Dr. Mark Humphrys

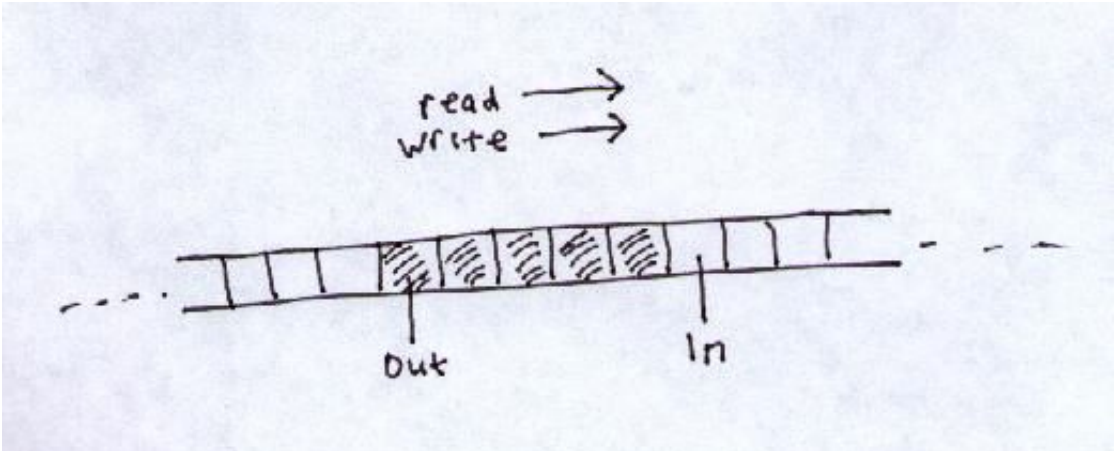
School of Computing. Dublin City University.

Warning to external readers:
These notes (in ~humphrys/Notes) are only meant to be readable as part of a course at DCU.
They are not designed for the use of anyone else.

