LECTURE 3 - Deadlock Management:

Computer systems have resources that can be used by one process at a time e.g. having two processes using the same slot in the process table will lead to a system crash. Consequently, the operating system has the ability to grant a process exclusive access to certain resources.

In certain circumstances, a process needs exclusive access to more than one resource. In a database system, for example, a process may have to lock several records to avoid *race conditions*. If process A locks record R1 and process B locks record R2 and then each process tries to lock the other one's record, we have a **deadlock**. Deadlocks can also occur in hardware.

2.5.1 Resources:

Deadlocks can occur when processes have been granted exclusive access to devices and files. Resources come in two types: pre-emptable and non-preemptable. A **preemptabl**e resource is one that can taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource.

A **non-preemptable** resource is one that cannot be taken away from its current owner without causing the computation to fail. If a process has begun printing, taking the printer away from it and giving it to another process will result in garbled output. Printers are not preemptable. In general deadlocks involve non-preemptable resources. Potential deadlocks that involve preemptable ones can usually be solved by reallocating resources from one process to another.
The sequence of events required to use a resource is:

1. Request the resource
2. Use the resource
3. Release the resource

If the resource is not available when it is requested, the requesting process is forced to wait. In some systems, the process block when a resource request fails and is awakened when it becomes available.

2.5..2 Principles of Deadlocks

A deadlock can be formally defined as: *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.*

Because all the processes are waiting, none of them will ever cause any of the event that could wake up any of the other members of the set, and all the processes continue to wait forever.

In most cases, the event that each process is waiting for is the release of some resource that is currently possessed by another member of the set. None of the processes can run, none of them can release any resource and none of them can be awakened.
There are four conditions that must hold for a deadlock to occur:

- Mutual exclusion condition – Each resource is either currently assigned to exactly one process or is available
- Hold and wait condition – Processes currently holding resources granted earlier can request new resources
- Non preemption condition – Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them
- Circular wait condition – There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All of these conditions must hold for a deadlock to occur. If one or more of these conditions is absent, no deadlock can occur.

In general four strategies are used for dealing with deadlocks:

1. Just ignore the problem
2. Detection and recovery
3. Dynamic avoidance by careful resource allocation
4. Prevention, by structurally negating one of the four required conditions

2.5.3 The ostrich algorithm

The simplest approach is the Ostrich algorithm: stick your head in the sand and pretend there is no problem. Different people react to this algorithm in different ways. Some people say that it is totally unacceptable and that deadlocks must be prevented at all costs.
In the UNIX operating system there are potentially many deadlocks that are not even detected, let alone, automatically broken. The total number of processes in the system is determined by the number of entries in the process table. Thus process table slots are finite resources. If a FORK fails because the table is full, reasonable approach is to wait a random time and try again.
The UNIX approach is to ignore such problems on the assumption that processes and users will prefer an occasional deadlock rather than having restrictions.

2.5.4 Detection and Recovery

When this technique is used, the system does not do anything except monitor the requests and release of resources. Every time a resource is requested or released, the resource graph is updated and a check is made to see if any cycles exist. If a cycle exists, one of the processes in the cycle is killed. If this does not break the cycle, another process is killed until the cycle is broken.
Another method is to periodically check to see if there are any processes that have been continuously blocked for more than say, 1 hour. Such processes are then killed. This strategy is common in large main frame computers.

2.5.5 Deadlock Prevention

This imposes suitable restrictions on processes so that deadlocks are structurally impossible. The four conditions stated provide some possible solutions. If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be impossible.
Let us look at the mutual exclusion condition. If no resources were ever assigned exclusively to a single process, no deadlocks would occur. However, doing this would lead to chaos, for instance, in the case of a printer. By spooling printer output, several processes can generate output at the same time. In this model, the only process that requests the physical hardware is allowed to have exactly one program to be used for service provision. Since the daemon never requests any other resources, we can eliminate deadlocks for the printer. Unfortunately, not all devices can be spooled.
The second condition looks more promising. If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way is to require that all processes request all their resources before starting execution. If everything were available, the process would be allocated whatever they needed and could run to completion. If one ore more resources were busy, nothing would be allocated and the process would just wait. The problem with this approach is that many processes would not know what the required until they have started running.
A slightly different approach to break the hold-and-wait condition is to require a process requesting for a resource to first temporarily release all the resources it currently hold. Only if the request is successful can it get the original resources back.

It is impossible to tackle the third condition. If a process has been assigned the printer and it is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available will lead to a mess.

The circular wait condition can be eliminated in several ways.

1. Say that a process is entitled to a single resource at a time. If it needs a second one, it must release the first one. For a process that needs to copy a large file from tape to disk, this is not acceptable.
2. Provide a global numbering of all resources in the system, for example, as shown below. The rule is that: processes can request for resources whenever they want to, but all the requests must be made in a certain numerical (canonical) order. A process may first request a printer and then a tape drive, but it may not request a plotter then a printer.

1. CD-ROM
2. Printer
3. Plotter
4. Tape Drive
5. Robot Arm

2.5.6 Process Termination

Some reasons for process termination would include:

1. Normal completion
2. Time limit exceeded
3. Memory unavailable
4. Bounds violation
5. Protection error
6. Arithmetic error
7. I/O failure
8. Invalid instruction
9. Privileged instruction
10. Data misuse
11. Operator or Operating System Intervention
12. Parent termination.