DEADLOCK

In a deadlocked system, two or more processes cannot proceed because each wants some resource held by another. If this happens, there is no question of resources being poorly allocated when they become available, as in cases of starvation; they just don't become available. (There used to be another name for deadlock – the "deadly embrace". We may be thankful that it appears to have fallen into disuse.) To break a deadlock, at least one process must be forced to surrender a physical resource, or to provide an abstract resource. In either case, a process must be forced to do something which it would not do if it followed its programme, so some sort of damage is inevitable.

Because deadlock is a phenomenon of competing processes, it did not become significant for computing until multiprogramming methods began to develop. Once observed, though, it was soon recognised as something to be avoided if possible. As compared with other examples from the material world, deadlock in computer systems is insidious; just as in the case of livelock (which wasn't really recognised as a significant problem until much later than deadlock), a deadlocked process is doing nothing remarkable to distinguish it from any other process, resources might be abstract entities like locks or semaphores so they might be invisible too, and other processes might continue to function giving the impression that all is well. Generally, it was left to operators to notice deadlock and to try to get out of it as best they could.

It is fair to remark that, despite the material to be discussed in this chapter, not much has changed. Or, perhaps better, only one factor has changed a lot : we are now so much better endowed with resources that deadlock is much less likely to happen. As we have seen, the decreasing cost of hardware made it possible to run interactive systems in which it is essential to maintain a surplus of machinery to guarantee that peak demands can be met, and systems are therefore rarely run at anywhere near the limits of their capacities.

As well as this happy circumstance, our understanding of deadlock has developed considerably, and a few methods are known which can help to stave off deadlock even if they can't necessarily abolish it completely. These ideas are based in one way or another on the conditions which lead to deadlock, so that is where we'll start.

DEADLOCK AND RESOURCES.

Any sort, or sorts, of resource which is, or are, in limited supply can contribute to deadlock. In this context, a resource is anything which a process must acquire before it can continue. Resources come in two main varieties :

- **Reusable resources**, generally including material objects such as hardware, but also including data when access is limited. They usually fixed in number. Some examples :

  Memory, processors, devices.
  Semaphores, locks, critical sections.
  Things in files (depending on access restrictions).
  Eating utensils (if you happen to be philosophers).

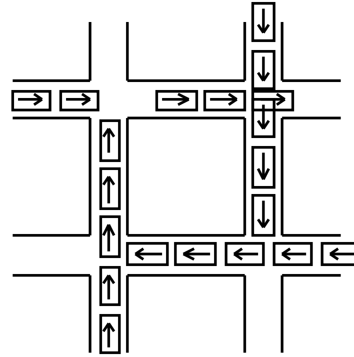
- **Consumable resources**, usually data of one sort or another, which can be constructed and destroyed. Examples :

  Messages, interrupts, data input.
  Things in streams.

CONDITIONS FOR DEADLOCK OF SEVERAL PROCESSES.

These conditions were presented by J.W. Havender in 1968. All these conditions are necessarily satisfied simultaneously in a deadlock. There are four conditions, and we'll illustrate them by inspecting the traffic analogy.

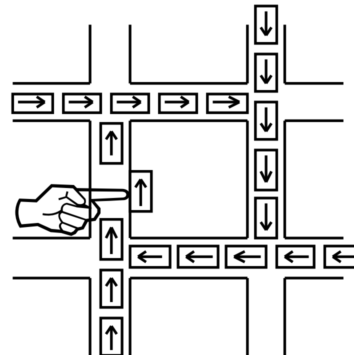
Resources cannot be shared.



If two vehicles could simultaneously occupy the same piece of road, the traffic deadlock could be resolved; the diagram illustrates one way. Notice that it's enough for *one* of the junctions to be sharable – we don't require that all the road should be miraculous, nor even all the junctions.

This is only possible if one resource can in some sense serve two processes at once.

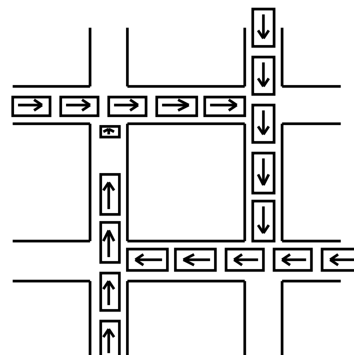
Only the owner of a resource can release it.



