DEADLOCKS

Deadlocks can occur in a variety of different situations besides requesting dedicated I/O devices. Mostly, deadlocks involve resources when processes have been granted exclusive access to devices, data records, files, and so forth. Such objects are referred to resources. A computer system will normally have different resources that can be acquired. For some resources, several identical instances may be available, such as drives. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that must be acquired, used, and released over the course of time.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

i). **Request**. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

ii). **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

iii). **Release**. The process releases the resource.

Resources come in two types: **preemptable** and **nonpreemptable**. A preemptable resource is one that can be taken away from the process owning it with no ill effects. A nonpreeptable resource, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail. Deadlocks involve nopreeptable resources.

If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again. A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again. Although this process is not blocked, for all intents and purposes it is as good as blocked, because it cannot do any useful work. We will assume that when a process is denied a resource request, it is put to sleep. The exact nature of requesting a resource is highly system dependent. In some systems, a request system call is provided to allow processes to explicitly ask for resources. In others, the only resources that the operating system knows about are special files that only one process can have open at a time. These are opened by the usual open call. If the file is already in use, the caller is blocked until its current owner closes it.

DEFINING DEADLOCK

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. It can therefore be defined *as the permanent blocking of a set of processes that either compete for system resources or communicate with each other.*
Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by, say, an alarm, and then causing events that release other processes in the set.

In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant. This result holds for any kind of resource, including both hardware and software. This kind of deadlock is called a **resource deadlock**. It is probably the most common kind, but it is not the only kind.

DEADLOCK CHARACTERIZATION

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. a deadlock situation can arise if the following four conditions hold simultaneously in a system.

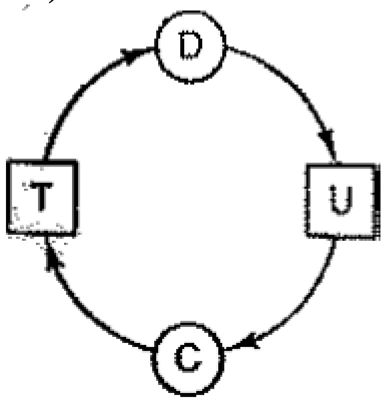
i). **Mutual exclusion** - At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
ii). **Hold and wait** - A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
iii). **No preemption** - Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
iv). **Circular wait** - A set $\{P0, P1, ..., Pn\}$ of waiting processes must exist such that $P0$ is waiting for a resource held by $P1$, $P1$ is waiting for a resource held by $P2$, ..., $Pn-1$ is waiting for a resource held by Pn, and Pn is waiting for a resource held by $P0$.

All four of these conditions must be present for a resource deadlock to occur. If one of them is absent, no resource deadlock is possible. It is worth noting that each condition relates to a policy that a system can have or not have.

DEADLOCK MODELING

A useful tool in characterizing the allocation of resources to processes is the **resource allocation graph.** This is a directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node. The graphs have two kinds of nodes: **processes,** shown as circles, and **resources**, shown as squares. A directed arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by,

granted to, and is currently held by that process. A directed arc from a process to a resource means that the process is currently blocked waiting for that resource.
For example, consider two processes, D and C, and two resources T and U. Process C is waiting for resource *T*, which is currently held by process *D.* Process *D* is not about to release resource *T* because it is waiting for resource *U*, held by C. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). This can be modelled as shown below.



For instance, there is a deadlock because there is a cycle in the waiting order. If there is no cycle, then there is no deadlock.

After deadlock is detected and seen to exist among a set of processes, what next?

DEALING WITH DEADLOCKS
There are four strategies that can be used for dealing with deadlocks.

1. **Deadlock Ignorance** - Just ignore the problem. Maybe if you ignore it, it will ignore you.
2. **Deadlock Detection and recovery** - Let deadlocks occur, detect them, and take action.
3. **Deadlock Avoidance** - Dynamic avoidance by careful resource allocation.
4. **Deadlock Prevention** - through structurally negating one of the four required conditions.