Process Synchronisation and Deadlocks

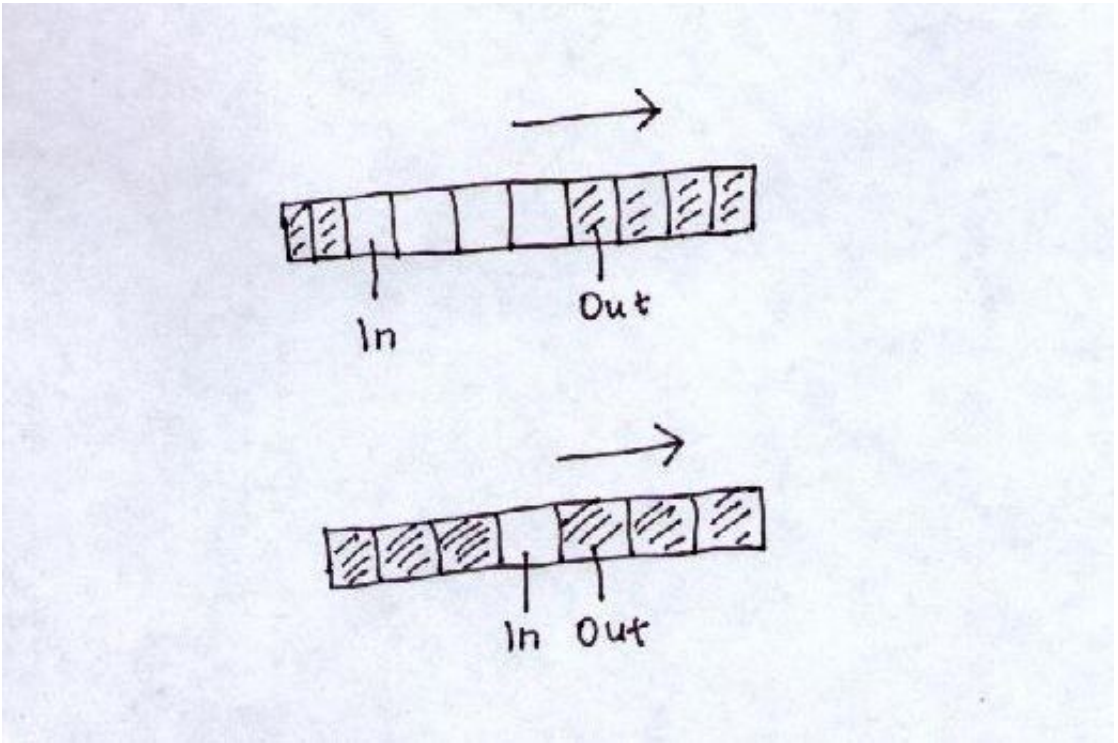
Example - Producer-Consumer problem

Shared memory buffer.
Producer writes next object to In. Then advances In 1 step rightwards.
Consumer reads next object from Out. Then advances Out 1 step rightwards.



Need infinite buffer.

Producer-Consumer with bounded-buffer



Producer:

```
repeat
  produce item in temporary variable
  while ( ( In + 1 ) mod n = Out )
    no-op
  buffer[In] = the new item
  In := ( In + 1 ) mod n
```

Would go on forever advancing rightwards if not for addition of the "mod n".

With this code actually can only ever have `n-1` items.

Consumer:

```
repeat
  while ( In = Out )
    no-op
  read buffer[Out] into temporary variable
  Out := ( Out + 1 ) mod n
  consume the new item
```

Refresher - mod

Some people are unsure what mod is:

```
0 mod 16 = 0
1 mod 16 = 1
..
15 mod 16 = 15
16 mod 16 = 0
17 mod 16 = 1
18 mod 16 = 2
..
31 mod 16 = 15
32 mod 16 = 0
...
```

Pipe

This Producer-Consumer problem is actually "pipe" | in UNIX. Each can work at different speeds.

`sort` is a bad example. Why?

```
prog | sort
```

`grep` is a better example of working in parallel at different speeds:

```
prog | grep "string"
```

`grep` doesn't have to wait until `prog` terminates. It can get to work filtering partial output.

Producer waits - buffer full.
Consumer waits - buffer empty.
In UNIX pipe, sometimes one might wait, sometimes other.

Producer may do lots of calculation, then produce 1 line of output, lots more calculation (this is *not* a CPU burst - calculation may involve background I/O, e.g. file I/O), then 1 more line of output, and so on. Producer may produce at different speeds depending on what it is producing. May take more time to construct some outputs than others. Also, may take more time to consume some than others. Consumer may consume at different speeds throughout, sometimes fast, sometimes slow.

Note: Pipe in UNIX is technically a file with no name. In UNIX, we cache file data in memory (see later), and pipe is small file (e.g. 4 K), so typically the entire read-write buffer is actually in memory (i.e. it is actually shared memory).

Process Synchronisation and Deadlocks

Producer:

```
(write item)
counter++;
```

Consumer:

```
(read item)
counter--;
```

Surely we can run these in any order and it will work, and counter will stay up to date?

Not necessarily - it depends when the interrupts arrive.

The problem is that the HLL are not atomic (indivisible) instructions. The above HLL translates to Assembly like the following. Let's say the variable "counter" is stored at memory location 100. Then the Producer implements:

```
(write item)
MOV     AX, [100]               ; read memory
INC     AX
MOV     [100], AX               ; write memory
```

Consumer:

```
(read item)
MOV     BX, [100]
DEC     BX
MOV     [100], BX
```

(It might also be AX, but we can *regard* it as using a different register because the context switch saves/restores registers).

Now, consider this pattern of interrupts. Say counter=5 beforehand. After running one produce and one consume in any order, counter should still be 5:

```
MOV     AX, [100]       ; AX=5
INC     AX              ; AX=6
(interrupt)
MOV     BX, [100]       ; BX=5
DEC     BX              ; BX=4
(interrupt)
MOV     [100], AX       ; counter=6
(interrupt)
MOV     [100], BX       ; counter=4
```

Counter=4!
And interrupts could just as easily have come in so that the last 2 statements were swapped round and counter=6.