If there were some way to take away the resource used by one vehicle for a while, and give it to another, we could resolve the traffic deadlock.

This is only possible if a process is not in some sense using the resource it owns while it is waiting.

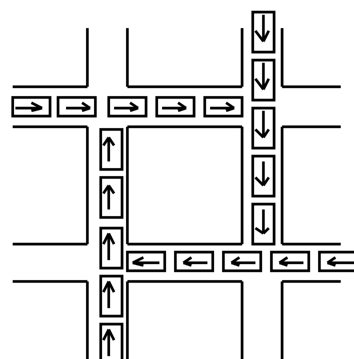
Processes hold resource A while requesting resource B.



If the vehicle didn't need road space while it was waiting, our traffic problems would be solved.

This is quite similar to the previous case; the difference is that here we consider the possibility of the process voluntarily relinquishing its hold on the resource, whereas formerly we wanted some external agency to force it to yield the resource.

There is a circular list of process waiting for resource owned by process ....



This is almost a definition of deadlock rather than a condition, but it's certainly a condition to the extent that if we can make it false we've eliminated the deadlock.

PREVENTING DEADLOCK.

If each of the conditions above is necessarily satisfied in a deadlock, then any procedure, rule, device, stratagem, or what have you which guarantees that any one of them cannot be satisfied has prevented deadlock in the area to which it applies. This leads us to four groups of deadlock prevention techniques, one group corresponding to each rule.

- **Make resources sharable.** Substitute "virtual" devices for real physical ones. Strictly, of course, the usual implementation of virtual devices doesn't really share an existing device – it replaces one or a few physical devices with a much larger number of imaginary devices which work just as well. There are very few devices which can really be shared. One reason is that most devices have some sort of

internal state which must be preserved if the device is to work. The state of a printer, for example, includes the position of the paper and the marks on it; you can't share a printer unless you can remove the paper and put it back in exactly the same position. Sharing is usually impossible if the device concerned is required to operate in real time, while the process is running, for then it might be required to do different things for different processes simultaneously – so it won't work for a terminal (unless you think of windows as virtual terminals). It does work well for a number of other physical devices, and has been used for a long time in spooling systems. The traditional virtual devices are card readers and line printers, but the same approach works for other devices of the same general type. Virtual memory falls under this heading, too. Packet switching, as used in data communications, is a way of getting virtual telephone lines. Virtual discs, at least as used in the IBM VM operating systems, are different; they are partitions of the real disc, and are intended to prevent contention for the real disc space by restricting each process to its own disc area.

- **Give processes all their resources at once.** You can only manage this if you know upper limits to what resources a process will require before it starts. You have to include anything that just *might* be required, and allow for the most extravagant possible use of resources like memory and disc space. It doesn't usually work at all for resources like messages, which might not exist when the process starts, and it cannot possibly work for resources which must be passed back and forth between processes as part of their execution – such as locks, semaphores, and critical sections.

- **Make waiting processes surrender their resources.** This is feasible only with resources which do not have any significant internal state, or for which the state can be saved and restored again. The prime example is the processor; another is memory, as used in virtual memory systems.

- **Impose a standard order on requests for resources.** This is good when you can do it, and is one of the few techniques that do have some utility in dealing with the more abstract resources. You allocate numbers to the resources – in principle, arbitrarily, but the order affects the efficiency of the algorithm – and require that processes ask for resources in strictly increasing numerical order. Its major defect is that you might be forced to acquire some resource with a low number just in case you want it, even though you might know that you will often only require a resource with a higher number. Notice that all devices allocated the same number must be effectively identical. (PROOF : The resources owned by a process must always have numbers less than a resource which it is trying to acquire. Therefore, if a process ***P*** requires resource *N*, it is certain that no process owning resource *N* is waiting for anything owned by ***P***. Q.E.D.)

These methods ensure that the system can never get into a state where deadlock is a possibility. Provided that you stick to the rules, you can start as many processes as you like, and the combination is guaranteed to be free of deadlock. The methods work where they're feasible, but none of them works for all resources, and they tend to be excessively cautious about resource allocation.

