we make a number of assumptions:

(1) We only consider programming languages in which all concurrency is explicitly specified within programs.

(2) We won't concern ourselves with whether programs are executed on a truly concurrent computer, or are just simulated on a multiprogramming computer.

(3) We'll ignore physical details concerning the

- number of computers available;
- way the computers are connected;
- speed of the computers and their connections;
- overhead of managing concurrency

LECTURE PLAN

1. Introduction
2. Issues in concurrency
3. Mutual exclusion I
4. Mutual exclusion II
5. Semaphores
6. Monitors
7. The dining philosophers
8. CCS I
9. CCS II
10. Semantics of concurrent programs
11. Tasks in Ada
12. Occam I
13. Occam II
14. Parallel algorithms
15. Summary of the course.

COURSE MATERIAL

- My slides will form the lecture notes for the course, and will be handed out at each lecture.
- Everything I expect you to know will be on the slides, but I also strongly recommend that you buy the following text book:

> M. Ben-Ari. Principles of Concurrent Programming. Prentice-Hall International, 1982.

There are a number of more recent books, but this is still the "original and best".

ASSESSMENT

- There will be no formal coursework, but a written examination at the end of the course.

A TASTE OF CONCURRENT PROGRAMMING

Consider the problem of sorting an array a of 100 integers into ascending numerical order.

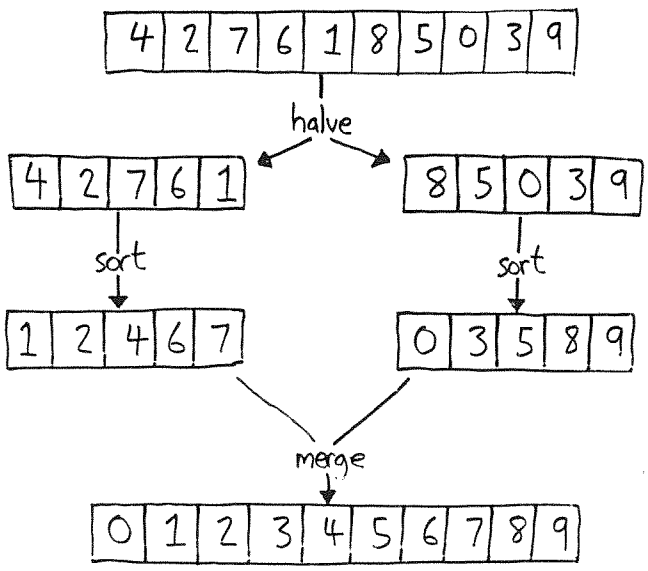
Here's a Pascal-like implementation of "selection sort", a simple sorting algorithm that works by finding the smallest number, then the next smallest, etc:

```
procedure sort (left, right : integer);
var
  i, j : integer;
begin
  for i := left to right - 1 do
    for j := i+1 to right do
      if a[j] < a[i] then
        swap (a[j], a[i])
end;
```

Now, sort (1,100) sorts the complete array a.

There are better sorting algorithms, but even just using selection sort we can do better, by thinking concurrently!

Consider dividing the array into two halves, sorting the two halves concurrently using selection sort, and then merging the two sorted halves into a sorted whole:



Assuming that cobegin ... coend means that statements in the sequence ... may be executed in parallel, and an auxilliary procedure merge for merging two sorted parts of the array, this idea can be implemented by:

```
procedure bettersort (left,right : integer);
var
  mid : integer;
begin
  mid := (left + right) div 2;
  cobegin
    sort (left, mid);
    sort (mid+1, right)
  coend;
  merge (left, mid, right)
end;
```

To compare the efficiency of the two sorting programs, let us count the "number of comparisons" in each.

First of all, sorting n integers using sort requires:

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

comparisons, which is approximately $\boxed{\frac{n^2}{2}}$.

Merging two sorted sequences totalling n integers requires at most $(n-1)$ comparisons. Hence sorting n integers using bettersort requires approximately

$$\underbrace{\frac{(n/2)^2}{2}}_{\text{sort the first halve}} + \underbrace{\frac{(n/2)^2}{2}}_{\text{sort the second halve}} + \underbrace{(n-1)}_{\text{merge the sorted parts}} = \boxed{\frac{n^2}{4} + n - 1}$$

comparisons, or if both sorts can be executed concurrently, we only count one $\frac{(n/2)^2}{2}$ term, to give:

$$\boxed{\frac{n^2}{8} + n - 1}$$

Examples

	sort	bettersort	with concurrency
n	$\frac{n^2}{2}$	$\frac{n^2}{4} + n - 1$	$\frac{n^2}{8} + n - 1$
10	50	34	21
100	5000	2599	1349
1000	500000	250999	125999

Notes

- Even if bettersort is executed on a sequential or multiprogramming computer, it is still better than sort!
- Defining bettersort recursively in terms of itself gives one of the best known algorithms ("mergesort").
- Adding cobegin ... coend raises many issues, which we will study in this course.

