Concepts of Concurrency

Graham Hutton

Department of Computer Science
University of Nottingham

January 1997

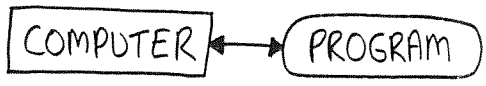
Concepts of Concurrency

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.

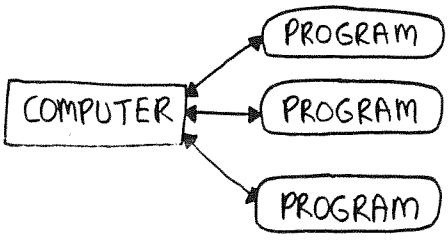
Since program instructions are executed in series, such computers are called sequential computers, and are programmed using sequential programming languages.

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.

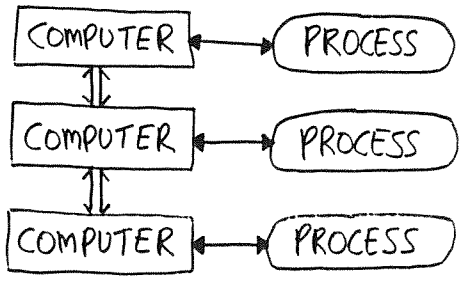
Such computers are called multiprogramming computers, and the programs themselves are usually called processes.

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 textbook:

> 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.

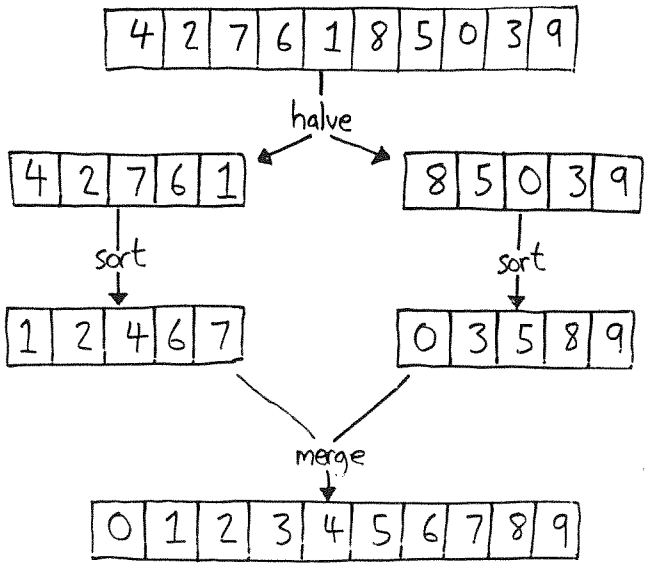
Here's a Pascal-like implementation of "selection sort", a simple sorting algorithm that works by finding the smallest number, then the next smallest, etc:

```
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 <u>sort</u> requires:

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

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

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

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

comparisons, or if both <u>sorts</u> can be executed <u>con-currently</u>, 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 <u>bettersort</u> is executed on a sequential or multiprogramming computer, it is still better than <u>sort</u>!
- Defining bettersort <u>recursively</u> in terms of itself gives one of the best known algorithms ("mergesort").
- Adding <u>cobegin</u> ... <u>coend</u> 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

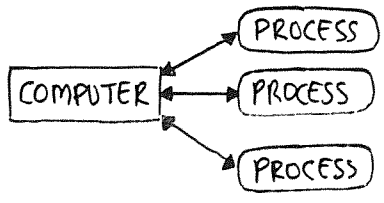
- In a computer network, data (email, www pages, ...) are continuously passed between computers.
- In an operating system, processes must negotiate for access to resources (disks, memory, ...).
- In a graphical user-interface, events (mouse movements, key presses, ...) must be sent to the appropriate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

There are two basic methods of process communication:

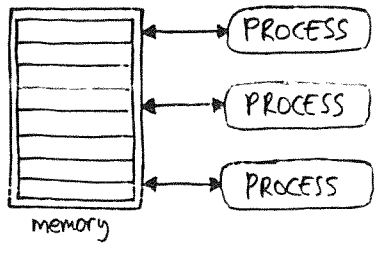
1. By "shared memory";
2. By "channels".

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:



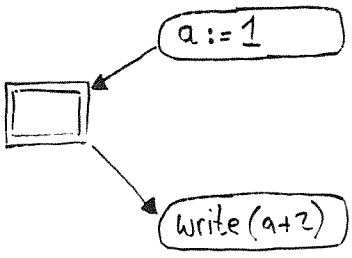
In terms of our Pascal-like implementation language, "shared memory" corresponds to "global variables".

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.

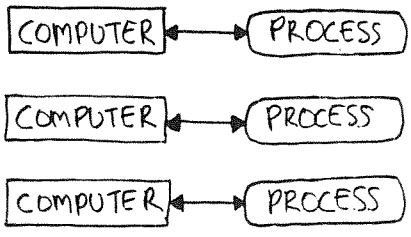
Solution

There are many ways of introducing control, including

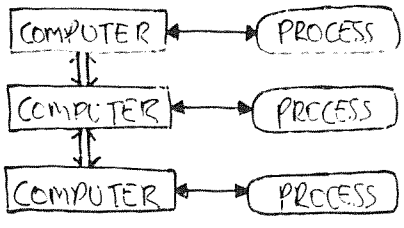
- Mutual exclusion algorithms;
- Semaphores;
- Monitors.

COMMUNICATION BY CHANNELS

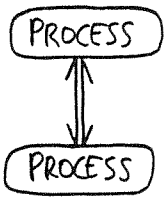
Even if we are really using a multiprogramming system, it will often be convenient to imagine that we are using a truly concurrent system, in which each process is executed on a separate computer:



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:



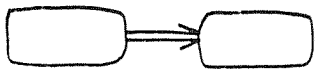
Examples

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

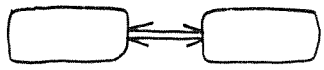
In the Unix operating system, processes can communicate by simple channels that are called "pipes".

- **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

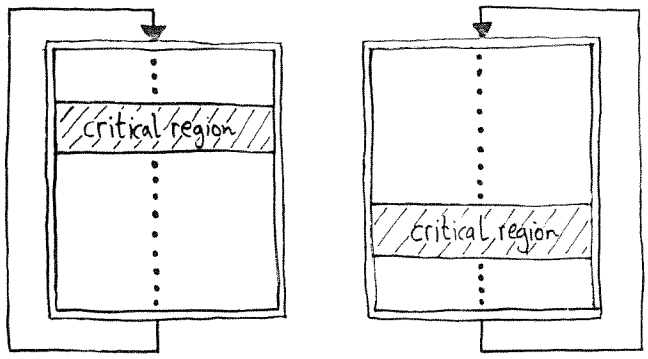
Graham Hutton

Lecture 3 - Mutual Exclusion I

THE PROBLEM

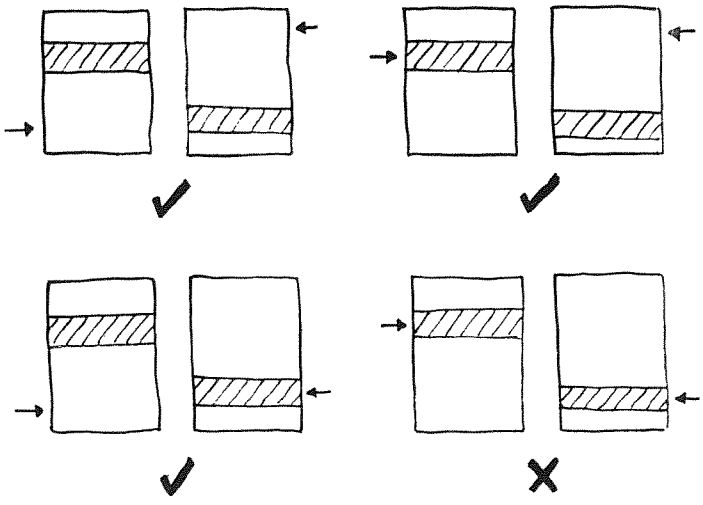
Many problems that arise in concurrent programming can be seen as instances of ensuring mutual exclusion.

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.

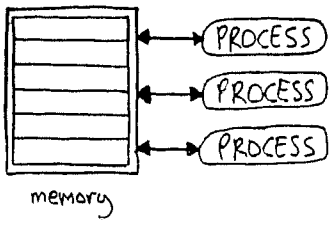
- If one process requests to enter its critical region more frequently than the other, the algorithm should not preclude this process gaining access more often.

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.

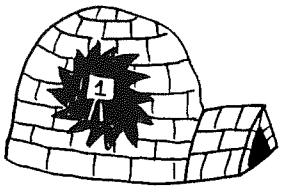
There are many mutual exclusion algorithms. The original and most famous is Dekker's algorithm, named after the Dutch mathematician who invented it.

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 <u>turn</u> 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;
```

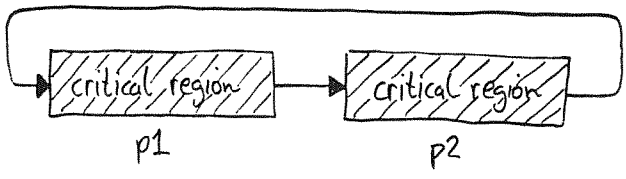

Dually, process 2 is implemented as follows:

```
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 <u>turn about</u> to execute their critical regions:



Hence the algorithm ensures

- Mutual exclusion;
- Fairness.

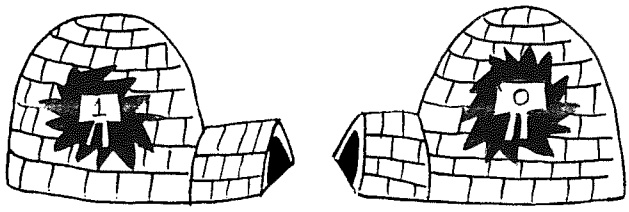
However, there are <u>problems</u> with this algorithm:

- 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 <u>one more time</u>, 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 <u>just one</u> access.

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

SECOND ATTEMPT

The problem with the first attempt is that p1 and p2 are too tightly coupled. One way to loosen the coupling is to use two igloos — one for each process —, rather than a single igloo for both processes:



The blackboard for each process contains 0 or 1:

0 means the process is in its critical region;

1 means the process is not in its critical region.

Now we can specify a new mutual exclusion algorithm...

The second algorithm

Initially 1 is written on both blackboards.

A process wanting to enter its critical region crawls into the igloo of the other process, and inspects the blackboard. One of two things then happens:

- If the blackboard contains 1, the process leaves the igloo, changes the number on the blackboard in its own igloo to 0, and enters its critical region. When the process eventually leaves its critical region, it changes the number on the blackboard in its own igloo back to 1.

or

- If the blackboard contains 0, the process leaves the igloo, waits a while, and then retries to enter its critical region.

Assuming global integer variables c1 and c2 that represent the blackboards for the two processes, this algorithm translates to the following for process 1:

```
procedure p1;

begin
  repeat
      [   ⋮   ];
      while c2 = 0 do
          [ wait a while ];
      c1 := 0;
      [ critical region ];
      c1 := 1;
      [   ⋮   ]

  forever
end;
```


Process 2 is implemented similarly, and the two processes are now combined by:

```
begin
  c1 := 1 ; c2 := 1;
  cobegin
    p1 ; p2
  coend
end;
```

The above algorithm allows p1 and p2 to execute concurrently, and satisfies the healthyness requirement of "loose coupling" that the first algorithm did not:

- If p1 dies at some point outwith its critical region, then c1 will be 1, which means that p2 will still be able to gain access to its critical region.
- If p1 wants to enter its critical region more frequently than p2, the algorithm does not preclude this

However, there is a problem with this algorithm:

- The following is a valid interleaving of the actions of the concurrent processes p1 and p2:

  p1 finds c2 = 1;
  p2 finds c1 = 1;
  p2 sets c2 to 0;
  p1 sets c1 to 0;
  p1 enters its critical region;
  p2 enters its critical region.

That is, it is possible for both p1 and p2 to be in their critical regions at the same time, and so our second mutual exclusion algorithm does not ensure mutual exclusion!

To be continued...

Concepts of Concurrency

Graham Hutton

Lecture 4 - Mutual Exclusion II

THIRD ATTEMPT

The problem with the second attempt is that with concurrent execution of p1 and p2, we cannot be sure that nothing will happen between one process finding 1 on the other processes blackboard and writing 0 on its own:

p1 finds c2 = 1;
p2 finds c1 = 1;
p2 sets c2 to 0;
p1 sets c1 to 0;

X

One solution is to ensure that c1 = 0 immediately that we find c2 = 1, by swapping the order of the operations:

> A process wanting to enter its critical region changes its own blackboard to 0 before waiting for the other blackboard to become 1.

Process 1 is now modified as follows:

```
procedure p1;

begin
  repeat
    [      ⋮      ];
    c1 := 0;
    while c2 = 0 do
      [ wait a while ];
    [ critical region ];
    c1 := 1;
    [      ⋮      ]
  forever
end;
```


Process 2 is modified similarly, and the code for combining the two processes remains the same.

It can be shown that the above algorithm now ensures mutual exclusion. However:

- The following is a valid interleaving of the actions of the concurrent processes p1 and p2:

  p1 sets c1 to 0;
  p2 sets c2 to 0;
  p1 loops until c2 = 1;
  p2 loops until c1 = 1;

That is, it is possible for both processes to reach a deadlocked state in which each loops forever waiting for the other to change its blackboard to 1.

So our third mutual exclusion algorithm does not satisfy our healthyness requirement of fairness.

FOURTH ATTEMPT

The problem with the third attempt is that when a process writes 0 on its own blackboard, it is not just requesting to enter its critical region, but <u>insisting</u> that it gains entry, even if the other process is sim--ultaneously requesting to enter its own critical region.

One solution to the problem is to introduce <u>politeness</u>: a process temporarily gives the other process the chance to "<u>go first</u>" if both processes request to enter their critical regions at the same time.

As in the third attempt, both blackboards contain 0 or 1:

0 means the process <u>wants</u> to enter its critical region;

1 means it <u>does not want</u> to enter its critical region.

Now we can specify a new mutual exclusion algorithm...

The fourth algorithm

Initially 1 is written on both blackboards.

A process wanting to enter its critical region writes 0 on its own blackboard, and inspects the other black--board. One of two things then happens:

- If the other blackboard also <u>contains 0</u>, the process changes its own blackboard back to 1, waits a while, and then retries to enter its critical region.

<u>or</u>

- If the other blackboard <u>contains 1</u>, the process enters its critical region. When the process eventually leaves its critical region, it changes its own blackboard back to 1.

Process 1 is now implemented as follows:

```
procedure p1;

begin
  repeat
    [      ⋮      ];
    c1 := 0;
    while c2 = 0 do
      begin
        c1 := 1;
        [ wait a while ];
        c1 := 0
      end;
    [ critical region ];
    c1 := 1;
    [      ⋮      ]
  forever
end;
```


Process 2 is implemented similarly, and the code for combining the two processes remains the same.

The above algorithm again ensures mutual exclusion, but still <u>does not</u> ensure fairness. For example:

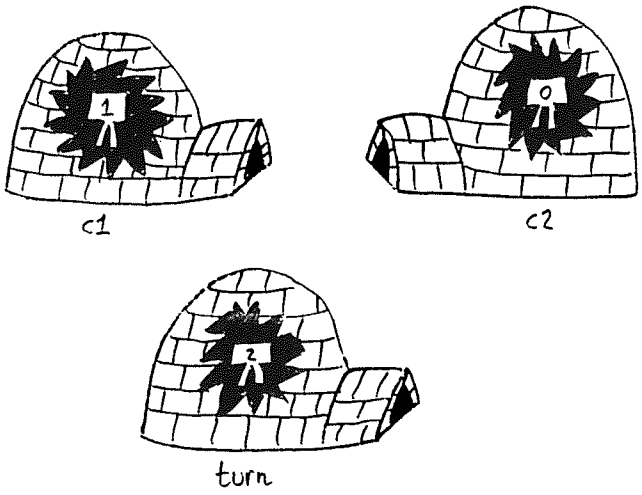
```
p1 sets c1 to 0;
p2 sets c2 to 0;
p1 finds c2 = 0;
p2 finds c1 = 0;
p1 sets c1 to 1;
p2 sets c2 to 1;
```

That is, it is now possible for both processes to say "after you" turn about <u>infinetely</u> <u>often</u>, and never enter their critical regions.

DEKKER'S ALGORITHM

The problem with the fourth attempt is that the processes are <u>too polite</u>: if both want to enter their critical regions at the same time, there is nothing that precludes each process offering the other the right to go first, infinitely often.

Our solution is to use <u>three</u> igloos, with the extra igloo being used to resolve conflicts in a fair way:



As in the fourth attempt, each process has its own blackboard (c1 and c2), which contain 0 or 1:

0 means the process <u>wants</u> to enter its critical region;

1 means it <u>does not</u> want to enter its critical region.

Inspired by the first attempt, the extra blackboard contains the number of the process (1 or 2) whose <u>turn</u> it is if both processes request to enter their critical regions at the same time.

We can now specify a new mutual exclusion algorithm...

<u>Dekker's algorithm</u>

Initially, all three blackboards contain 1.

A process (say p1) wanting to enter its critical region writes 0 on its blackboard, and inspects the blackboard of the other process. One of two things then happens:

- If the other blackboard also <u>contains 0</u>, then the process consults the <u>turn</u> blackboard:

  <u>turn = 2</u> means the other process has priority, so the process changes its own blackboard to 1, waits until turn changes to 1, and then tries again.

  <u>turn = 1</u> means the process itself has priority, so the process waits a while (allowing the other process to change its blackboard to 1), and then tries again.

<u>or</u>

- If the other blackboard <u>contains 1</u>, the process enters its critical region. When the process eventually leaves its critical region, it changes turn to 2, and its own blackboard back to 1.

Process 1 is now implemented as follows:

```
procedure p1;
begin
  repeat
      [      ⋮      ];
    c1 := 0;
    while c2 = 0 do
        if turn = 2 then
            begin
                c1 := 1;
                while turn = 2 do
                    [ wait a while ];
                c1 := 0
            end
        else
            [ wait a while ];
    [ critical region ];
    turn := 2 ; c1 := 1 ;
      [      ⋮      ]
  forever
end;
```

Dually, process 2 is implemented by:

```
procedure p2;
begin
  repeat
      [   ⋮   ];
    c2 := 0;
    while c1 = 0 do
        if turn = 1 then
            begin
              c2 := 1;
              while turn = 1 do
                  [ wait a while ];
              c2 := 0
            end
        else
            [ wait a while ];
    [// critical region //];
    turn := 1; c2 := 1;
      [   ⋮   ]
  forever
end;
```


The two processes are combined by:

```
begin
  c1 := 1 ; c2 := 1 ; turn := 1;
  cobegin
    p1 ; p2
  coend
end;
```

Dekker's algorithm satisfies <u>all</u> our requirements:

<u>mutual exclusion</u>; <u>fairness</u>; <u>loose coupling</u>.

<u>Note</u>: In the first attempt, using <u>turn</u> lead to one process being <u>locked out</u> its critical region if the other process dies. This doesn't happen with Dekker's algorithm, since turn is only used if both processes want to enter their critical regions at the same time; if one process dies, then turn won't be used again, so there is no lockout.

A HARDWARE SOLUTION

Ensuring mutual exclusion is a tricky problem for concurrent systems in which single reads and writes are the only atomic operations on shared memory.

<u>The problem</u>:

Read and write operations for different concurrent processes are <u>interleaved</u>, and we cannot predict the order in which the operations for different processes will be executed.

<u>A software solution</u>:

Dekker's algorithm.

<u>A hardware solution</u>:

Assume an extra atomic operation that <u>both</u> reads and writes to shared memory.

One such operation is "<u>test and set</u>", which is provided by many modern microprocessors.

In terms of our Pascal-like implementation language, we can think of "test and set" as a <u>special function</u>:

```
function testandset (var x: integer) : integer;
```

such that testandset(x) is an <u>atomic</u> operation that stores ("tests") the value of x, then "sets" x to 1, and finally returns the stored value of x as the result.

<u>Note</u>: <u>atomic</u> means that when testandset(x) is executed, no other operation on shared memory can be executed until testandset(x) is complete.

Assuming a global integer variable c such that:

c=1 means a process is in its critical region;

c=0 means neither process is in its critical region...

... it is now easy to ensure mutual exclusion. For example, process 1 can be implemented as follows:

```
procedure p1;

begin
  repeat
      [      ⋮      ];
      while testandset(c) = 1 do
          [ wait a while ];
      [// critical region //];
      c := 0;
      [      ⋮      ]
  forever
end;
```

<u>Note</u>: this solution has a minor <u>fairness</u> problem - there is nothing that precludes one process <u>always</u> gaining entry to its critical region and the other process <u>never</u>.

Concepts of Concurrency

Graham Hutton

Lecture 5 - Semaphores

BACKGROUND

Concurrent programming using low-level operations, such as load/store/testandset has a number of drawbacks, including:

- Programs often involve "busy waiting" loops, which take up valuable processor time that could be better used (particularly on a multiprogramming system):

```
while ... do
    [wait a while];
```

- Solving concurrent problems (e.g. mutual exclusion) and proving the solution correct can be tricky.

Semaphores (due to Dijkstra, 1968) provide higher-order operations that avoid the need for busy waiting loops, and allow many concurrent problems to be solved and proved correct in a more elegant manner.

DEFINITION

A semaphore is an integer variable s such that

s only takes on values ≥ 0,

together with two atomic operations on s:

wait(s) and signal(s)

Semaphores are usually implemented by the underlying operating system, but the behaviour of the operations can be explained using Pascal-like definitions:

```
procedure wait (var s: integer);

begin
  if s>0 then
     s := s - 1
  else
     [suspend the process that called wait(s)]
end;
```



```
procedure signal (var s: integer);

begin
  if [no processes are suspended on s] then
     s := s + 1
  else
     [restart one of the suspended processes]
end;
```

There are a number of points to note, including:

- Terminology

  If a semaphore s is used so that s only takes on values 0 and 1, the semaphore is called a binary semaphore. Otherwise it is called a general semaphore.

- Abstraction

  wait and signal are the only operations allowed on a semaphore s, except for a single assignment of s to an initial value at the beginning of the program.

- **Atomicity**

  wait and signal are atomic operations: at most one such operation can be executed on a particular semaphore at any given time. Operations on different semaphores may be executed in parallel.

  Note: wait terminates after suspending its calling process, thus allowing further calls to wait and signal.

- **Fairness**

  signal doesn't specify which process should be restarted if more than one is suspended. There are many definitions of "fairness" for semaphores, but for simplicity we will assume a FIFO queue of processes to be restarted.

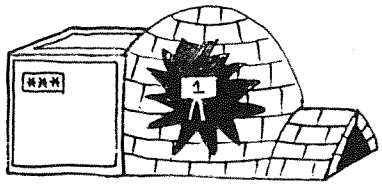
- **Busy waiting**

  Because wait(s) suspends its calling process if s=0, there is no "busy waiting" loop wasting processor time.

MUTUAL EXCLUSION

Using a binary semaphore s it is easy to ensure mutual exclusion for two concurrent processes.

We can think of the semaphore as an igloo containing a blackboard, attached to a deep freezer!



On the blackboard is written the value of the semaphore (0 or 1), and the deep freezer is used to suspend processes until they are restarted by signal.

Now we can specify a mutual exclusion algorithm...

Initially 1 is written on the blackboard.

A process wanting to enter its critical region crawls into the igloo (when empty), and executes a wait operation. One of two things then happens:

- If the blackboard contains 1, it is decremented to 0, and the process leaves the igloo and enters its critical region. When the process eventually leaves its critical region, it crawls into the igloo again, and executes a signal operation:

  If the other process is suspended in the freezer, it is restarted; otherwise the blackboard is incremented from 0 to 1.

or

- If the blackboard contains 0, the process crawls into the freezer and suspends. The igloo itself is then empty, allowing the other process to enter.

Assuming a global semaphore s, this algorithm translates into the following implementation for process 1:

```
procedure p1;
begin
   repeat
      [      ⋮      ];
      wait(s);
      [critical region];
      signal(s);
      [      ⋮      ]
   forever
end;
```

Process 2 is implemented precisely the same, and the two processes are combined by ...

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

This algorithm ensures <u>all</u> our requirements:

- <u>Mutual exclusion</u> (wait and signal are atomic).
- <u>Fairness</u> (if both processes want to enter their critical regions at the same time, one will go first and the other will go immediately afterwards.)
- <u>Loose coupling</u> (if one process dies outwith its critical region, this doesn't affect the other; the processes can also execute at different frequencies.)

In addition, it avoids the "<u>busy waiting</u>" loops used in Dekker's algorithm and the testandset solution.

MUTUAL EXCLUSION FOR n PROCESSES

The mutual exclusion algorithm using a binary semaphore extends naturally to the case of <u>$n > 2$</u> processes.

The code for the processes remains the same, and the n processes are combined by:

```
begin
  s := 1 ;
  cobegin
    p1 ; p2 ; ... ; pn
  coend
end;
```

However, this algorithm <u>may not</u> ensure fairness:

- Depending on <u>which</u> process is restarted by signal if more than one is suspended, some processes may never be restarted and never enter their critical regions. This doesn't occur with a FIFO queue.

GENERALISING MUTUAL EXCLUSION

Mutual exclusion requires that at most <u>1</u> process from n is in its critical region at any given time.

The mutual exclusion algorithm for n processes generalises naturally to allow at most <u>k</u> processes to be simultaneously in their critical regions.

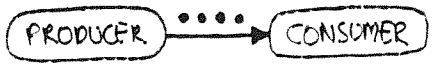
The code for the processes remains the same, but the binary semaphore s is replaced by a <u>general</u> semaphore, and the n processes are combined by:

```
begin
  s := k ;
  cobegin
    p1 ; p2 ; ... ; pn
  coend
end;
```


THE PRODUCER-CONSUMER PROBLEM

Along with mutual exclusion, many problems that arise in concurrent programming can be seen as instances of the <u>producer-consumer</u> problem.

Assume we are given two sequential processes executing concurrently, one process <u>producing</u> data values and the other process <u>consuming</u> them:



The problem is to allow the two processes to communicate <u>asynchronously</u>: data can be transmitted even when the consumer is not ready to receive it. This behaviour requires some kind of <u>buffering</u>.

The problem has a natural solution using semaphores...

Assuming we are given an infinite buffer (or queue), the problem boils down to satisfying two requirements:

① The consumer process never tries to remove a data value from the buffer if it is empty;

② The operation of adding and removing data values from the buffer are mutually exclusive.

Our solution uses two global semaphores:

n - a general semaphore used to count the number of data values in the buffer;

b - a binary semaphore used to ensure mutual exclusion of the two buffer operations.

The two processes can now be implemented by ...


```
procedure producer;
begin
  repeat
    [produce a data item];
    wait(b);
    [append it to the buffer];
    signal(b);
    signal(n)
  forever
end;
```

```
procedure consumer;
begin
  repeat
    wait(n);
    wait(b);
    [remove the first item from the buffer];
    signal(b);
    [process it]
  forever
end;
```


The two processes are combined by:

```
begin
  n:=0;
  b:=1;
  cobegin
    producer ; consumer
  coend
end;
```

This algorithm satisfies both our requirements.

Using semaphores rather than lower-level operations, we avoid the need for busy-waiting loops.

A BINARY SOLUTION

The producer-consumer problem can also be solved only using binary semaphores:

The general semaphore n is replaced by an integer variable n, together with a binary semaphore nonempty that is used to suspend the consumer process when the buffer is empty (n=0).

The producer process is now implemented by:

```
procedure producer;
begin
  repeat
    [produce a data item];
    wait(b);
    [append it to the buffer];
    n := n+1
    if n=1 then signal(nonempty);
    signal(b)
  forever
end;
```

The consumer process is implemented by:

```
procedure consumer;
var
  temp : integer;
begin
  wait (nonempty);
  repeat
     wait (b);
     [remove the first item from the buffer];
     n := n - 1;
     temp := n;
     signal (b);
     [process it];
     if temp = 0 then wait (nonempty)
  forever
end;
```

The two processes are combined by:


```
begin
   n := 0;
   b := 1;
   nonempty := 0;
   cobegin
      producer ; consumer
   coend
end;
```

Note: the use of the local variable temp in consumer is essential. If the conditional statement

if temp = 0 then wait (nonempty)

is simply replaced by

if n = 0 then wait (nonempty)

then the consumer process may attempt to remove a data item when the buffer is empty. (Why?)

Concepts of Concurrency

Graham Hutton

Lecture 6 - Monitors

BACKGROUND

Semaphores allow synchronisation between programs without the need for busy-waiting loops. However:

- wait and signal can be used anywhere;
- The compiler provides no support for checking that wait and signal are used correctly.

Monitors provide a more structured approach to solving concurrent problems than semaphores:

A monitor is a kind of abstract data type: some data together with some operations.

The compiler ensures that:

- The data is only accessed using the operations;
- The operations are executed mutually exclusively.

DEFINITION

A monitor m comprises three components:

1. Variable declarations;
2. Code that initialises the variables;
3. Procedures that operate on the variables;

subject to the following conditions:

- Hiding

The variables are only accessible within the initialisation code and the monitor procedures, but exist throughout execution of the main program: the variables have local scope, but the same lifetime as the main program.

Trying to access the variables outwith a monitor is an error, and will be detected by the compiler.

- Communication

The monitor procedures usually communicate with the outside world explicitly using parameters, rather than implicitly using global variables.

- Mutual exclusion

The difference between a monitor and an abstract data type (or module) is that the run-time system ensures that at most one procedure for a given monitor can be executing at any given time: the procedures are executed mutually exclusively.

In summary, we have the following slogan:

$$\boxed{\text{monitor} = \text{abstract data type} + \text{mutual exclusion}}$$

SYNTAX

We'll assume our Pascal-like language provides the following syntax for defining monitors:

```
monitor m;

  [ variable declarations ];

  [ procedure declarations ];

begin
    [ initialisation code ]

end;
```


CONDITION VARIABLES

For synchronisation between monitor procedures we need some kind of special primitives. A condition variable is a variable c within a monitor, together with two atomic primitives on c:

wait(c) - Suspends the procedure that called wait(c) on a FIFO queue of procedures for c;

signal(c) - Restarts the first procedure in the queue for c, if the queue is non-empty.

There are a number of points to note, including:

- Semaphores?

  wait and signal are similar to the operations of a semaphore, except that a condition variable has no counter. If required, a counter can be implemented manually using a monitor variable.

- Abstraction

  wait and signal are the only operations allowed on a condition variable c. Trying to assign or test c is an error, and will be detected by the compiler.

- Atomicity

  wait and signal are atomic, but operations on different condition variables may be executed in parallel.

  (In fact this can't happen, because monitor procedures are sequential and executed mutually exclusively.)

- Behaviour

  A monitor procedure suspended by wait(c) frees up mutual exclusion, allowing another procedure to execute.

  signal(c) has no effect if the queue of procedures suspended on c is empty.

PRIORITY OF PROCESSES

Problem: if a monitor procedure executes signal(c), there may be three kinds of processes wanting to execute, while mutual exclusion requires that only one does:

1. The procedure that executed signal(c);
2. The first procedure suspended in the queue for c;
3. Other procedures blocked by mutual exclusion.

Solution

- Monitor procedures must terminate immediately after executing a signal. This eliminates 1.
- Procedures suspended on c have priority over blocked procedures: 2 > 3. This is called the immediate resumption requirement.

THE PRODUCER-CONSUMER PROBLEM

Using a monitor to handle the buffer, it is easy to solve the producer-consumer problem.

The monitor encapsulates three variables,

- b — an infinite buffer;
- n — the number of items in the buffer;
- nonempty — a condition variable used to suspend a remove when the buffer is empty;

and provides two procedures:

- append — appends an integer to the buffer;
- remove — remove the first integer from the buffer.

The monitor is implemented as follows ...


```
monitor buffer;

var
   b        : [a type of infinite buffers];
   n        : integer;
   nonempty : condition;

procedure append (v : integer);

begin
   [append v to the buffer];
   n := n+1;
   signal (nonempty)
end;

procedure remove (var v : integer);

begin
   if n=0 then
      wait (nonempty);
   [remove the first item v from the buffer];
   n := n-1
end;
```



```
begin
   [initialise the buffer];
   n := 0
end;
```

The producer/consumer processes are now defined by:

```
procedure producer;

var
   v : integer;

begin
   repeat
      [produce a data item v];
      append (v)
   forever
end;
```



```
procedure consumer;

var
   v : integer;

begin
   repeat
      remove (v);
      [process the data item v]
   forever
end;
```

Note

This solution is more structured than the semaphores version: the buffer monitor takes care of all the concurrency issues in the problem (synchronisation + mutual exclusion), allowing the producer/consumer processes to be programmed more simply.

MONITORS SIMULATING SEMAPHORES

It is easy to define a monitor that simulates a general semaphore s. The monitor comprises:

- s - an integer counter;
- nonzero - a condition variable used to suspend a wait when the counter is zero;
- wait_s - the wait operation;
- signal_s - the signal operation;

and is implemented as follows:

```
monitor sem_s;
Var
    s        : integer;
    nonzero  : condition;
```



```
procedure wait_s;
begin
    if s = 0 then
        wait (nonzero);
    s := s - 1
end;

procedure signal_s;
begin
    s := s + 1;
    signal (nonzero)
end;

begin
    s := 0
end;
```

Such a simulation allows a semaphore-based algorithm to be converted to a monitor-based programming language.

SEMAPHORES SIMULATING MONITORS

Dually, we can use semaphores to simulate a monitor. As a concrete example, we show how to simulate the buffer monitor using semaphores. We require the following global variables:

- b - an infinite buffer;
- n - the number of items in the buffer;
- s - a binary semaphore used to ensure mutual exclusion of the procedures;
- nonempty - a binary semaphore used to suspend a remove when the buffer is empty;
- suspended - the number of procedures suspended in the FIFO queue for nonempty.

Note

suspended is required because the meaning of signal(nonempty) depends on the number of suspended procedures.

The append procedure is now implemented by:

```
procedure append (v : integer);
begin
    wait (s);
    [append v to the buffer];
    n := n + 1;
    if suspended > 0 then
        signal (nonempty)
    else
        signal (s)
end;
```

Note

The monitor operation signal(nonempty) is translated to a conditional expression on the number of suspended procedures.

The <u>remove</u> procedure is now implemented by:

```
procedure remove (var v : integer);
begin
  wait (s);
  if n=0 then
     begin
        suspended := suspended + 1;
        signal (s);
        wait (nonempty);
        suspended := suspended - 1
     end;
  [remove the first item v from the buffer];
  n := n - 1;
  signal (s)
end;
```

Note

The monitor operation <u>wait(nonempty)</u> is translated to a sequence of operations also involving suspended and s.

Note also that the translation:

- Doesn't deal with "<u>hiding</u>". For this we would need modules or abstract data types.
- Is rather <u>subtle</u> in how it ensures our requirements about the priority of processes
- Allows a monitor-based algorithm to be <u>converted</u> to a semaphore-based programming language.

Conclusion

Semaphores and monitors are <u>equivalent</u> in expressive power: each can faithfully simulate the other.

However, simulation one way is easier than the other: monitors are "<u>higher-level</u>" than semaphores.

Concepts of Concurrency

Graham Hutton

Lecture 7 - The Dining Philosophers

BACKGROUND

The "dining philosophers" problem is a classic problem in the field of concurrent programming.

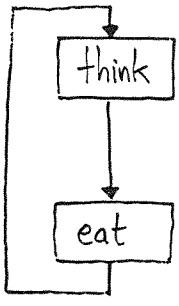
The problem:

- Was posed by Dijkstra in 1971;
- Illustrates a number of common mistakes that can be made in concurrent programs;
- Serves as a useful test-case for designers of new concurrent programming primitives.

We'll look at two solutions to the problem: one using semaphores, and one using a monitor.

THE PROBLEM

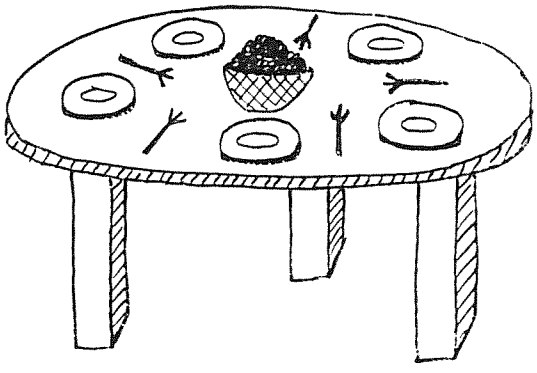
The problem concerns five philosophers, who each engage in an endless cycle of thinking/eating:



Eating happens together at a single table, set with:

- Five plates;
- Five forks;
- One (limitless) heap of spaghetti.

The table is arranged as follows...



There are two "safety properties" for eating:

1. A philosopher needs both forks on either side of his chosen plate in order to eat.

   (This would be more realistic if the spaghetti was rice and the forks were chopsticks!)

2. A philosopher can't pick up both forks simultaneously, but must pick up one at a time.

In addition to the above safety (or correctness) properties, we require two "liveness properties":

① If a philosopher wants to eat, eventually he will get two forks and be able to eat.

For example, there must be no:

- individual starvation! (someone never eats);
- deadlock (no progress can be made);
- livelock (no useful progress can be made).

② If one philosopher wants to eat more frequently than another, the algorithm should not [illegible] this philosopher getting two forks more often.

FIRST ATTEMPT

A simple approach to solving the dining philosophers problem is to use a separate binary semaphore s to control access to each fork:

$s = 0$ means the fork is in use;
$s = 1$ means the fork is free.

The algorithm

Initially, all five fork semaphores are set to 1.

A philosopher wanting to eat waits for use of his left-fork, then waits for use of his right-fork, and finally begins eating. When finished, the philosopher releases the left-fork, then the right-fork.

Assuming a global array fork of binary semaphores (indexed 0 to 4), this algorithm can be translated to the following implementation for philosopher i:

```
procedure philosopher (i : integer);
begin
  repeat
    [think];
    wait (fork [i]);
    wait (fork [(i+1) mod 5]);
    [eat];
    signal (fork [i]);
    signal (fork [(i+1) mod 5])
  forever
end;
```


Assuming a global integer variable i, the semaphores are initialised and the philosophers combined by:

```
begin
  for i:= 0 to 4 do
      fork [i] := 1;
  cobegin
      philosopher (0);
      philosopher (1);
      philosopher (2);
      philosopher (3);
      philosopher (4)
  coend
end;
```

This algorithm satisfies our safety requirements:

① The wait operations suspend a philosopher until he has use of two forks;

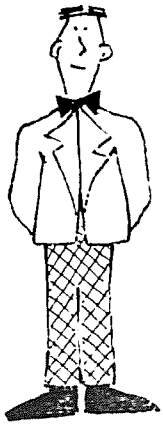
② The forks are "picked up" one at a time;

but not our liveness requirement:

There is nothing in the algorithm that prevents all five philosophers simultaneously "picking up" their left-forks. This results in deadlock, because there are no right-forks left to pick up, and no-one will release their fork.

A SOLUTION USING SEMAPHORES

One way to solve the problem with the previous attempt is to introduce a butler, who ensures that at most 4 from the 5 philosophers sit down at the table at any given time:



In this way, even if 4 philosophers pick up their left-fork simultaneously, there is still one right-fork left to pick up, and hence no deadlock.

The butler can be represented by a general semaphore, whose value is the number of philosophers currently allowed to sit down at the table.

The algorithm

Initially the butler semaphore is set to 4, and all five fork semaphores are set to 1.

A philosopher wanting to eat:

① waits until the butler allows him to sit;

② waits for his left/right fork in turn;

③ eats;

④ releases his left/right fork in turn;

⑤ signals to the butler that he has finished.

Philosopher i is now implemented by:

```
procedure philosopher (i : integer);

begin
  repeat
    [think];
    wait (butler);
    wait (fork [i]);
    wait (fork [(i+1) mod 5]);
    [eat];
    signal (fork [i]);
    signal (fork [(i+1) mod 5]);
    signal (butler)
  forever
end;
```

and the philosophers are combined by...