AVOIDING DEADLOCK.

If we want a less restrictive system, we must relax some conditions. One possibility is to accept that we shall have to watch what processes do as they operate, and guard against deadlock situations developing. If we follow this path, we can start rather more processes, but pay the cost of continuous monitoring. Most of the time, it pays off. In order to construct the watchdog procedure, we must find a way to detect the potential for deadlock well before it happens, so that we can prevent any action which might move the system into a state from which it could be forced into deadlock.

We define the *system state* (or, better, part of it, but it's the part we're interested in here) as a list of active processes and the resources which they hold. The state will

change from time to time as the processes start and stop, and acquire or release resources. Each state change is initiated by a request from a process, but only completed when the system responds to the request, so we can control the movement of the system among its possible states by deciding whether or not to grant a request for a resource. Requests to release resources are always safe, so we'll always accept them, but requests for resource allocation must be handled with more care.

What we're trying to do is avoid states in which requests for resources can fall into the pattern of deadlock. To do so, we must first identify the states in question, then devise a strategy which ensures that the system won't get into them. As we don't know what the sequence of requests will be, we have to play safe, and only permit allocations which result in transitions to states which must be safe under all possible circumstances. (The "give-all-the-resources-at-once" strategy does the same thing by ensuring that, once a process has started, there won't be any requests; now we're trying to be rather more flexible.)

In fact, as it's stated above, the problem is impossible to solve. If we know nothing about the processes beforehand, it could be that every process we start will immediately request twice as much of every resource as we have, so that deadlock is inevitable. We must therefore have some prior information; when giving each process all the resources it required right at the start, we had to know the upper limits to the demands, so perhaps that's a reasonable constraint here too. We can still not know the really critical information – that's the sequence of requests at all times – but if we know the maxima then we also know that in some circumstances (when it already has its maximum demand for resource X) a process won't ask for any more X. That's the trick which makes the method work.

Coming back to the problem, then, we want to make sure that a request for a resource is only granted if we are certain that the state of the system after granting the request will be safe. What does that mean ? We identify *unsafe* states as states from which there is any *possible* sequence of requests that would lead to deadlock, and we make sure that the new state is not one of those.

AN EXAMPLE.

Think of a model of the system in which each action which allocates or releases a resource marks a transition from one state to another. Then we can identify the state of a system by listing the amount of resource of each type held by each process. For example, suppose we have two processes, ***P*** and ***Q***, and two resources, *R* and *S*. Then we could describe a system state as a combination of the states of the two processes by writing something like

$$\boldsymbol{P} \text{ has } (3R, 2S); \ \boldsymbol{Q} \text{ has } (0R, 1S).$$

That notation can obviously be extended to describe systems with any number of processes and resources.

Suppose the system owns three *R* resources and four *S* resources; then we can enumerate all *imaginable* states of a process from ($0R$, $0S$) to ($3R$, $4S$). It might be that for some process we know that the maximum demand is ($2R$, $1S$), so that not all the states are *relevant* for that process; that doesn't matter for the moment, but we'll come back to maximum demands later.

We can also enumerate all *conceivable* states of a system by combining all the *imaginable* states of all the processes in all possible ways. Some of these conceivable states are obviously absurd, as they allocate more of some resource than the system owns; but it's convenient to keep them there for the time being.

What sorts of state transition are possible in this system ? One can imagine all manner of elaborate changes, but any change can always be thought of as a sequence of single allocations and releases concerning individual processes and individual resources, so we can reasonably suppose that the state described above can change in exactly seven ways so far as the conceivable system states are concerned – though there might in reality

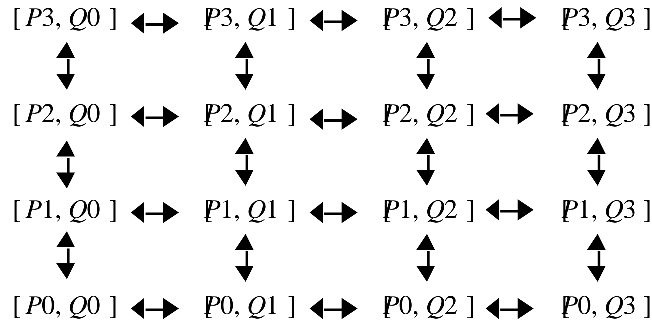
be fewer possibilities because some of the conceivable states aren't *attainable* in practice, or the actual requirement of a process for some resource might be less than that allocated to it in some of the configurations.