Critical Section

A program, *by definition*, will consist of a pattern of alternating:

```
code that accesses shared data
code that doesn't
shared
non-shared
shared
non-shared
...
```

Call code that accesses shared data a *critical section*. Then can view every process as having structure:

```
repeat
{
  critical section
  remainder section
}
```

Of course, the program has to help us out. Each program looks like:

```
repeat
{
  code to mark entry to critical section ("entry section")
    critical section
  code to mark exit from critical section ("exit section")
    remainder section
}
```

Restricting interrupts

When in critical section, no other process in my group of co-operating processes is allowed interrupt.

Question - why not simply *no* interrupts?
Answer - OS still has to be able to interrupt.

Also, *other* processes (not involved with this memory space) can interrupt at will. Normal CPU scheduling still goes on. You can't hog the CPU just because you've written 2 co-operating processes.

And in fact, other processes in my group can interrupt (normal CPU scheduling) if they are in their *remainder* section.

Criteria for solution to critical-section problem for *n* processes running at possibly different speeds

Problem with any solution for *n* processes:
What if *many* processes are waiting to go into critical? Who goes first? After a go, do you go back to end of queue? But what if you're ready again before processes ahead of you in the queue are ready?

Criteria we want:

1. *Mutual exclusion* - If one process in critical section, then no other (sister) process can be in theirs.
2. *Progress* - If no process currently in their critical section, and one or more want to get in, then selection of a process to get in cannot be postponed indefinitely.
3. *Bounded waiting* - A bound exists on number of times other processes can enter critical while I am waiting.

Producer-Consumer (pipe) Solution with 2 processes - Attempt 1

More shared data: `turn = i` - it's Process i's turn.

Process P_0:

```
repeat
   wait until turn=0
       Critical section
   turn=1
       Remainder section
```

Process P_1:

```
repeat
   wait until turn=1
       Critical section
   turn=0
       Remainder section
```

Mutual exclusion ok.
Bounded waiting ok - each only waits at most 1 go.

Progress not good - each *has* to wait 1 go. P_0 gone into its (long) remainder, P_1 executes critical and finishes its (short) remainder long before P_0, but still has to wait for P_0 to finish and do critical before it can again. Strict alternation not necessarily good - Buffer is actually pointless, since never used! Only ever use 1 space of it.

If P_1 terminates, can P_0 still finish? - Use strategy of "Giving up your go" before remainder rather than claiming your go. exit() is in remainder section.

What if exit() is in critical section? - It can't be, by definition.

Attempt 2

New data: `flag[i] = true` - Process i is ready to go into critical.

Process P_0:

```
repeat
   flag[0] := true
   while flag[1] do no-op
       Critical section
   flag[0] := false
       Remainder section
```

Process P_1:

```
repeat
   flag[1] := true
   while flag[0] do no-op
       Critical section
   flag[1] := false
       Remainder section
```

Doesn't work at all. Both flags set to true at start. "After you." "No, after you." "I insist." etc.

Attempt 3

Separation of flag and turn.
flag says I'm ready, turn says "whose turn it is".
One of these turns will last.

Process P_i, where j refers to "the other process":

```
repeat
{
  flag[i] := true
  turn := j
  while ( flag[j] and turn=j ) do no-op

    critical

  flag[i] := false

    remainder
}
```

"flag" maintains a truth about the world - that I am at start/end of critical. "turn" is not *actually* whose turn it is. It is just a variable for solving conflict if two processes are ready to go into critical. They all give up their turns so that one will win and go ahead.

e.g. flags both true, turn=1, turn=0 lasts, P_0 runs into critical, P_1 waits.
Eventually P_0 finishes critical, $flag_0$=false, P_1 now runs critical, even though turn is still 0.
Doesn't matter what turn is, each can run critical so long as other flag is false. Can run at different speeds.
If other flag is true, then other one is either *in* critical (in which case it will exit, you wait until then) or at start of critical (in which case, you both resolve conflict with turn).