Concepts of Concurrency

Graham Hutton

Lecture 2 - Issues in Concurrency

CONCURRENCY vs PARALLELISM

Does "concurrency" = "parallelism"?

Generally, the terms are interchangeable, and both refer to "doing many things at the same time."

However, some authors make the following distinction:

Concurrency: concerns applications that inherently involve doing things at the same time.

Examples: networks; operating systems.

Parallelism: concerns applications in which performance can be improved by doing things at the same time.

Examples: sorting; searching.

COMMUNICATION

What makes concurrency interesting to study is that the concurrent processes are usually not independant, but most communicate with each other in some way.

Examples

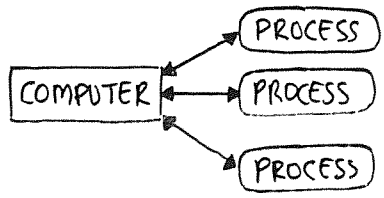
- In a computer network, data (email, www pages, ...) are continuously passed between computers.
- In an operating system, processes must negotiate for access to resources (disks, memory, ...).
- In a graphical user-interface, events (mouse movements, key presses, ...) must be sent to the appropriate processes for handling.

There are two basic methods of process communication:

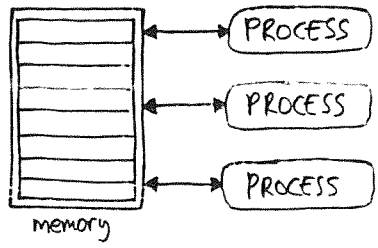
1. By "shared memory";
2. By "channels".

COMMUNICATION BY SHARED MEMORY

Many computers that support concurrent programming will actually be multiprogramming systems that give the illusion of more than one process executing at the same time by periodically switching between them:



In this setting, it makes sense to have processes communicate by a portion of shared memory in the computer, which can be accessed by any process:



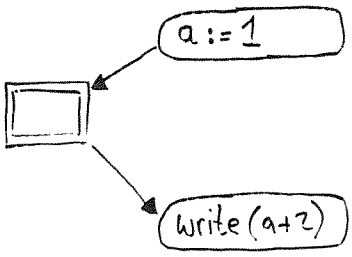
In terms of our Pascal-like implementation language, "shared memory" corresponds to "global variables".

Consider the following program fragment:

```
cobegin
  a := 1 ;
  write (a+2)
coend ;
```

It comprises two processes, a:=1 and write (a+2), that can be executed in parallel (indicated by cobegin ... coend.)

The two processes can communicate, in that the first writes to the global variable a and the second reads it:



The "expected" output from the program is 3.

However, assuming the initial value of a is 0, there are two possible outputs from the program:

3 or 2

This arises because we can't predict the order in which the two processes will be executed:

- If a:=1 is executed before write(a+2), then we get the "expected" output 3.
- If write (a+2) is executed before a:=1, then a is still 0, and we get the output 2.

Note:

Only one access to memory is possible at one time, so even if the two processes are executed at the same time, one will always get first access to a.

Summary

With communication by shared memory, concurrent programs may give "unexpected" results, depending on the order in which instructions are executed.

Conclusion

We need some means to control the order of execution of instructions between concurrent processes.

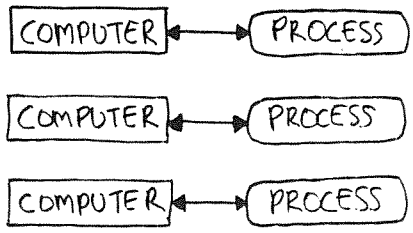
Solution

There are many ways of introducing control, including

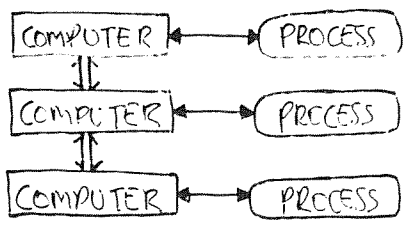
- Mutual exclusion algorithms;
- Semaphores;
- Monitors.

COMMUNICATION BY CHANNELS

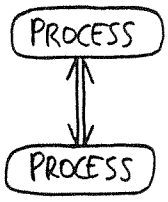
Even if we are really using a multiprogramming system, it will often be convenient to imagine that we are using a truly concurrent system, in which each process is executed on a separate computer:



In this setting, it is natural to think of the processes communicating by special "channels" that link the different computers together in some way:



It is convenient now, to remove the distinction between a process and the computer that executes it, and think of a channel as a direct link between two processes, over which data can be passed:



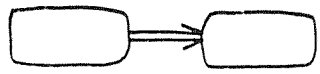
Examples

In a telephone network, phones communicate by channels that are made up of wires, fibre-optic cables, radio links, and satellite links, among others.