Now, that representation of the system state turns out to be rather inconvenient for our purposes, because to draw the state transitions conveniently we would need four dimensions. Alternatively, and more conveniently for our immediate purposes, we can write a description which emphasises the resources :

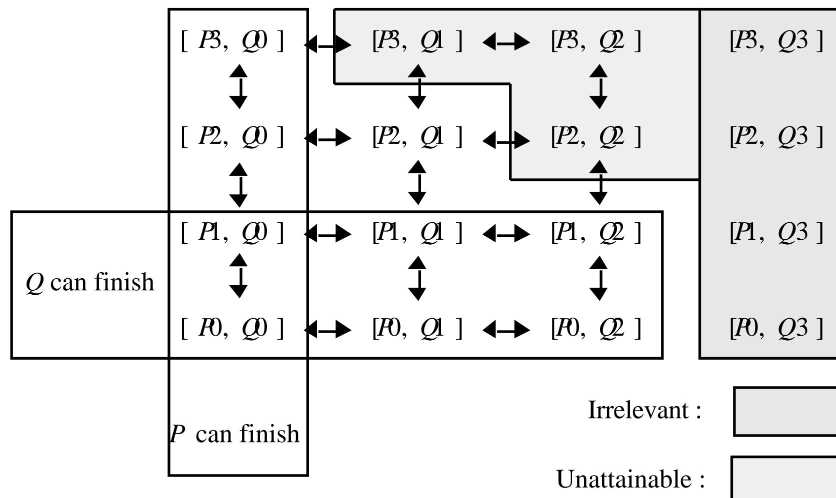
R belongs to [***P*** 3, ***Q*** 0]; *S* belongs to [***P*** 2, ***Q*** 1].

The two descriptions are obviously precisely equivalent. (These notations, like the vocabulary we are using, are in no sense standard; we've made them up.) This description is more convenient because it collects into one place all resources of one type, so we can see them all together. The state transitions now look like a sequence of changes, each affecting exactly one of the resources; in the discussion which follows, we shall consider only a single resource, so remember that in reality the same arguments should be applied to all resources in the system. It should be clear, though, that the separate arguments are independent, so we won't be cheating in any way.

Consider, then, the resource *R*. We are supposing that there are just three *R* resources in the system, so this table describes all conceivable *R* allocations and transitions :



We have seen that there are two reasons why some of these states might be inaccessible. First, all the states [***P****n*, ***Q****m*] with $n + m > 3$ are *unattainable* , because there is insufficient resource to satisfy both process's demands. For the second reason, we need more information : we shall assume that the maximum demands of ***P*** and ***Q*** for the *R* resource are 3 and 2 units respectively. Now the states [***P****n*, ***Q***3] are *irrelevant*, because they will never be needed; further, we can forget about any transitions involving these states, because they will never be attempted. Here's the diagram with unattainable and irrelevant states marked, and forgettable transitions forgotten :



Our deadlock problems occur because the processes don't know that some states are unattainable, and they try to make transitions into these states. Examples of such impossible transitions are from [*P*2 , *Q*1] and [*P*1 , *Q*2] to [*P*2 , *Q*2]. In either case, the process requesting the transition would have to be suspended. Because these transitions could be requested, both [*P*2 , *Q*1] and [*P*1 , *Q*2] are potentially in some sense perilous; but in fact only one of them is necessarily dangerous. Consider what happens next in the two cases.