Semaphore

Atomic instructions provided by OS to make it *easy* to write entry and exit sections.

wait(S):

```
while S <= 0 do no-op
S--
```

signal(S):

```
S++
```

Wait if S negative or 0.
Finally someone else's signal makes S=1.
Wait loop ends, sets S=0 before it leaves wait()

Multiple-processes Mutual Exclusion with

Semaphores

Each process is simply:

```
repeat
  wait(mutex)
    critical
  signal(mutex)
    remainder
```

At start mutex=1,
one of processes sees it, takes it, mutex=0, goes into critical,
other processes see mutex=0,
first process finally finishes, signals mutex=1, goes into its remainder,
somebody sees mutex=1, takes it, sets mutex=0 before goes into critical,
and so on.

Who gets to go next is pretty much random (perhaps unfair). It might be nice to have a FIFO queue.

Question - *Why* is who goes next pretty much random?

Busy waiting (looping with no-op)

Busy waiting - Keeps getting scheduled by CPU. May run for days, doing nothing except interfering with running of other programs.

To avoid busy waiting, wait() system call sends process voluntarily out of Ready queue and into Wait state. signal() sends wakeup to a process on the Wait list, restores it to the Ready queue.

New semaphore definition to use wakeup() instead of busy wait:

wait(S):

```
S--
if S < 0
  suspend this process
```

signal(S):

```
S++
if S <= 0
  wakeup 1 process at head of queue
```

S = - (no. of processes waiting).
S = 1 means no process in critical, none waiting.
S = 0 means 1 in critical, none waiting.
S = -1, 1 in critical, 1 waiting.
S = -2, 1 in critical, 2 waiting.
etc.

In signal(), someone just finished critical, so if S < 0, wake one up.
But we did S++ already, so if S <= 0, wake one up.

In wait(), if S < 0, we have to wait along with others,
if S=0, we are first to wait,
i.e. if S <= 0 we wait,
but we did the S-- already, so if S < 0 we wait.

Consider multiple processes:

```
start S=1,
1st process, S=0, carries on into its critical,
2nd process, S=-1, wait,
next process, S=-2, wait,
...
S=-(n-1),
finally, 1st process finishes critical,
 signal(), wakes up a single process,
 sets S=-(n-2), and goes into its remainder
```

Deadlocks

Processes end up waiting for each other. e.g. If more than 1 semaphore:

Process P_0:

```
wait(S)
wait(Q)
...
signal(S)
signal(Q)
```

Process P_1:

```
wait(Q)
wait(S)
...
signal(Q)
signal(S)
```

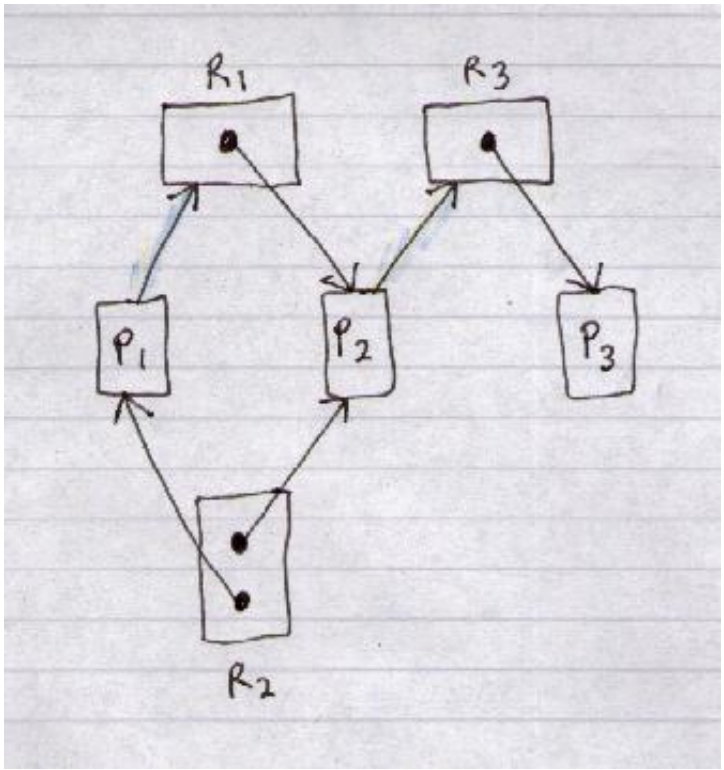
In general, a deadlock involves process *holding* a resource, will only release it if it gets another resource, which is being held by other process, etc.