In the Unix operating system, processes can communicate by simple channels that are called "pipes".

- Unidirectional vs Bidirectional

A channel may be able to transfer data only in one direction (unidirectionally) between two processes:



or in both directions (bidirectionally):

Examples: unidirectional - a Unix "pipe";
bidirectional - a phone link.

- Synchronous vs Asynchronous

Data may only be able to be transferred over a channel when the receiver is ready (synchronous transmission), or at any time (asynchronous transmission.) In the later case, data can be lost unless the channel is "buffered".

Examples: synchronous - connecting by telephone;
asynchronous - sending an email.

SEMANTICS

A semantics for a programming language is a formal statement of what programs "mean".

Among other things, a formal semantics:

- Is a complete and precise specification of the language for programmers and implementors;
- Is necessary for justifying proofs about programs;
- Can give new insights into language design.

Two popular styles of giving semantics are:

① Denotational semantics;

② Operational semantics.

Denotational semantics

Sequential languages are often given semantics using the denotational approach, in which programs are given meaning as mathematical functions.

For example, the "state" during execution might be modelled as a function from variable names to values.

Operational semantics

Concurrent languages can be given denotational semantics, but problems arise from functions always returning one result, but concurrent programs possibly returning many.

Concurrent languages are usually given operational semantics, in which programs are given meaning as "abstract machines" that execute by making "transitions".

CORRECTNESS

What does it mean for a program to be correct?

How can we verify that it is correct?

For sequential programs:

1. We make a formal specification of what the program is expected to do, without saying how it should be done. The specification is often written using some variant of predicate logic.

   Example: specifying a "square root" function:

   $$\forall x \in \mathbb{N}.\ (\text{sqrt}\ x)^2 = x$$

2. We write a program that implements the specification:

   ```
   function sqrt (x: integer): integer ;
   ```

3. We prove the implementation satisfies the specification, using some kind of programming logic.

Concurrent programs can be thought of as several sequential programs running at the same time.

It is not surprising then that timing (or temporal) properties play an important rôle in the correctness of concurrent programs. There are two kinds:

1. "Safety" properties

   Properties that must always be true.

Examples

"Mutual exclusion" - at most one process in a given set can be in its "critical region" at any time.

Freedom from "deadlock" - at least one process must be able to make useful progress at any time.

2. "Liveness" properties

   Properties that must eventually be true.

Examples

"Fairness" - if a process makes a request, it will eventually be granted. This is also known as "freedom from individual starvation."

There are various kinds of fairness, involving timing constraints, priority queues, etc.

Safety and liveness properties are usually specified and proved using some variant of temporal logic.

Concepts of Concurrency

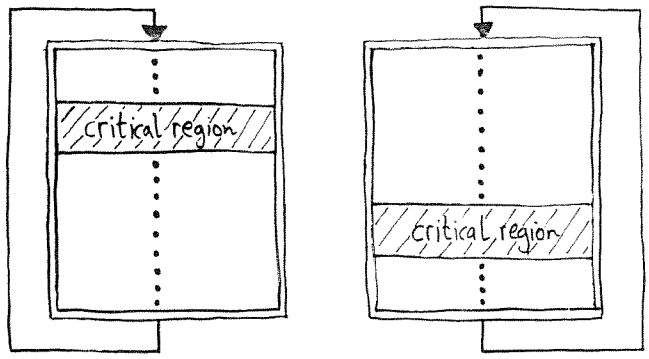
Graham Hutton

Lecture 3 - Mutual Exclusion I

THE PROBLEM

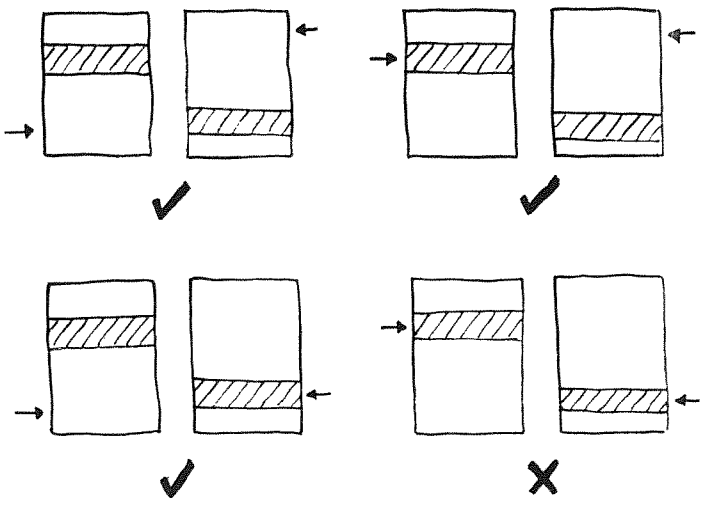
Many problems that arise in concurrent programming can be seen as instances of ensuring mutual exclusion.