From [*P*1 , *Q*2] , *P* has requested another unit of *R*, and is therefore suspended. The next move must therefore come from *Q*; but as *Q* already has its maximum requirement for *R*, the next transition must be to [*P*1 , *Q*1], which is necessarily possible. This position is safe, at least so far as resource R is concerned – though that demonstration alone doesn't prove it. The condition for safety is that there should be a way to finish off *all* the processes, one by one. Here, we can give *Q* its full requirement of resource (as it happens, Q already has its maximum requirement), so if we just stop *P* altogether for a while we can finish *Q* (assuming that there is no complication from any other resource). Then all *Q* s resource becomes available for *P*, which will then be able to finish. In fact, the diagram shows us that we are bound to do something rather like that, for the only attainable system state in which *P* can have 3 units of *R* is [*P*3 , *Q*0]. In a more complicated situation with several processes, being able to finish one process is not a guarantee that the others are not deadlocked – so there has to be an identifiable sequence in which all the processes are demonstrably finishable.

From [*P*2 , *Q*1] , *P* has requested another unit of *R*, and is therefore suspended. The next move must therefore come from *Q*; but now *Q* has less than its maximum requirement for *R*, so we don't know whether the next request will be to release *R*, leading to [*P*2 , *Q*0], and at least possible safety, or to allocate more *R*, leading to the unattainable state [*P*2 , *Q*2], suspension of *Q*, and deadlock. [*P*2 , *Q*1] is therefore an unsafe state. Notice that we cannot assert that [*P*2 , *Q*0] is absolutely safe : we have no information on what's happening with the *S* resource. If the "safe" strategy from [*P*2 , *Q*0] is to finish off *P* first, and *P*s next request is for an *S* resource, and *Q* is holding all the *S* resource, then the overall state is unsafe. The system as a whole is only in a safe state if the safety condition is satisfied for every resource and all possible sequences of resource requests by all processes.

The point of the argument is that, although we don't know what the actual sequence of requests for *R* will be, it is *possible* that from [*P*2 , *Q*1] *both* processes' next requests will be for more *R*. More generally, a state is certainly unsafe if it is both attainable and relevant, but there is an adjacent unattainable and relevant state for every process concerned.

That's about as far as we can show the argument in terms of diagrams. Unfortunately, the true situation is rather worse than that. Consider a similar system in which both *P* and *Q* may require up to three units of *R*. Now [*P*1 , *Q*1] is unsafe – because the next two requests of both *P* and *Q* might all be for more *R*. The general rule is that

a state is unsafe if there is any possible sequence of requests for some resource which will drive it into an unattainable state, and these requests cannot be reordered to avoid the unattainable state by suspending processes.

There is no easy way to tell whether a state is safe or not – and the problem is not made any easier if we have to take account of several resources, when we have to be able to satisfy *the same* process's requirements simultaneously for each. It's no good saying that resource *R* is safe because *Q* can finish, followed by *P*, and that resource *S* is safe because *P* can finish, followed by *Q*. The only known technique is to search for a

possible completion sequence, and that is an expensive operation. Nothing can be done beforehand, as the computation depends on knowing all the current processes' maximum demands for resources, and therefore changes every time a new process is started.

THE BANKER'S ALGORITHM.

This algorithm, introduced by E.W. Dijkstra in 1965, uses the method just described as the basis of a technique for avoiding deadlock. The principle is simple : on receiving a process's request for a resource, suppose that the request is granted, then check the resulting state for safety, and refuse the request if the state would be unsafe. To perform the check :

```
repeat until no more processes can be finished :
     first search for a process which can be given all its
resources and can therefore finish;
     return all that process's resources to the system, and
suppose it has gone away.
```

If all the processes have gone, that was a safe state; otherwise, it was unsafe.

The algorithm is reliable, but it's expensive. You have to run it every time you allocate any of the resources it controls; and it has to cover all the processes in the system. It's less pessimistic than the deadlock prevention methods, but, being based on a process's maximum demand, is still unrealistically cautious. It gains over the more conservative methods in that, even though processes' maximum demands are taken into account, they need not all be satisfied simultaneously.

It's worth asking, too, for what resources it can sensibly be used. It was designed for systems in which a process's requirements could be summed up in a list such as "two card readers, four tape drives, and two printers"; it works best (or least badly) when there are a few processes and a few discrete resources, typically hardware units of some sort. There are very few, if any, computer systems nowadays which conform to that pattern. The Banker's Algorithm remains instructive and interesting, but it might no longer be very useful.