1. Deadlock Ignorance
This is the simplest approach, also referred to as **Ostrich Algorithm** - stick your head in the sand and pretend there is no problem at all. Different people react to this strategy in different ways. Mathematicians find it totally unacceptable and say that deadlocks must be prevented at all costs. Engineers ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is. If deadlocks occur on the average once every five years, but system crashes due to hardware failures, compiler errors, and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.
To make this contrast more specific, consider an operating system that blocks the caller when an open system calls on a physical device such as a CD-ROM driver or a printer cannot be carried out because the device is busy. Typically, it is up to the device driver to decide what action to take under such circumstances. Blocking or returning an error code are two obvious possibilities. If one process successfully opens the CD-ROM drive and another successfully opens the printer and then

each process tries to open the other one and blocks trying, we have a deadlock. Few current systems will detect this but will just ignore.

2. Deadlock Detection and Recovery

When this technique is used, the system does not attempt to prevent deadlocks from occurring. Instead, it lets them occur, tries to detect when this happens, and then takes some action to recover after the fact.

✓ **Deadlock Detection**

This activity begins with constructing resource allocation graph. If this graph contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked.

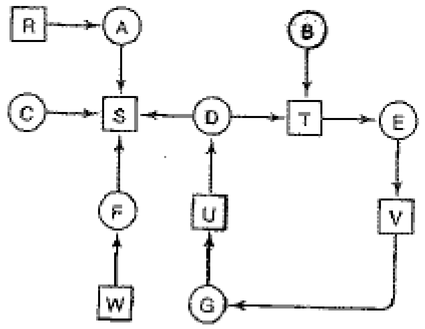
Consider the following example:

Consider a system with seven processes, *A* though G, and six resources, *R* through *W.* The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process *A* holds *R* and wants *S.*
2. Process *B* holds nothing but wants *T.*
3. Process C holds nothing but wants *S.*
4. Process *D* holds *U* and wants *S* and *T.*
5. Process *E* holds Tand wants *V.*
6. Process *F* holds *W* and wants *S.*
7. Process *G* holds *V* and wants *U.*

The question is: "Is this system deadlocked, and if so, which processes are involved?"

To answer this question, we can construct the resource allocation graph as follows:



This graph contains one cycle, which can be seen by visual inspection. From this cycle, we can see that processes D, *E,* and *G* are all deadlocked. Processes *A, C,* and *F* are not deadlock because *S* can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete. Although it is relatively simple to pick out the deadlocked processes by eye from a simple graph, for use in actual systems, the need a formal algorithm for detecting deadlocks. Many algorithms for detecting cycles in directed graphs are known.

✓ **Deadlock Recovery**

Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. What next? Some way is needed to recover and get the system going again. Various ways of recovering from deadlock exist and they include:

i). **Recovery through Preemption**

In some cases, it may be possible to temporarily take a resource away from its current owner and give it to another process. In many cases, manual intervention may be required, especially in batch processing operating systems running on mainframes.

For example, to take a laser printer away from its owner, the operator can collect all the sheets already printed and put them in a pile. Then the process can be suspended (marked as not runnable). At this point the printer can be assigned to another process. When that process finishes, the pile of printed sheets can be put back in the printer's output tray and the original process restarted.

The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

ii). **Recovery through Rollback**

If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes check-pointed periodically. Check-pointing a process means that its state is written to a file so that it can be restarted later. The checkpoint contains not only the memory image, but also the resource state, in other words, which resources are currently assigned to the process. To be most effective, new checkpoints should not overwrite old ones but should be written to new files, so as the process executes, a whole sequence accumulates.

When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired that resource by starting one of its earlier checkpoints. All the work done since the checkpoint is lost (e.g., output printed since the checkpoint must be discarded, since it will be printed again). In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

iii). **Recovery through Killing Processes**

The crudest, but simplest way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue. If this does not help, it can be repeated until the cycle is broken.

Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources. In this approach, the process to be killed is carefully chosen because it is holding resources that some process in the cycle needs. For example, one process might hold a printer and want a plotter, with another process holding a plotter and wanting a printer. These two are deadlocked. A third process may hold another identical printer and another identical plotter and be happily running. Killing the third process will release these resources and break the deadlock involving the first two.

Where possible, it is best to kill a process that can be rerun from the beginning with no ill effects. For example, a compilation can always be rerun because all it does is read a source file and produce an object file. If it is killed partway through, the first run has no influence on the second run. On the other hand, a process that updates a database cannot always be run a second time safely. If the process adds 1 to some field of a table in the database, running it once, killing it, and then running it again will add 2 to the field, which is incorrect.

3. **Deadlock Avoidance**

In most systems, however, resources are requested one at a time. The system must be able to decide whether granting a resource is safe or not and only make the allocation when it is safe. Thus the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time? The answer is a qualified yes—we can avoid deadlocks, but only if certain information is available in advance and by careful resource allocation. Commonly used algorithm is the banker's algorithm. (Students to research on it).

4. Deadlock Prevention

Deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known. Deadlock prevention mechanism ensure that at least one of the conditions is never satisfied.

i). Attacking the Mutual Exclusion Condition

If no resource were ever assigned exclusively to a single process, we would never have deadlocks. However, it is equally clear that allowing two processes to write on the printer at the same time will lead to chaos. By spooling printer output, several processes can generate output at the same time. In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.

If the daemon is programmed to begin printing even before all the output is spooled, the printer might lie idle if an output process decides to wait several hours after the first burst of output. For this reason, daemons are normally programmed to print only after the complete output file is available. However, this decision itself could lead to deadlock. What would happen if two processes each filled up one half of the available spooling space with output and neither was finished producing *its full output? In this case* we *have two processes that* have each finished part, but not all, of their output, and cannot continue. Neither process will ever finish, so we have a deadlock on the disk.

Nevertheless, there is a germ of an idea here that is frequently applicable. Avoid assigning a resource when that is not absolutely necessary, and try to make sure that as few processes as possible may actually claim the resource.

ii). Attacking the Hold and Wait Condition

If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require all processes to request all their resources before starting execution. If everything is available, the process will be allocated whatever it needs and can run to completion.

If one or more resources are busy, nothing will be allocated and the process would just wait. An immediate problem with this approach is that many processes do not know how many resources they will need until they have started running. In fact, if they knew, the banker's algorithm could be used. Another problem is that resources will not be used optimally with this approach. Take, as an example, a process that reads data from an input tape, analyzes it for an hour, and then writes, an output tape as well as plotting the results. If all resources must be requested in advance, the process will tie up the output tape drive and the plotter for an hour.

Nevertheless, some mainframe batch systems require the user to list all the resources on the first line of each job. The system then acquires all resources immediately and keeps them until the job finishes. While this method puts a burden on the programmer and wastes resources, it does prevent deadlocks.

A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once.

iii). Attacking the No Preemption Condition

Attacking the third condition (no preemption) is also a possibility. If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available is tricky at best and impossible at worst. However, some resources can be virtualized to avoid this situation. Spooling printer output to the disk and allowing only the printer daemon access to the real printer eliminates deadlocks involving the printer, although it creates one for disk space. With large disks, however, running out of disk space is unlikely. However, not all resources can be virtualized like this.

iv). Attacking the Circular Wait Condition

The circular wait can be eliminated in several ways.

One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.

Another way to avoid the circular wait is to provide a global numbering of all the resources. Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer. With this rule, the resource allocation graph can never have cycles.

With more than two processes, the same logic holds. At every instant, one of the assigned resources will be highest. The process holding that resource will never ask for a resource already assigned. It will either finish, or at worst, request even higher numbered resources, all of which are available. Eventually, it will finish and free its resources. At this point, some other process will hold the highest resource and can also finish. In short, there exists a scenario in which all processes finish, so no deadlock is present.