Assume we are given two sequential processes executing concurrently, such that both processes loop forever, and contain a special "critical region":



The two processes satisfy mutual exclusion if they collaborate to ensure that at most one process can be in its critical region at any given time.

Examples



The critical region usually involves access to a shared resource (global variable, communication channel, file, ...) for which it is essential to the correctness of the complete program that only one process can access the resource at any given time.

HEALTHYNESS REQUIREMENTS

In addition to the requirement of mutual exclusion, we impose a number of healthyness requirements:

Fairness

- If a process requests to enter its critical region, eventually the request will be granted.

Loose coupling

- If something goes wrong and a process dies outwith its critical region, this must not prevent the other process from continuing to access its critical region.

  In general, we can't hope to cope with the case when a process dies within its critical region.

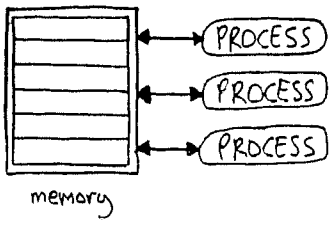
- If one process requests to enter its critical region more frequently than the other, the algorithm should not preclude this process gaining access more often.

SYSTEM ASSUMPTIONS

We will develop an algorithm for ensuring mutual exclusion, under the following system assumptions:

Shared memory

Communication between processes is by shared memory, i.e. by reading/writing global variables:



Atomic operations

Single read or writes to shared memory are atomic: at most one read or write to memory can be executed at any given time, and if two concurrent processes both try to access memory at the same time, a memory arbiter will force one access to be executed before the other.

DEKKER'S ALGORITHM

Mutual exclusion is a tricky problem.

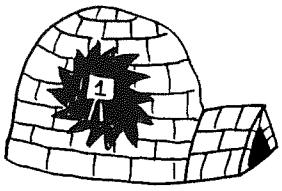
There are many mutual exclusion algorithms. The original and most famous is Dekker's algorithm, named after the Dutch mathematician who invented it.

Rather than presenting the algorithm straight off, we follow Dijkstra's approach from 1968: the algorithm is developed from a series of failed attempts.

The failed attempts illustrate a number of common mistakes that can be made in concurrent programs.

FIRST ATTEMPT

Imagine an igloo containing a blackboard:



On the blackboard is written the number of a process (1 or 2) whose turn it is to enter its critical region.

The small size of the igloo represents the memory arbiter: at most one process can enter the igloo and inspect or change the blackboard at any given time.

Now we can specify a mutual exclusion algorithm...

The first algorithm

Initially 1 is written on the blackboard.

A process wanting to enter its critical region crawls into the igloo (when it is empty) and inspects the blackboard. One of two things then happens:

- If the blackboard contains the number of the process, the process leaves the igloo and enters its critical region. When the process eventually leaves its critical region, it changes the number on the blackboard to that of the other process.

or

- If the blackboard does not contain its number, the process leaves the igloo, waits a while, and retries to enter its critical region.

Assuming a global integer variable turn that represents the blackboard, this algorithm can be translated into the following Pascal-like implementation for process 1:

```
procedure p1;

begin
  repeat
    [ ⋮ ];
    while turn = 2 do
      [wait a while];
    [critical region];
    turn := 2;
    [ ⋮ ]
  forever
end;
```


Dually, process 2 is implemented as follows:

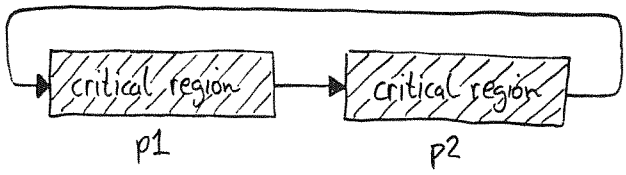
```
procedure p2;

begin
  repeat
    [ ⋮ ];
    while turn = 1 do
      [wait a while];
    [critical region];
    turn := 1;
    [ ⋮ ]
  forever
end;
```


The two processes are now combined by

```
begin
  turn := 1;
  cobegin
    p1; p2
  coend
end;
```

The above algorithm allows p1 and p2 to execute concurrently, but ensures that the two processes take turn about to execute their critical regions:



Hence the algorithm ensures

- Mutual exclusion;
- Fairness.

However, there are problems with this algorithm:

- Suppose something goes wrong and p1 dies at some point outwith its critical region. Then p2 will only be able to enter its critical region at most one more time, since p1 no longer exists to change turn back to 2 again.
- Suppose p1 wants to enter its critical region 100 times per day, while p2 only requires access to its critical region once per day. Then p1 will be restricted by p2 to also having just one access.

That is, our first mutual exclusion algorithm does not satisfy our healthyness requirement of "loose coupling".