– so far as computing alone is concerned. But many computers nowadays are not alone. We remarked at the beginning of the chapter that the phenomenon of deadlock was also found in traffic engineering – and computers are now used to control traffic light systems. Deadlock happens in materials transport operations in manufacturing processes – also controlled by computers. We must therefore still be aware of deadlock in designing computer systems which deal with such areas of application. Deadlock is a possibility in any system in which several entities compete for a finite resource (as cars compete for space on the road), and for these cases the Banker's Algorithm might still be useful.

COPING WITH DEADLOCK.

We have seen that there are ways to prevent or to avoid deadlock, but that these are expensive to run, and typically very cautious in allocating resources. If we want an absolute guarantee that an operating system will never deadlock, we will be forced to use something of the sort – but most operating systems don't, and most of the time they get away with it. This is at least partly because of the change in pattern of computing generally, which we noticed when discussing the Banker's Algorithm : internal and external memory is comparatively cheap; no one nowadays expects to monopolise a disc drive, except for the little hard and floppy disc drives which are so cheap that we can have one or more each; everybody has a screen and keyboard, so there's no contention for input and output devices – and so on.

Of course, a system which doesn't take any measures against deadlock is likely to be stricken with it from time to time, so should provide means to deal with it when it happens. This cannot be done painlessly – if it could, it wouldn't be deadlock – and at

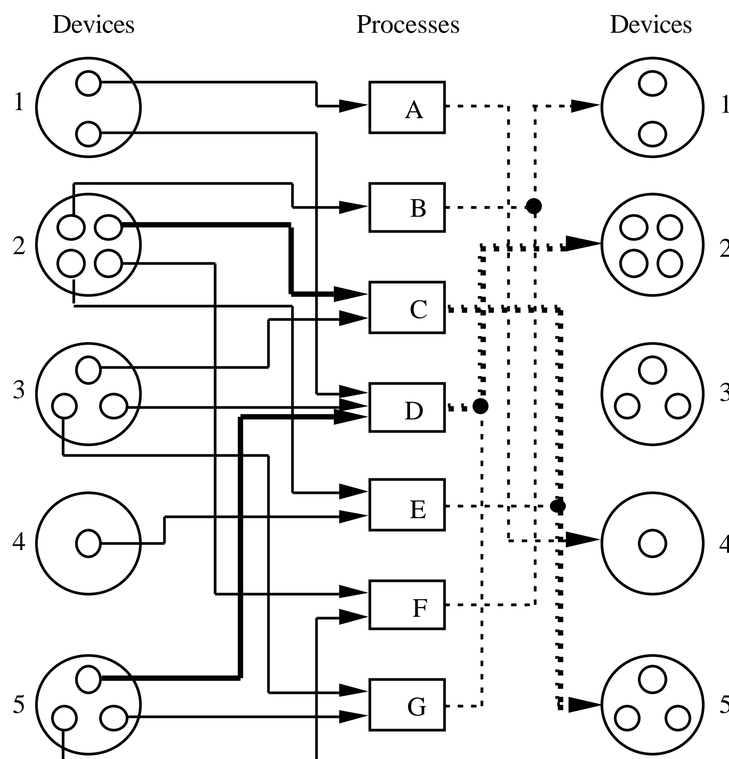
the very least some process requesting a resource must suffer. On the other hand, except for the victim, the pain need not always be great. Consider this (quite imaginary) baroque deadlock of seven processes and five resources, each considered to be a device type :

PROCESS TABLE		
Process	Owns	Awaiting
A	1	4
B	2	1
C	2, 3	5
D	1, 3, 5	2
E	2, 4	5
F	2, 5	1
G	3, 5	2

DEVICE TABLE		
Device	How many	Owned by
1	2	A, D
2	4	B, C, E, F
3	3	C, D, G
4	1	E
5	3	D, F, G

(The device table is provided for convenience; it contains no information not available from the process table and the assertion of deadlock, but it's a lot easier to see in that form.)