```
begin
   butler := 4;
   for i:= 0 to 4 do
      fork [i] := 1;
   cobegin
      philosopher (0) ; philosopher (1);
      philosopher (2) ; philosopher (3);
      philosopher (4)
   coend
end;
```

This algorithm satisfies both our requirements:

safety & liveness.

Note: To be fair, the algorithm relies on the suspended processes for butler being a FIFO queue. This ensures that each philosopher eventually gets to eat.

A SOLUTION USING A MONITOR

Another approach to solving the dining philosophers problem is to use a monitor to control access to the five forks.

The monitor encapsulates two variables:

- fork – an array counting the number of forks (0, 1, or 2) available to each philosopher;
- ready – an array of condition variables, used to suspend philosophers until they have two forks,

and provides two procedures:

- request – a philosopher requests two forks;
- release – a philosopher releases his two forks.

The monitor is implemented as follows:

```
monitor forks;

var
   fork  : array [0..4] of integer;
   ready : array [0..4] of condition;
   i, l, r : integer;

procedure request (i: integer);

begin
   if fork [i] ≠ 2 then
      wait (ready [i]);
   l := (i-1) mod 5 ; r := (i+1) mod 5;
   fork [l] := fork [l] - 1;
   fork [r] := fork [r] - 1
end;
```



```
procedure release (i: integer);

begin
   l := (i-1) mod 5 ; r := (i+1) mod 5;
   fork [l] := fork [l] + 1;
   fork [r] := fork [r] + 1;
   if fork [l] = 2 then
      signal (ready [l]);
   if fork [r] = 2 then
      signal (ready [r])
end;

begin
   for i:= 0 to 4 do
      fork [i] := 2
end;
```

Philosopher i is now implemented by:

```
procedure philosopher (i : integer);

