Two approaches to deadlock avoidance are:

* Do not start a process if its demands might lead to deadlock.
* Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

Both of these approaches require knowing the maximum requirements of each resource in advance. The first approach will only allow a process to start if all of the current processes maximum resource requirements plus the new process maximum resources can be met. This is very stingy and pessimistic.

The second approach doesn’t allow an incremental resource request to process if it might lead to deadlock. The Banker’s Algorithm is the common example used for this approach. It postpones requests that could lead to deadlock. This is Resource Allocation Denial.

Because both of these approaches are based on the maximum resource requirements for each process, there are delays when processes could be completed because they do not use the maximum requirements. If they aren’t using the maximum requirements, deadlock may not occur. Instead, because of the maximum requirements, processes can be blocked for long periods of time.

Also, future resource requirements may not be known by the Operating System.I found that deadlock detection or avoidance is expensive. From a lecture on deadlocks at Illinois:

* Unix and Windows use the Ostrich Approach. They do nothing.
* Typical Apps use deadlock Prevention (order locks)
* Transaction systems need to use deadlock detection/recovery/avoidance/prevention [1]

Looking into distributed systems I also found that it is best to avoid deadlock avoidance.

From Rutgers: "The same conditions for deadlock in uniprocessors apply to distributed systems. Unfortunately, as in many other aspects of distributed systems, they are harder to detect, avoid, and prevent. Four strategies can be used to handle deadlock:

1. **ignorance**: ignore the problem; assume that a deadlock will never occur. This is asurprisingly common approach.
2. **detection**: let a deadlock occur, detect it, and then deal with it by aborting and later restarting a process that causes deadlock.
3. **prevention**: make a deadlock impossible by granting requests so that one of the necessary conditions for deadlock does not hold.
4. **avoidance**: choose resource allocation carefully so that deadlock will not occur. Resource requests can be honored as long as the system remains in a safe (non-deadlock) state after resources are allocated.

The last of these, deadlock avoidance through resource allocation is difficult and requires the ability to predict precisely the resources that will be needed and the times that they will beneeded. This is difficult and not practical in real systems. The first of these is triviallysimple but, of course, ineffective for actually doing anything about deadlock conditions.

We will focus on the middle two approaches.

In a conventional system, the operating system is the component that is responsible for resource allocation and is the ideal entity to detect deadlock. Deadlock can be resolved by killing a process. This, of course, is not a good thing for the process. However, if processes are transactional in nature, then aborting the transaction is an anticipated operation. Transactions are designed to withstand being aborted and, as such, it is perfectly reasonable to abort one or more transactions to break a deadlock. The transaction can be restarted later at a time when, we hope, it will not create another deadlock." [2]

Interestingly, While researching deadlock avoidance, I found the article “Avoiding Deadlock Avoidance” by Hari K. Pyla and Srinidhi Varadarajan. In their article, they provide some performance result on:

"a language independent runtime system called Sammati that provides automatic deadlock detection and recovery for threaded applications that use the POSIX threads (pthreads) interface - the de facto standard for UNIX systems. The runtime is implemented as a pre-loadable library and does not require either the application source code or recompiling/relinking phases, enabling its use for existing applications with arbitrary multi-threading models." [3]

In conclusion I think that there may be better alternatives to deadlock avoidance. In CIS 5200 we are learning about Concurrency, Threads and Synchronization in Java.I also think that Programmers need to be aware of deadlock issues when programming. According to the book, "Java: How to Program":

When a higher-priority thread enters the ready state, the operating system generally preempts the currently running thread (an operation known as***preemptive scheduling)***. Depending on the operating system, a steady influx of higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads. Such***indefinite postponement***is sometimes referred to more colorfully as***starvation***. Operating systems employ a technique called aging to prevent starvation—as a thread waits in the ready state, the operating system gradually increases the thread’s priority to ensure that the thread will eventually run.

Another problem related to indefinite postponement is called***deadlock*** This occurs when a waiting thread (let’s call this thread1) cannot proceed because it’s waiting (either directly or indirectly) for another thread (let’s call this thread2) to proceed, while simultaneously thread2 cannot proceed because it’s waiting (either directly or indirectly) for thread1 to proceed. The two threads are waiting for each other, so the actions that would enable each thread to continue execution can never occur." [4]

I am just learning this in Java, there is more information on the Oracle website.

[1] Illinois University, (2012). Deadlock,

Web. Feb 2016.  [https://courses.engr.illinois.edu/cs241/fa2012/lectures/25-Deadlock.pdf (Links to an external site.)](https://courses.engr.illinois.edu/cs241/fa2012/lectures/25-Deadlock.pdf)

[2]  Krzyzanowski P, Rutgers University, (2012). Distributed Deadlock.

Web. Feb 2016. [https://www.cs.rutgers.edu/~pxk/417/notes/deadlock.html (Links to an external site.)](https://www.cs.rutgers.edu/~pxk/417/notes/deadlock.html)

[3] Pyla, H., & Varadarajan, S. (2010). Avoiding deadlock avoidance. Paper presented at the PACT '10 Proceedings of the 19th international conference on Parallel architectures and compilation techniques. pp. 75-86. doi:10.1145/1854273.1854288

[4] Deitel & Deitel, (2015). "Java How to Program", Prentice-Hall.

# Original: DQ2

Five major activities of an operating system with respect to process management are:

**Creation and deletion of both user and system processes.** The processes in the system can execute concurrently for information sharing, computation speedup, modularity, and convenience. Concurrent execution requires a mechanism for process creation and deletion. The required resources are given to the process when it is created, or allocated to it while it is running. When the process terminates, the OS needs to reclaim any reusable resources.