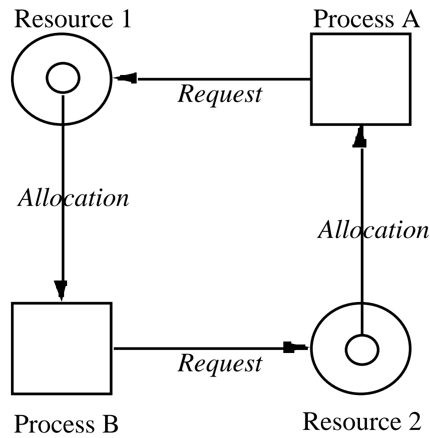
You can use this information to draw a *process-resource graph*, which shows which processes require which resources, and how the resources are distributed between the processes. The system is deadlocked if there is a closed cycle in the graph – so, in this example, D requires 2, which is owned by C, which requires 5, which is owned by D (and there are many other cycles as well). That's easy to do on this little example, but with a large processor, many processes, and many resources, it can take quite a long time. Even then, "easy" is relative. Suppose that one of the type 5 devices were owned by another process, say H. The difference to the table is tiny, but provided that H will eventually finish the deadlock is broken. How easy is it to tell the difference ? Some large systems do look for deadlocks every so often, but not too frequently. Others don't; they trust to luck, or hope that operators will notice deadlocks, or that people using the system interactively will get fed up of waiting and stop their programmes.



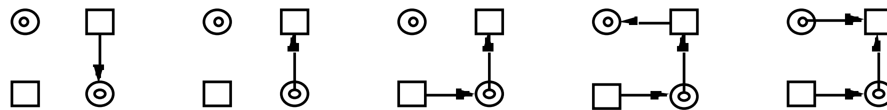
An interesting feature of the diagram is that no process is waiting for device 3, so in fact it contributes nothing to the deadlock. We leave it there for old time's sake; we had been using this example of deadlock for several years but never noticed that device 3 was unnecessary until we drew the diagram. In fact and hindsight, it's obvious enough (just check the "Awaiting" column in the process table), but it does make the point that it's hard to see what's going on without some sort of systematic search.

DETECTING DEADLOCK.

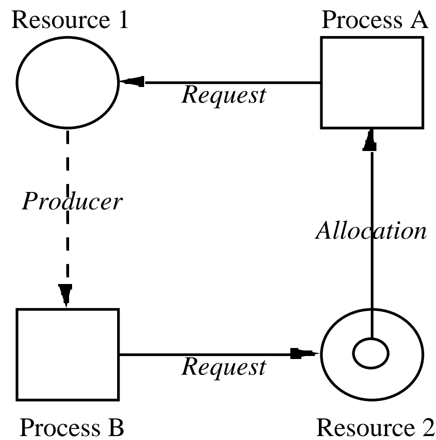
The process-resource graph – or a more abstract representation thereof – is the basis of deadlock detection methods. Here's a simple example of a deadlocked state :



Making an allocation reverses the direction of the arrow, so if there had been a cycle the operation destroys it. Here's a sequence of requests and allocations which doesn't lead to a deadlock to illustrate :



It works for abstract (consumable) resources too :



Once you find a deadlock, then what ? An interesting property of this deadlock, and of most others, is its *fragility*. If by some means any one of the deadlocked processes can be completed – or, equivalently, stopped – so that its resources become available, then the whole deadlock is broken. That's obviously true for a deadlock described by a single closed loop in the graph, but it's also often true of much more complicated patterns of dependence. There is therefore a good chance that by killing off one process, all will be made well. Which process shall we choose ? None of them is at fault; all are trying to complete perfectly legitimate requests for resources. There is rarely sufficient information available to make a sensible choice, unless all the processes are your own. One simple answer has the merit of equal treatment : restart all the processes, and hope that the deadlock – which might only happen with a small proportion of the possible sequences of resource requests – doesn't happen this time. Unfortunately, not all processes can sensibly be restarted from scratch; any process which significantly changes the state of the system in a way which survives the restart might behave differently next time round, and there is no way to identify the unrestartable processes. An equally democratic approach is to kill off a process chosen at random, and to keep doing so until the deadlock is clear.