Deadlock definition

More precisely, deadlock arises if there is:

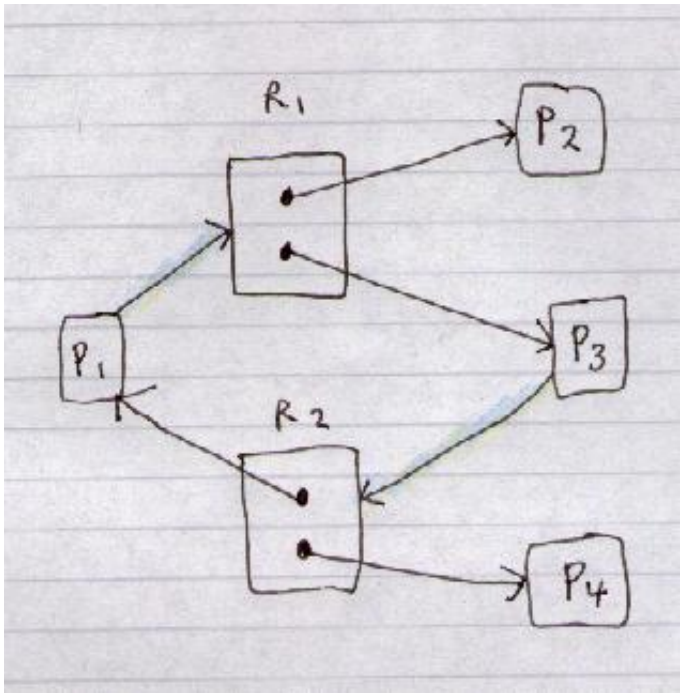
1. *Mutual exclusion* - Only 1 process at a time can hold a resource.
2. *No preemption* - A resource can only be released voluntarily, i.e. when the process has completed its code.
3. *Hold and wait* - There is a process holding at least 1 resource and waiting to acquire additional resources held by others.
4. *Circular wait* - There exists a set of processes s.t. P_0 is waiting for a resource held by P_1, P_1 for P_2, ..., P_{n-1} for P_n and P_n for P_0

Hold and Wait, but no cycle:



Lines from resource R to process P - an instance of R is allocated to P.
Lines from P *to* R - P is waiting for an instance of R.

Cycle, but no deadlock:



Let D = Deadlock, C = cycle:

D implies C
C does not imply D

(No C) implies (No D)
(No D) does not imply (No C)

Deadlock is proper subset of cycles.

If C plus only *one* instance of each resource, then D.

OS policy on deadlocks

UNIX ignores deadlocks.

- OS is coded so that normal operation won't cause deadlock.
- If user gets his co-operating processes into a deadlock, that's his problem.
- Similar to an infinite loop - users' processes have hung, rest of CPU scheduling (other users etc.) can carry on as normal.
- One proviso maybe - His processes may be holding resources that other processes want. - The circle of deadlock can *expand*.

Deadlock Prevention - *Static* rules/guidelines to follow, which (hopefully) will prevent deadlock. e.g. Tell me when you *start* how much resources you will need, and hold them for whole execution. Why won't this work?

Deadlock Avoidance - OS queries what is going on and decides *dynamically* if next request will cause deadlock (or danger of deadlock).

Processes are 1-way.

Remember - Processes are 1-way. You can't "rewind" processes. You must let them run forward to finish.

```
open(file1)
do lots of calculations
write some to file1
 -----
open(file2)
do more calculations
write to file2
write more to file1
close(file2)
close(file1)
```

Can't interrupt at the point marked and say "Please close file1 because other program, which is holding file2, needs file1 to finish". Can't do this because file1 would be closed in an incomplete state, perhaps with inconsistent or partial data. Would be better if file1 had never been opened.