**Suspension and resumption of processes.** In process scheduling, the OS needs to change the process's state to waiting or ready state when it is waiting for some resources. When the required resources are available, OS needs to change its state to running state to resume its execution.

**Provision of mechanism for process synchronization.** Cooperating processes may share data. Concurrent access to shared data may result in data inconsistency. OS has to provide mechanisms for processes synchronization to ensure the orderly execution of cooperating processes, so that data consistency is maintained.

**Provision of mechanism for process communication.** The processes executing under the OS may be either independent processes or cooperating processes. Cooperating processes must have the means to communicate with each other.

**Provision of mechanisms for deadlock handling.** In a multiprogramming environment, several processes may compete for a finite number of resources. If a deadlock occurs, all waiting processes will never change their waiting state to running state again, resources are wasted and jobs will never be completed.

**Multithreading**refers to the ability of an operating system to support multiple threads of execution within a single process. The traditional approach of a single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach. In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection. The following are associated with processes:

* A virtual address space that holds the process image
* Protected access to processors, other processes (for interprocess communication), files, and I/O resources (devices and channels)

Within a process, there may be one or more threads, each with the following:

* A thread execution state (Running, Ready, etc.).
* A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process.
* An execution stack.
* Some per-thread static storage for local variables.
* Access to the memory and resources of its process, shared with all other threads in that process.

While the process is running, processor registers are controlled by that process, and the contents of these registers are saved when the process is not running. In a multithreaded environment, there is still a single process control block and user address space associated with the process, but now there are separate stacks for each thread, as well as a separate control block for each thread containing register values, priority, and other thread-related state information.

**Critical section**: A section of code within a process that requires access to shared resources and that may not be executed while another process is in a corresponding section of code.

**Deadlock**: A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

**Livelock**: A situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.

**Mutual Exclusio**n: The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

**Race Condition:**A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

**Starvation**: A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Deadlock can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set. Deadlock is permanent because none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case. All deadlocks involve conflicting needs for resources by two or more processes.

The strategy of deadlock prevention is, simply put, to design a system in such a way that the possibility of deadlock is excluded.

Deadlock detection strategies do not limit resource access or restrict process actions. With deadlock detection, requested resources are granted to processes whenever possible. Periodically, the operating system performs an algorithm that allows it to detect the circular wait condition. Once deadlock has been detected, some strategy is needed for recovery.

In **deadlock prevention**, we constrain resource requests to prevent at least one of the four conditions of deadlock (Mutual exclusion, Hold and wait, No preemption, Circular wait). This is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly, by preventing circular wait. This leads to inefficient use of resources and inefficient execution of processes. **Deadlock avoidance**, on the other hand, allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached. As such, avoidance allows more concurrency than prevention. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. Deadlock avoidance thus requires knowledge of future process resource requests. Two approaches to deadlock avoidance are:

* Do not start a process if its demands might lead to deadlock.
* Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

Discuss these approaches.

Follow up:

Shawn gives a good overview of the Deadlock Avoidance, the Banker's algorithm and their disadvantages.Shawn talks about the 3 conditions that must be satisfied for deadlock to occur. He also gives an example of the safe states and unsafe states in the Bankers Algorithm.

I think the disadvantages of deadlock avoidance is why most operating systems use the Ostrich Approach. The text book says that most Operating Systems, including Unix, ignore the deadlock problem and pretend that deadlocks(two or more processes are waiting indefinitely for an event  that can be caused by only one of the waiting processes) do not occur in the system. I think that they do this to let users work instead of making them wait until all the possible resources they might use are available. Mark Handley says in his slide presentation on Deadlock Avoidance that this is a trade off between convenience and correctness [1].

The cost of using a deadlock avoidance algorithm is greater than the cost of people being unable to work or waiting to work. Concerning unsafe states and the Bankers Algorithm, Rashid Bin Muhammad from Kent State University says: "It is important to note that an unsafe state does not imply the existence or even the eventual existence a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later. Haberman [1969] has shown that executing of the algorithm has complexity proportional to N2 where N is the number of processes and since the algorithm is executed each time a resource request occurs, the overhead is significant. [2]" Toshimi Minoura says in his research paper, that some deadlock prevention schemes and some deadlock avoidance algorithms that are well known, are too restrictive for many practical applications. Maximum claims make too many resources unavailable [3].

The text book says that if a system does not employ a protocol that ensures deadlocks will not occur, then a detection and recovery scheme must be employed. In my experience, this may be a manual process. For example a System Administrator gets calls that users can't work because the system is frozen. The System Administrator has to look at processes, kill a process or reboot the system. Programmers have to be aware of deadlock and write code to prevent deadlock for their programs to work correctly (prevent the program from hanging). This reduces the downtime that results from deadlocks.

[1] Handley, M., (2004) . 10. Deadlock Avoidance,

Web. Feb 2016. [http://nrg.cs.ucl.ac.uk/mjh/3005/2004/10-deadlock-avoidance.pdf (Links to an external site.)](http://nrg.cs.ucl.ac.uk/mjh/3005/2004/10-deadlock-avoidance.pdf)

[2] Muhammad, R.B., Deadlock Avoidance,

Web. Feb 2016.  [http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/deadlockAvoidance.htm (Links to an external site.) (Links to an external site.)](http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/deadlockAvoidance.htm)

[3] Toshimi Minoura. 1982. Deadlock Avoidance Revisited. J. ACM 29, 4 (October 1982), 1023-1048.