begin
  repeat
    [think];
     request (i);
    [eat];
     release (i)
  forever
end;
```

This solution satisfies <u>most</u> of our requirements:

- The request operation <u>suspends</u> a philosopher until he has the use of two forks;
- The forks are "picked up" one at a time;

- Deadlock is not possible;

but can cause <u>starvation</u>, as follows:

philosopher 0 takes forks 0 and 1;

philosopher 2 takes forks 2 and 3;

philosopher 1 <u>suspends</u>, waiting for forks 1 and 2;

philosopher 0 releases forks 0 and 1;

philosopher 0 takes forks 0 and 1;

philosopher 2 releases forks 2 and 3;

philosopher 2 takes forks 2 and 3;

Thus, philosopher 2 never gets access to <u>both</u> forks 1 and 2 <u>at the same</u> time, and <u>starves</u>.

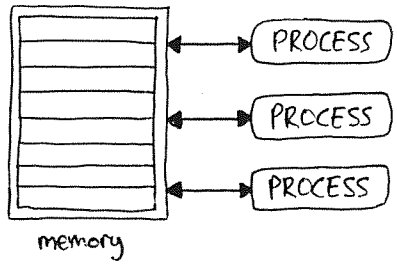
Concepts of Concurrency

Graham Hutton

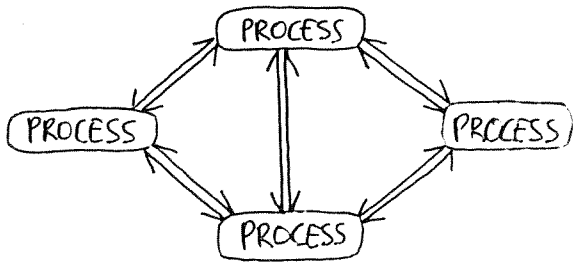
Lecture 8 - CCS I

COMMUNICATION

In the first half of the course we have looked at communication by shared memory:



In the second half of the course we will focus on communication by channels:



CCS

The Calculus of Communicating Systems (CCS):

- Is a simple programming language designed for studying issues in concurrent programming;
- Was developed by Turing-award winner Prof. Robin Milner (Cambridge) in the 1980s;
- Has been a major topic in theoretical computing science research for many years;
- Provides synchronous/unbuffered channels as the basic means of process communication;
- Is a lot of fun!

PROCESSES

Processes in CCS are defined by equations.

Example: the equation

CLOCK = tick

defines a process CLOCK that executes the action "tick" and then simply terminates.

The behaviour of processes can be illustrated using simple pictures such as the following:

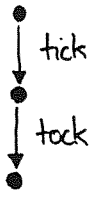
Such pictures are graphs: the nodes represent the "states" of the process, and the edges (labelled by actions) represent the "transitions" between states.

SEQUENCING

The equation

$$\text{CLOCK2} = \text{tick}.\text{tock}$$

defines a process CLOCK2 that executes the action "tick", then executes the action "tock", and then terminates. As a picture, we have:



The equation

$$\text{CLOCK3} = \text{tock}.\text{tick}$$

defines a process with the same set $\{\text{tick}, \text{tock}\}$ of actions, but executes them in the opposite order.

The "." operator indicates sequencing.

In general, if x is an action and P is a process, then $x.P$ is the process that first executes the action x, and then executes the process P.

Note

- To be fully correct we should write

$$\text{CLOCK2} = \text{tick}.\text{tock}.\text{STOP}$$
$$\text{CLOCK3} = \text{tock}.\text{tick}.\text{STOP}$$

  where STOP is the trivial process that can do no actions, but usually we will leave the final STOP in a sequence implicit.

- The "." operator "brackets to the right":

$$x.y.z.P \quad \text{means} \quad x.(y.(z.P))$$

RECURSION

Up to now, all our processes make a finite number of transitions, and then terminate.

Processes that can make an infinite number of transitions can be pictured by allowing loops.

Example

is the process that executes an infinite number of "tick" actions in sequence, and never terminates.

The process can be defined by the equation:

$$\text{CLOCK4} = \text{tick}.\text{CLOCK4}$$

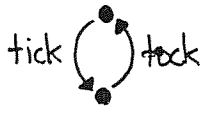
Equations such as CLOCK4 that refer to themselves in their own definition are called recursive.

Example

The recursive equation

$$\text{CLOCK5} = \text{tick}.\text{tock}.\text{CLOCK5}$$

represents the process



that executes a tick action and then a tock action an infinite number of times.

That is, (informally)

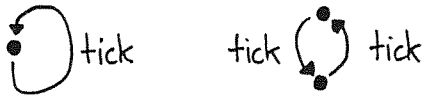
$$\text{CLOCK5} = \text{tick}.\text{tock}.\text{tick}.\text{tock}.\text{tick}.\text{tock}.\ \ldots$$

EQUALITY OF PROCESSES

Consider the processes defined by

$$CLOCK4 = tick.\,CLOCK4$$
$$CLOCK6 = tick.\,tick.\,CLOCK6$$

The processes are physically different:



but in terms of "behaviour" they are equal: both processes execute an infinite number of "tick" actions and never terminate.

Informally,

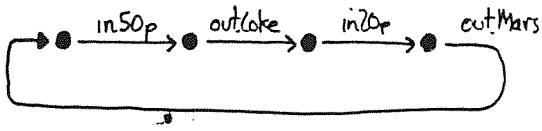
> Two processes are equal if an external observer cannot distinguish them by their actions.

A VENDING MACHINE

A simple vending machine that dispenses coke and mars bars can be defined by:

$$VM1 = in50p.\,outCoke.\,in20p.\,outMars.\,VM1$$

In pictures:



However, the machine is not very flexible:

- It only accepts the exact money;
- The customer has no choice: the machine dispenses coke and mars bars alternately.

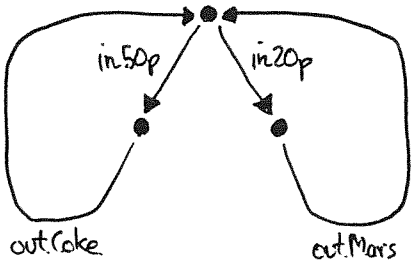
CHOICE

To make a more flexible vending machine, we require some kind of choice operator.

Example: the equation

$$VM2 = (in50p.\,outCoke.\,VM2) + (in20p.\,outMars.\,VM2)$$

defines a process VM2 that repeatedly either inputs 50p and outputs a coke, or inputs 20p and outputs a mars bar. As a picture, we have:



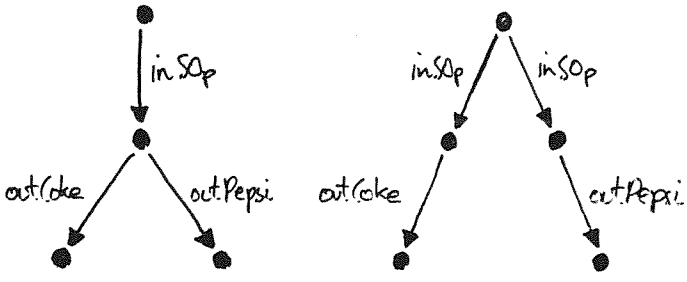
The "+" operator indicates "choice".

The meaning of choice is more subtle than we may first expect. Consider, for example:

$$VM3 = in50p.\,(outCoke + outPepsi)$$

$$VM4 = (in50p.\,outCoke) + (in50p.\,outPepsi)$$

or as pictures,



The two processes look similar (both accept 50p and offer a choice between coke and pepsi), but have very different behaviours:

VM3: the customer decides the purchase

After inserting 50p, if the customer wants a coke the machine can give a coke, and if he wants a pepsi, the machine can give a pepsi;

VM4: the machine decides the purchase

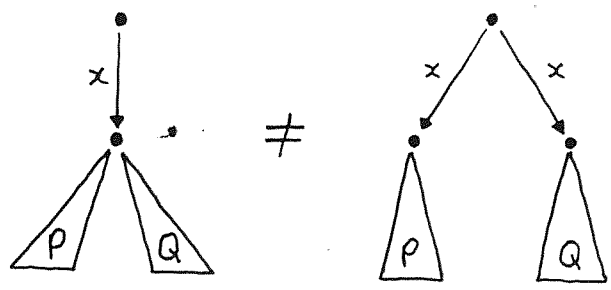
After inserting 50p the machine has the choice of two transitions to make: in the first case, the machine can only output a coke; in the second case it can only output a pepsi.

In general

For an action x and processes P and Q,

$$\boxed{x.(P+Q) \neq (x.P) + (x.Q)}$$

or in pictures,



That is,

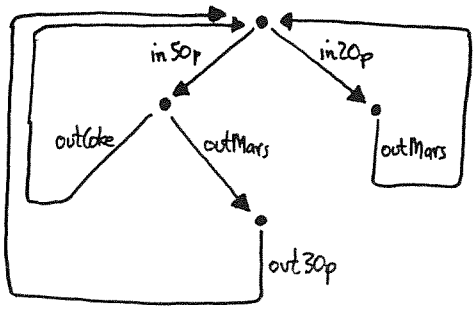
> Sequencing does not distribute over choice: when a choice is made is important.

ANOTHER VENDING MACHINE

A vending machine that can give change if the customer wants to buy a mars bar with 50p is defined by the following equations:

$$VM5 = (in50p.VM6) + (in20p.outMars.VM5)$$
$$VM6 = (outCoke.VM5) + (outMars.out30p.VM5)$$

In pictures, we have:



Note how + is used to ensure that the customer gets the choice of a coke or a mars bar in the case of 50p being input, rather than the machine.

MORE CLOCKS

A clock that ticks repeatedly:

$$CLOCK4 = tick.CLOCK4$$

A clock that can STOP at any time:

$$CLOCK6 = tick.(CLOCK6 + STOP)$$

A clock that ticks or tocks at each cycle:

$$CLOCK7 = (tick.CLOCK7) + (tock.CLOCK7)$$

A clock that ticks each cycle, or tocks each cycle:

$$CLOCK8 = CLOCK9 + CLOCK10$$
$$CLOCK9 = tick.CLOCK9$$
$$CLOCK10 = tock.CLOCK10$$

Concepts of Concurrency

Graham Hutton

Lecture 9 - CCS II

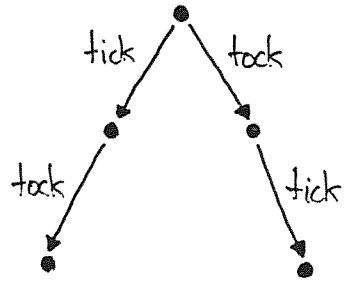
PARALLELISM

The equation

$$\text{CLOCK1} = \text{tick} \mid \text{tock}$$

defines the process CLOCK1 that <u>either</u> executes the action tick and then the action tock, <u>or</u> executes tock and then tick.

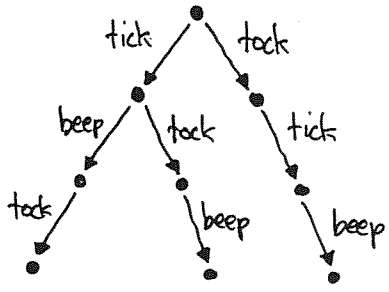
As a picture, we have:



Similarly, the equation

$$\text{CLOCK2} = \text{tick.beep} \mid \text{tock}$$

defines the process



The "|" operator indicates <u>parallelism</u>.

In general, if P and Q are processes, then P|Q is the process that executes P and Q in parallel, <u>interleaving</u> their actions.

NOTE

- Recall that actions are <u>atomic</u>: two actions <u>cannot</u> be executed in parallel.

- If two actions want to execute in parallel using |, an <u>arbiter</u> will force one action to be executed before the other.

  $\boxed{\text{Parallel actions are interleaved.}}$

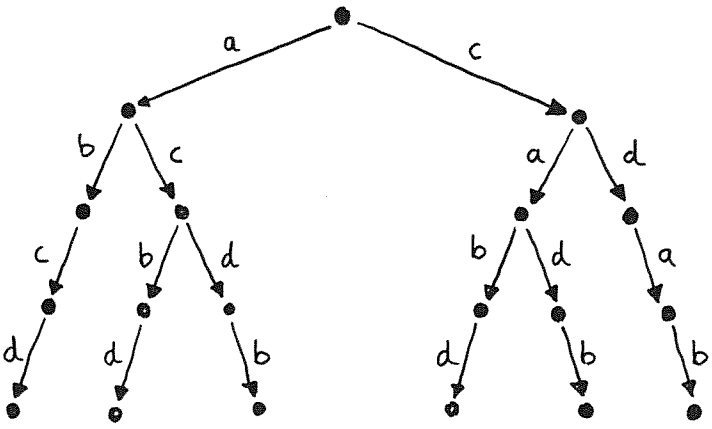
- The use of an arbiter means that parallel processes implicitly involve <u>choice</u>.

EXAMPLE

The equation

$$P1 = a.b \mid c.d$$

defines the process

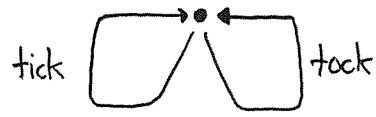


EXAMPLE

Two clocks ticking and tocking in parallel can be defined as follows:

$$CLOCK3 = CLOCK4 \mid CLOCK5$$
$$CLOCK4 = tick.\ CLOCK4$$
$$CLOCK5 = tock.\ CLOCK5$$

As a picture, we have:



Note: this process can also be defined by

$$CLOCK6 = (tick.\ CLOCK6) + (tock.\ CLOCK6)$$

SYNCHRONIZATION

It will often be the case that two parallel processes must synchronize at some point.

Example: a process may want to synchronize certain actions with ticks of a clock.

In CCS, actions can synchronize with coactions, which are indicated by a horizontal bar.

Actions: tick tock inSOp outMars ...

Coactions: $\overline{tick}$ $\overline{tock}$ $\overline{inSOp}$ $\overline{outMars}$ ...

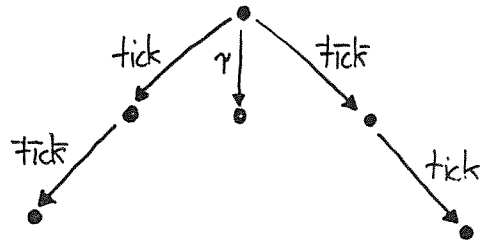
Note: it is sometimes useful to think of actions as output events and coactions as input events.

EXAMPLE

The equation

$$P2 = tick \mid \overline{tick}$$

defines the process



If an action synchronizes with a coaction, both disappear, and are replaced by a special action τ ("tau") indicating a synchronization.

Note: synchronization is not guaranteed.

RESTRICTION

If desired, synchronization can be forced to occur by restricting certain actions from being executed.

Example : the equation

$$P3 = (tick \mid \overline{tick}) \setminus tick$$

defines the process

The "\" operator indicates restriction.

In general, if P is a process and x is an action, then $P \setminus x$ is the process that behaves as P, except that x and $\bar{x}$ cannot execute separately.

EXAMPLE

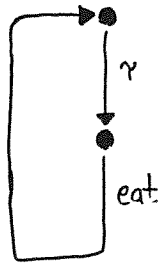
A man that eats every time a clock ticks can be defined by the following equations :

$$MAN = \overline{tick}.\ eat.\ MAN$$

$$CLOCK4 = tick.\ CLOCK4$$

$$P4 = (MAN \mid CLOCK4) \setminus tick$$

As a picture, we have :



CHANNELS

When two parallel processes synchronize, we will often want to transmit an item of data.

Example : after synchronizing one fax machine with another we want to transmit a fax.

In CCS, data is transmitted between processes by allowing actions and coactions to have parameters :

Actions : $a(3)$ $a(15)$ $b(True)$ ...

Coactions : $\bar{a}(x)$ $\bar{a}(y)$ $\bar{b}(z)$...

The parameter of an action is the value to be transmitted ; the parameter of a coaction is a variable to store the transmitted value.

When parameters are used, we can think of actions and coactions as channels rather than events :

$a(3)$ - output value 3 to channel a ;

$\bar{a}(x)$ - input value x from channel a.

EXAMPLE

The equation

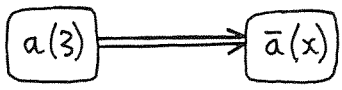
$$P5 = (a(3) \mid \bar{a}(x)) \setminus x$$

defines the process that transmits 3 over channel a. In terms of pictures we have,

indicating that a synchronization has occurred.

PICTURES

The connectivity of parallel processes can be illustrated using simple pictures:



Such pictures are graphs: the nodes are processes, and the edges represent communication channels.

NOTE

- CCS channels are synchronous/unbuffered;
- In a term of the form

  $\bar{a}(x).P$

  the "scope" of variable x is the process P.

EXAMPLE

A simple buffer that can hold a single data value can be defined as follows:

$$BUFF = \overline{in}(x).\,out(x).\,BUFF$$

That is, BUFF is a process with one input channel $\overline{in}$ and one output channel out:

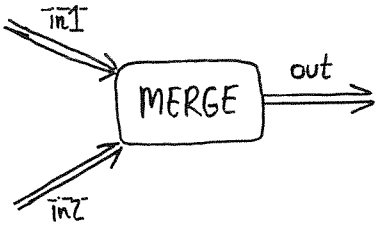
Larger buffers can be made by combining two or more BUFF processes in the appropriate way

EXAMPLE

A simple process that non-deterministically merges two streams of data values can be defined by:

$$P6 = \overline{in1}(x).\,out(x).\,P6$$
$$P7 = \overline{in2}(y).\,out(y).\,P7$$
$$MERGE = P6 \mid P7$$

That is, MERGE is a process with two input channels $\overline{in1}$ and $\overline{in2}$, and one output channel out:



SUMMARY OF NOTATION

$a.P$	–	Sequencing
$P+Q$	–	Choice
$P \mid Q$	–	Parallelism
$P \backslash a$	–	Restriction
$a(v)$	–	Output
$\bar{a}(x)$	–	Input

Concepts of Concurrency

Graham Hutton

Lecture 10 - Semantics of Concurrent Programs

SEMANTICS

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

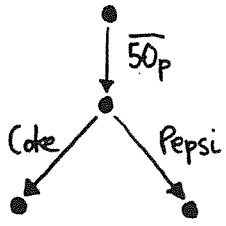
Among other things, a formal semantics:

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

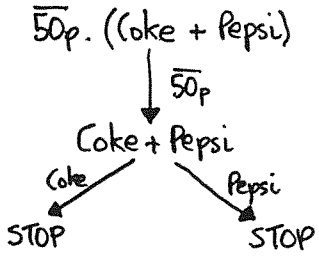
In this lecture we will give a semantics to the CCS programming language.

PICTURES

Up to now the behaviour of CCS programs has been illustrated using simple pictures:



The pictures can be made more informative by writing programs in place of the •'s:



such pictures can be formalised as labelled transition systems.

LABELLED TRANSITION SYSTEMS (LTSs)

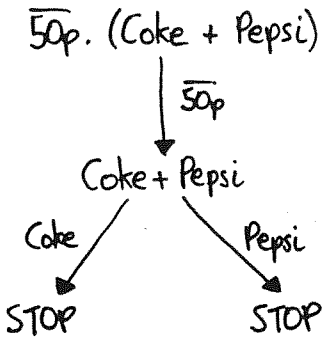
An LTS comprises three components:

1. a set S of "states";
2. a set A of "actions";
3. a transition relation $\rightarrow \subseteq S \times A \times S$.

Notes

- The relation $\rightarrow$ is a set of triples of the form (s, a, s'), where $s, s' \in S$ and $a \in A$.
- We abbreviate $(s, a, s') \in \rightarrow$ by $s \xrightarrow{a} s'$.
- We read $s \xrightarrow{a} s'$ as "from state s we can execute action a to reach state s'."

Example: the picture



is formalised by the following LTS:

S = all CCS programs;

A = all actions, coactions, and τ;

$$\begin{aligned} \rightarrow = \{ & (\overline{50p}.(\text{Coke}+\text{Pepsi}), \overline{50p}, \text{Coke}+\text{Pepsi}), \\ & (\text{Coke}+\text{Pepsi}, \text{Coke}, \text{STOP}), \\ & (\text{Coke}+\text{Pepsi}, \text{Pepsi}, \text{STOP}) \}. \end{aligned}$$

OPERATORS

Up to now, the meaning of CCS operators has been informally specified in English:

- $a.P$ is the process that executes the action a and <u>then</u> behaves as the process P;
- $P+Q$ is the process that can <u>either</u> behave as the process P <u>or</u> as the process Q;
- $P|Q$ is the process that executes processes P and Q in <u>parallel</u>, <u>interleaving</u> their actions, and allowing the processes to <u>synchronize</u>;

⋮

Such statements can be formalised as <u>inference rules</u>.

INFERENCE RULES

In logic we often write

$$\frac{A_1 \quad A_2 \quad \dots \quad A_n}{B} \text{ (RULE)}$$

to mean that RULE is an "<u>inference rule</u>" that allows us to <u>conclude</u> that B holds provided all the <u>assumptions</u> $A_1 \dots A_n$ hold.

Example: (modus ponens from Propositional logic)

$$\frac{A \quad A \Rightarrow B}{B} \text{ (mp)}$$

We can also use inference rules for CCS...

The rule for sequencing

$$\frac{}{a.P \xrightarrow{a} P} \text{ (SEQ)}$$

"Without making any assumptions, the process $a.P$ can execute action a to become process P."

Note: "action" means "action, coaction, or τ."

The rule for STOP

$$\frac{}{\neg(\text{STOP} \xrightarrow{a} P)} \text{ (NULL)}$$

"STOP is the null process that can execute no further actions."

The rules for choice

$$\frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'}\ (CH1) \qquad \frac{Q \xrightarrow{b} Q'}{P+Q \xrightarrow{b} Q'}\ (CH2)$$

"If process P can execute action a to become process P', then the process P+Q can also execute a to become P'. Dually for Q."

The rules for parallelism

$$\frac{P \xrightarrow{a} P'}{P|Q \xrightarrow{a} P'|Q}\ (PAR1) \qquad \frac{Q \xrightarrow{b} Q'}{P|Q \xrightarrow{b} P|Q'}\ (PAR2)$$

"If process P can execute action a to become process P', then the process P|Q can also execute a, to become P'|Q. Dually for Q."

The rule for synchronization

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}\ (SYNC)$$

"If process P can execute action a to become process P', and process Q can execute coaction $\bar{a}$ to become process Q', then process P|Q can execute action τ to become process P'|Q'."

The rule for restriction

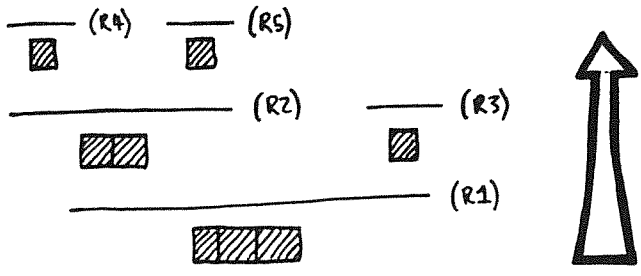
$$\frac{P \xrightarrow{a} P' \quad a, \bar{a} \neq b}{P \backslash b \xrightarrow{a} P' \backslash b}\ (RES)$$

"If process P can execute action a to become process P', and neither a nor $\bar{a}$ equals action b, then process P\b can execute a, to become P'\b."

PROOF TREES

Using inference rules we can construct a <u>proof tree</u> to show that a given statement holds.

Here is an (abstract) example:



Such proof trees are built "<u>bottom up</u>", starting with the statement to be proved, and applying inference rules until there are no assumptions left.

EXAMPLE

The CCS transition

$$(a.P) + Q \xrightarrow{a} P$$

can be proved valid as follows:

$$\frac{\dfrac{}{a.P \xrightarrow{a} P}\ (SEQ)}{(a.P) + Q \xrightarrow{a} P}\ (CH1)$$

That is, we first apply the rule <u>CH1</u> to eliminate the use of +, and the apply the rule <u>SEQ</u> to eliminate the use of ".".

EXAMPLE

The CCS transition

$$((a.P)+Q) \mid \bar{a}.R \xrightarrow{\tau} P|R$$

can be proved valid as follows:

$$\dfrac{\dfrac{\dfrac{}{a.P \xrightarrow{a} P}\,(SEQ)}{(a.P)+Q \xrightarrow{a} P}\,(CH1) \qquad \dfrac{}{\bar{a}.R \xrightarrow{\bar{a}} R}\,(SEQ)}{((a.P)+Q) \mid \bar{a}.R \xrightarrow{\tau} P|R}\,(SYNC)$$

That is, we first apply the rule <u>SYNC</u> to eliminate the $|$ and give two assumptions. The first is proved as on the previous slide, and the second by simply using <u>SEQ</u>.

Concepts of Concurrency

Graham Hutton

Lecture 11 - Tasks in Ada

ADA

- Is a large imperative programming language designed for safety-critical, real-time, systems-level, and concurrent programming;
- Is named after Augusta Ada Byron, (1815-1852), "the worlds first programmer and a collaborator of Charles Babbage;"
- Was developed by a committee of "experts" in the late 1970s / early 1980s for the US Department of Defense;
- Provides synchronous/unbuffered channels as the basic means of process communication.

TASKS

In Ada, processes are called tasks, and are defined in two separate parts:

1. A specification that declares the communication interface of the task;
2. A body that implements the task.

Example

A task P_1 with a single synchronization point S can be specified by:

```
task P1 is
   entry S;
end P1;
```


and implemented by:

```
task body P1 is
begin
   [      ⋮      ];
   accept S;
   [      ⋮      ];
end P1;
```

Note

- We can think of accept S as being like an "input action" $\bar{S}$ in CCS. As in CCS, the corresponding "output action" is written as S;
- Only accepted actions need to be declared in the specification part for a task.

A task P_2 that synchronizes with P_1 on S can now be <u>specified</u> by:

```
task P2 ;
```

and <u>implemented</u> by:

```
task body P2 is
begin
    [      ⋮      ] ;
    S ;
    [      ⋮      ] ;
end;
```

Note

Tasks are <u>automatically</u> executed as concurrent processes; there is no need to combine tasks using something like cobegin ... coend.

SIMULATING SEMAPHORES

It is easy to define an Ada task Sem that simulates a <u>binary semaphore</u>:

```
task Sem is
   entry Wait;
   entry Signal;
end Sem;
```

```
task body Sem is
begin
   loop
      accept Wait;
      accept Signal;
   end loop;
end Sem;
```

In <u>CCS</u> notation, we have:

$$SEM = \overline{wait}.\overline{signal}.SEM$$

MUTUAL EXCLUSION

Using the Sem task, we can now ensure <u>mutual exclusion</u> for two tasks P_1 and P_2.

For example, P_1 is defined by:

```
task P1 ;
```

```
task body P1 is
begin
   loop
      [      ⋮      ] ;
      Wait;
      [ critical region ] ;
      Signal;
      [      ⋮      ] ;
   end loop;
end P1 ;
```


In CCS notation, we have

$$P1 = \boxed{\cdots}.wait.\boxed{crit}.signal.\boxed{\cdots}.P1$$
$$P2 = \boxed{\cdots}.wait.\boxed{crit}.signal.\boxed{\cdots}.P2$$

and the three processes are combined by

$$(SEM \mid P1 \mid P2)\backslash wait\backslash signal$$

Note

- Ada is more <u>verbose</u> than CCS;
- Synchronization in Ada does not need to be forced: it happens <u>automatically</u>;
- Tasks waiting to synchronize with a given <u>accept</u> are held in a <u>FIFO queue</u>.

PARAMETER PASSING

Data can be transmitted between synchronizing Ada tasks by using parameters.

Example

A simple buffer task that can hold a single data value can be specified by:

```
task Buff is
   entry In (X : in Integer);
   entry Out (X : out Integer);
end Buff;
```

where

in denotes an input parameter;
out denotes an output parameter.

The buffer can be implemented by:

```
task body Buff is
   Value : Integer;
begin
   loop
      accept In (X : in Integer) do
         Value := X;
      end In;
      accept Out (X : out Integer) do
         X := Value;
      end Out;
   end loop;
end Buff;
```

In CCS notation, we have:

$$BUFF = \overline{in}(x).\,out(x).\,BUFF$$

Note

- accept statements are like subprograms: they have parameters (with local scope), and a body of code to be executed:

```
accept P ( parameters ) do
   body ;
end P;
```

- accept can both receive and transmit data: in parameters are received before executing the body; out parameters are transmitted after.
- The body is like an atomic action: the synchronizing task is suspended until the body has completed execution.

CHOICE

Tasks can make a choice between a number of possible synchronizations by using "select".

Example

A task that non-deterministically merges two streams of integers can be defined by:

```
task Merge is
   entry In1 (X : in Integer);
   entry In2 (X : in Integer);
   entry Out (X : out Integer);
end Merge;
```

```
task body Merge is
   Value : Integer;
begin
   loop
```

```
    select
        accept In1 (X : in Integer) do
            Value := X;
        end In1;
    or
        accept In2 (X : in Integer) do
            Value := X;
        end In2;
    end select;
    accept Out (X : out Integer) do
        X := Value;
    end Out;
  end loop;
end Merge;
```

In CCS notation, we have

$$MERGE = (\overline{in1}(x).out(x).MERGE) + (\overline{in2}(x).out(x).MERGE)$$

Note: select chooses the first accept that is ready; if more than one is ready, a random choice is made.

GUARDED CHOICE

The choices available when using select can be restricted by using "when".

Example

A task Sem that simulates a general semaphore can be defined as follows:

```
task Sem is
    entry Init (N : in Integer);
    entry Wait;
    entry Signal;
end Sem;
```

```
task body Sem is
    Count : Integer;
begin
```



```
    accept Init (N : in Integer) do
        Count := N;
    end Init;
  loop
    select
      when Count > 0 ⇒
        accept Wait do
            Count := Count - 1;
        end Wait;
    or
        accept Signal do
            Count := Count + 1;
        end Signal;
    end loop;
end Sem;
```

That is, accept Wait can only be executed when the counter is currently positive.

OTHER FEATURES

Ada provides a number of useful commands that can be used as the last alternative of select:

else [statements] ;

Provides a sequence of statements to be executed if no accept is currently ready;

delay t ; [statements] ;

Provides a sequence of statements to be executed if no accept is ready within t seconds;

terminate

Allows the task (and all its calling tasks) to terminate if no accept is currently ready.

Concepts of Concurrency

Graham Hutton

Lecture 12 - Occam I

OCCAM

- Is a simple imperative programming language designed for concurrent programming;
- Was developed from CSP, a theoretical language similar to CCS;
- Is the "assembly code" of the INMOS transputer, a processor chip for building parallel computers and systems;
- Is named after William of Occam, a 14th century philosopher who said: "Entities should not be replicated beyond necessity";
- Provides synchronous/unbuffered channels as the basic means of process communication.

SEQUENCING

Example: the Occam process

```
SEQ
  x := 2
  y := x-1
  z := x+1
```

executes the three assignment statements sequentially, one after the other.

THE LAYOUT RULE

Each statement must be indented slightly with respect to the SEQ keyword, and begin in precisely the same column as each other.

PARALLELISM

Example: the process

```
PAR
  y := x-1
  z := x+1
```

specifies that the two assignment statements can be executed in parallel.

For safety, the same variable cannot be both read and written by different components of a PAR. For example:

```
PAR
  x := 2
  y := x-1
  z := x+1
```

✗

PAR and SEQ can be nested arbitrarily.

For example, the sequential process:

```
SEQ
  x := 2
  y := x-1
  a := 1
  b := a+1
```

can be expressed more efficiently by:

```
PAR
  SEQ
    x := 2
    y := x-1
  SEQ
    a := 1
    b := a+1
```

or

```
SEQ
  PAR
    x := 2
    a := 1
  PAR
    y := x-1
    b := a+1
```

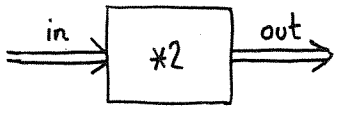

COMMUNICATION

Example: the process

```
SEQ
  in ? x
  out ! x*2
```

assigns the variable x by inputing a value from the channel in, and then outputs the value x*2 to the channel out.

As a picture, we have:



In general,

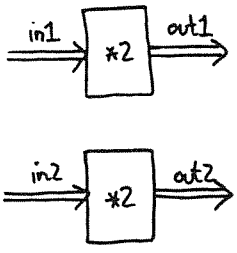
c ? x - input variable x from channel c;

c ! v - output value v to channel c.

Examples

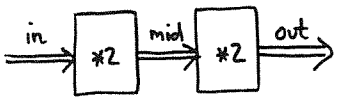
```
PAR
  SEQ
    in1 ? x
    out1 ! x*2
  SEQ
    in2 ? y
    out2 ! y*2
```

≡




```
PAR
  SEQ
    in ? x
    mid ! x*2
  SEQ
    mid ? y
    out ! y*2
```

≡



LOCAL VARIABLES

Example: the process

```
VAR x :
SEQ
  in ? x
  out ! x*2
```

declares a local variable x for use within the two sequential statements.

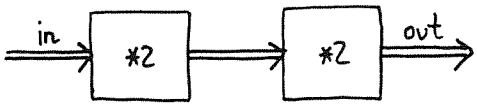
The above process is now equivalent to the following CCS process:

$$\overline{in}(x).\,out(x*2)$$

We can also use <u>local</u> <u>channels</u>:

```
CHAN mid :
PAR
  VAR x :
  SEQ
    in ? x
    mid ! x*2
  VAR y :
  SEQ
    mid ? y
    out ! y*2
```

As a picture, we have:



In CCS notation, we have:

$$\left(\overline{in}(x).mid(x*2) \mid \overline{mid}(y).out(y*2)\right) \setminus mid$$

NAMED PROCESSES

Example: the definition

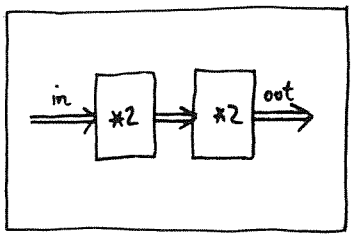
```
PROC double (CHAN input, output) =
  VAR x :
  SEQ
    input ? x
    output ! x*2 :
```

declares a process <u>named</u> <u>double</u>, with two channel parameters, named <u>input</u> and <u>output</u>.

Now, for example, we have:

```
CHAN mid :
PAR
  double (in, mid)
  double (mid, out)
```

≡



UNBOUNDED LOOPS

Example: the process

```
PROC double (CHAN input, output) =
  VAR x :
  SEQ
    input ? x
    WHILE x >= 0
      SEQ
        output ! x*2
        input ? x :
```

<u>repeatedly</u> doubles values from <u>input</u> to <u>output</u>, until the value is negative.

<u>Note</u>: For simplicity, <u>recursive</u> processes are not permitted in Occam.

BOUNDED LOOPS

Example: the process

```
SEQ x = [0 FOR 4]
  out ! x
```

<u>abbreviates</u> the process

```
SEQ
  out ! 0
  out ! 1
  out ! 2
  out ! 3
```

<u>Note</u>

- <u>FOR</u> loop variables <u>cannot</u> <u>be</u> <u>assigned</u> <u>to</u>, and are <u>local</u> within the loop;
- <u>PAR</u> can be used in place of <u>SEQ</u>.

EXAMPLE: SQUARE ROOT

A simple procedure for calculating the square root of a number x is as follows:

1. Take $x/2$ as the initial estimate;
2. Calculate an improved estimate:

   $est' = (est + (x/est))/2$;
3. Repeat step 2 until the accuracy of the estimate is sufficient.

For $x=2$, the first few estimates are:

1
1·5
1·41666667
1·41421569
1·41421356 ← correct

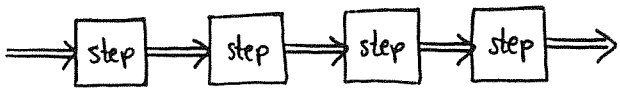
This procedure is known as the Newton-Raphson approximation technique.

It can be translated directly into an Occam process that operates on a stream of values:

```
PROC root (CHAN in, out) =
  WHILE TRUE
    VAR x, est :
    SEQ
      in ? x
      est := x/2
      SEQ i = [0 FOR n]
        est := (est + (x/est))/2
      out ! est :
```

where n is the number of iterations.

The procedure can also be implemented as a pipeline, in which each step in the loop becomes a separate parallel process:



The step process is implemented by:

```
PROC step (CHAN in, out) =
  WHILE TRUE
    VAR x, est :
    SEQ
      in ? x
      in ? est
      out ! x
      out ! (est + (x/est))/2 :
```


We also need a process to generate the first estimate in the pipeline:

```
PROC first (CHAN in, out) =
  WHILE TRUE
    VAR x :
    SEQ
      in ? x
      out ! x
      out ! x/2 :
```

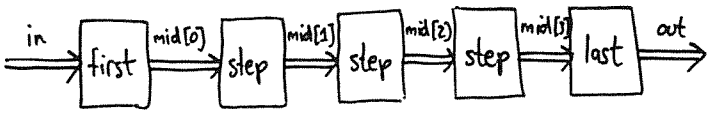
and a process to filter out the propogated x values at the end of the pipeline:

```
PROC last (CHAN in, out) =
  WHILE TRUE
    VAR x, est :
    SEQ
      in ? x
      in ? est
      out ! est :
```

Now we can define a new <u>root</u> process :

```
PROC root (CHAN in, out) =
  CHAN mid [n+1] :
  PAR
    first (in, mid [0])
    PAR i = [0 FOR n]
      step (mid [i], mid [i+1])
    last (mid [n], out) :
```

For example, with n=3 we have



<u>Note</u> : the <u>parallelism</u> in the pipeline means that the <u>throughput</u> of values is <u>much</u> <u>greater</u> than with the original sequential program.