Can't return file1 to original state, reset instruction counter to before "open(file1)", and continue, because global variables may have changed since then, and we have no memory of what they used to be.

Can't terminate process and close file1 because it is in inconsistent state. Get *correctness* problems unless we allow process to finish naturally.

Can only freeze and wait for file2 to become free.

Processes that are *designed* to be (partially) 2-way

An Exception to the above - We may explicitly *design* processes so they keep regular *checkpoints* to which they can be rolled back if necessary (i.e. they periodically save their entire state), e.g. Many processes accessing a database. *We* design the co-operating processes, so we can design them

to be rolled back if necessary.

Even these, though, can be rolled back *only* to certain pre-defined points, not to any arbitrary point.

By the way, it seems to me that in general rolling back to any arbitrary point is impossible. The program would need to save its entire state after every instruction - but the process of saving the state is *itself* a program that would need to be rolled back, and so on.

Deadlock Avoidance - Safe and Unsafe states

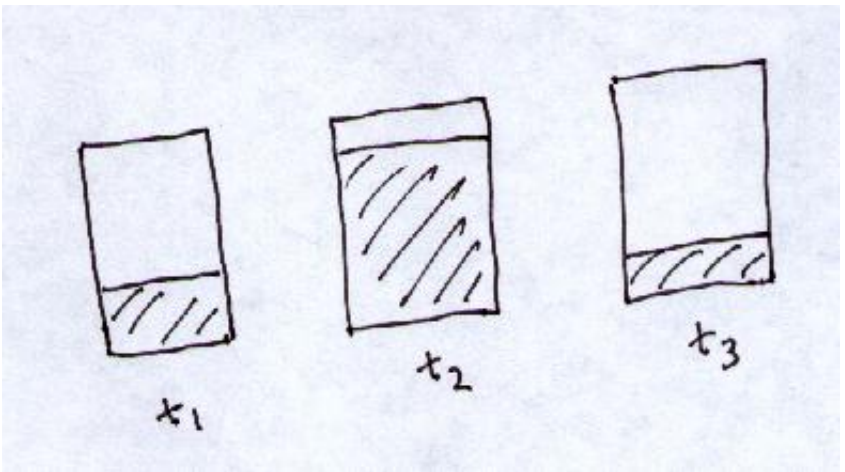
Safe state - there exists a possible order in which they can finish without any processes backing out (releasing resources). We will enforce this finishing order if necessary to avoid deadlock. We may not have to enforce it - the processes may be releasing resources soon anyway.

Unsafe - there is no possible finishing order unless one of the processes backs down themselves (releases resources). We could still avoid deadlock, but now we have to depend on the processes themselves.

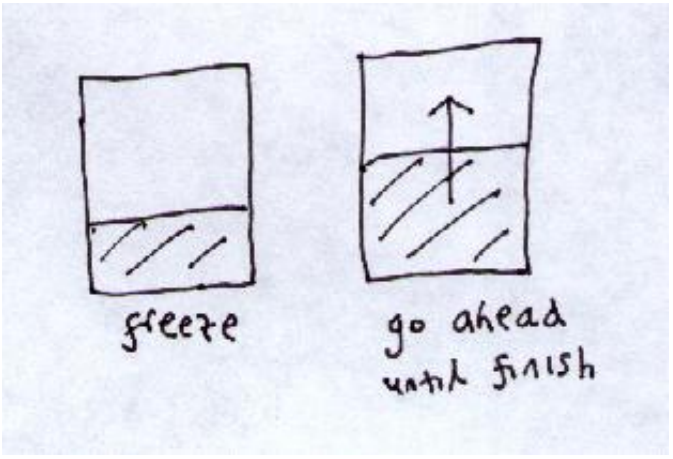
Deadlock Avoidance algorithm - Simply never let it go into an unsafe state.