AN UNRESTARTABLE PROCESS : In the days before disc space was cheap and plentiful, the university used a Cobol system called Stubol[MAN6] to provide service to a Cobol course offered by the Accountancy Department. Stubol included its own simplified file system, held on tape and brought onto the disc student by student as required. To make this work, the files were stored on the tape in order of student number, and the jobs – submitted as batches of punched cards – were sorted (on the disc) before the Cobol execution began. In any Stubol run, two tape files were open, one being read to retrieve the stored files, and one being written with the files to be carried on. To make sure that the tapes were properly identified, each had a header containing a serial number, and Stubol's first job was to read the two tapes, identify that with the higher serial number as the input tape, and write a new header with the next serial number onto the other ready for use as an output tape. The run could then start. But if something went wrong (with the operating system – Stubol itself was quite reliable) part way through, an attempt to restart the run would read the new serial numbers, choose what was the new partly written tape as the

input, and proceed to overwrite the other tape, thus destroying the real input. The result was that people with low student numbers had their jobs executed twice, which probably made nonsense of their files, while those with higher numbers lost any files they used to have. We finally fixed it by threatening the operators with public torture and slow execution if they forgot their instructions and restarted Stubol just once more .....

PROVING NO DEADLOCK.

In special cases, it might be possible to *prove* that deadlock cannot occur in a system. To do so, as should be clear from the preceding discussion, we must know a great deal about what's going on in the system – and, in particular, we need information about the precise sequence of resource requests made by each process. In a general-purpose system, this information is simply not available; but in some special systems where only a few known processes are ever executed, it might be possible. Real-time control systems are prime examples. The sequence of operations is also affected by the nature of the interactions between processes, and particularly any synchronisation requirements. Here, highly structured synchronising primitives, such as the Ada rendezvous, give more information than simple mechanisms like semaphores.

ERADICATING DEADLOCK.

If you can't *guarantee* no deadlock, at least you can try to find the sources of deadlock individually, and deal with them one by one : so any way to make sure they're identified and reported when the system is developed is at least potentially useful. One interesting approach[MAN4], particularly effective for the abstract resources like locks, is based on the deadlock prevention idea of imposing a standard order on resources. Each resource to be investigated is given a priority number which determines the desired order. Any departure from the strict stack-like order of acquisition and release of resources is reported, and investigated as an error. The priorities are not explicitly used during execution; the aim is to develop a programme in which the conflicts will not occur.

COMPARE :

Lane and Mooney[INT3] : Sections 7.4, 21.4 (on virtual devices); Silberschatz and Galvin[INT4] : Chapter 7.

REFERENCES.

MAN4 : J.G. Hunt : "Detection of deadlocks in multiprocess systems", *Sigplan Notices* **21#1**, 46 (January, 1986).

MAN6 : G.A. Creak : *The Stubol system*, Auckland University Computer Centre Technical Report #2 (1978).

QUESTIONS.

How would you classify preemption as a technique for preventing deadlock ?

Experiment with some resource demand patterns and the Banker's Algorithm. Find a case where you can complete some processes but not all. Experiment with multiple resources.

Work through the Banker's Algorithm for two different resources, A and B.

Look at some unsafe states, and try to guess the probability of deadlock actually happening. (You have no chance of getting a sensible answer without some idea of the behaviour of the running processes, but you can explore possibilities.)

Is it ever possible to give "an absolute guarantee that an operating system will never deadlock" ? Consider all sorts of resource that can lead to deadlock.

Deadlock is infectious. Suggest how the deadlocked example described by the tables given could have developed.

A true story : Once upon a time (July, 1991) someone brought one of us a file on a Macintosh disc. We looked at it on a Macintosh (running system 6) using Microsoft Word, discussed it for a while, and finally printed it. Then the visitor went away, taking his disc with him – but leaving the file still open in Word. Trouble started on trying to close it. "Insert disc Whatever", said the Macintosh. The disc had gone. Never mind – Apple have thought of that, and you can cancel the request using the and . keys, they say. Can you ? No, you can't. "Serious problem with file Thingummy", said Word, in an alert box. Click the "OK" button. "Insert disc Whatever", said the Macintosh. It was eventually necessary to restart the system. Questions : Is that deadlock ? How did it happen ? How could it have been avoided ? What should have happened ?