Example - Memory Resources

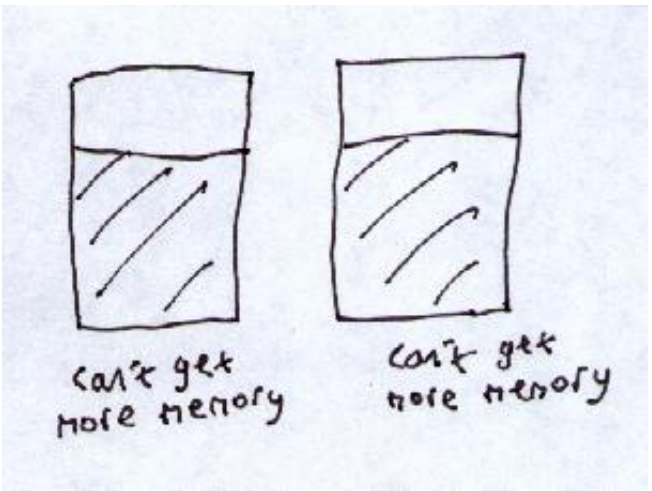
Process needs memory at different times in its execution. It acquires and releases memory at different points:



Given 2 processes, can they both finish? Remember a process can't back out. They can only *freeze* and wait for other to finish. We may have to freeze one (deny its requests for more memory, even though more memory is available) to make sure that at least one can finish:



to avoid this:



To do this, we need to know *maximum* amount of memory the process will need over entire course of execution.

This may be impossible with memory, but may be possible with other shared resources.

Exercise - Consider a system consisting of 4 resources of the same type that are shared by 3 processes, each of which needs at most 2 resources. Show that the system is deadlock-free.

Answer - We can't have a state where all 4 resources held and 1 process can't finish. If all 4 held, one must have 2. So it can finish. And if 1 finishes all can finish.

Exercise - What if we don't *know* the maximum amount of resources the process will need before terminating?

Then *any* resource allocation may be unsafe. Consider 2 processes that will each want *all* the resources before termination. Then assigning even 1 resource each gets us into an unsafe state.

Resource-allocation matrices

Define matrix to show what is allocated where, max that each process will need, and hence the extra resources it will need that it does not have now.
e.g. There are 10 instances of resource type A, 5 of B, 7 of C:

```
         Allocated   Max     Need
          A B C     A B C   A B C

    P0    0 1 0     7 5 3   7 4 3
    P1    2 0 0     3 2 2   1 2 2
    P2    3 0 2     9 0 2   6 0 0
    P3    2 1 1     2 2 2   0 1 1
    P4    0 0 2     4 3 3   4 3 1
```

and currently free:

```
     Free
    A B C
    3 3 2
```

Show that there exists a finishing order P1,P3,P4,P2,P0 (and another one P3,P1,..).
i.e. This system is in a safe state.

P1 increases its allocation to:

```
         Allocated   Max     Need
          A B C     A B C   A B C

    P0    0 1 0     7 5 3   7 4 3
    P1    3 0 2     3 2 2   0 2 0
    P2    3 0 2     9 0 2   6 0 0
    P3    2 1 1     2 2 2   0 1 1
    P4    0 0 2     4 3 3   4 3 1
```

and hence the free list:

```
     Free
    A B C
    2 3 0
```

This is still a safe state. - Why?

Now, can request for (0,2,0) by P0 be granted?

Summary of Process Synchronisation

"Stranger" process:

1. Many owners.
2. Could be hostile / incompetent. May grab all resources (OS will stop them).
3. Don't share data (memory).
4. No correctness problem - Any pattern of interrupts OK.
5. Ch.6 irrelevant, runs in its own world.

"Friend" process:

1. 1 owner, 1 programmer.
2. Assumed to be friendly. Will do wakeup().
3. Share data (memory / variables).
4. Correctness problem, depending on pattern of interrupts.
5. Need OS support (atomic instructions) to maintain correctness. But helped by fact that assumed friendly (hence casually say that P_i will signal wakeup).