# Deadlock – college essay

Deadlock occurs when each process in a set is waiting for an event that can only be caused by some other process in that set. Basically, deadlock is an operating system issue that reared its head with the advent of multiprogramming environments. First and foremost, a deadlock cannot occur unless there are at least two processes trying to run concurrently. There are actually four distinct circumstances that must be met for deadlock to occur, which will be discussed shortly, but it is worth noting that it is fundamentally a multiprocessing problem.

Mono-processing systems do not have to worry about deadlock. The reason is that deadlock involves resource allocation, and if there is only one process, it has uncontested access to all resources. Only certain types of resources are associated with deadlock, and they are of the exclusive-use, non-preemptible type. That is to say, only one process can use the resource at any given time, and once allocated the resource cannot be unallocated by the operating system, but rather the process has control over the resource until it completes its task.

Excellent examples of such resources are printers, plotters, tape drives, etc.. Resources that do not fit the criteria are memory and the CPU. While it is convenient to discuss deadlock in terms of hardware resources, there are software resources that are equally good candidates for deadlock, such as records in a data base system, slots in a process table, or spooling space. Hardware or software, all that matters is that the resources are non-preemptible and serially reusable.

The four conditions that must exist for deadlock are as follows: The first two are described above – mutually exclusive use of resources by the processes and non-preemption (resources cannot be removed from the processes). The third condition is called the ‘ hold and wait’ or ‘ wait for’ condition. This means that processes can maintain possession of a resource and simultaneously request or wait for other resources. The last is the circular wait condition, which just states that there must be a chain of processes, each of which is waiting for a resource that is held by the next process in the chain.

Since four separate, independent conditions must be met, one might be inclined to simply feel that deadlock is just to unlikely to worry about. Consider, however, that the ‘ wait for’ condition is virtually always met, since almost no system forces a process to wait for every resource it will ever need before starting. And consider also, that every time the right type of resource is used, two more conditions are immediately met (mutual exclusion and non-preemption). Essentially then, the likelihood of deadlock is equal to the likelihood of a circular wait.

Is this an acceptable risk? There are four practical strategies for deadlock (well, mostly practical). The first strategy is to just ignore the possibility of deadlock. This might seem absurd, until it becomes apparent there can be a major performance or functionality decrease associated with a more rigorous solution. Thus, if it can be shown that the likelihood of a deadlock as well as its cost of recovery are not sufficient to outweigh the disadvantages of solving the problem, then it does, in fact, become an acceptable risk.

Consider, for example, if a deadlock will only occur on average once every twenty years, and it will not cause system damage or permanent data loss; while the system may crash for many other reasons on average once very other month. Is it worthwhile to hinder the performance or convenience of the system to solve the deadlock problem? Probably not. Another approach is to attempt to prevent deadlocks from occurring, which is to say that the system is pre-conditioned to remove any possibility of deadlock. The theory is that it is worthwhile to pay the price to ensure that deadlock will never occur.

And if it can’t occur, you don’t have to worry about it. But how can it be guaranteed that deadlock will not occur? Simple, just prevent any of the four conditions from occurring. Preventing the mutual exclusion condition is generally not a good idea. To be sure, many resources could be effectively spooled, which would eliminated mutual exclusion, but spooling space itself then becomes another potential deadlock resource, not to mention the fact that certain resources simply can’t be spooled. Prevention of the hold and wait condition might lead to greater success.

This can be accomplished by forcing processes to request and be granted every needed resource before starting. Unfortunately, many processes don’t know which resources will be needed until they need them. Also, quite obviously, having a process hold all necessary resources until it completes leads to very inefficient resource utilization. Alternatively, the OS could require that a process relinquish all resources in its possession before requesting another. If the resource is available, the process is granted it in addition to the ones it just gave up.

Preventing the Non-preemption condition is also possible, although probably not recommended. Consider taking the printer away from a process that is only partially done using it. The last condition that can be avoided is that of circular wait. This can be avoided by numerically ordering the system resources and forcing processes to request resources in numerical order. This can work fairly well, except for two drawbacks. First, with many resources and many processes, it may be impossible to find an order that will allow all processes to finish.

Also, if a new resource ever becomes available, the system may have restart to include it in the list. A different approach to deadlocks is to allow them to happen, and then to detect and recover from them. This strategy avoids all the drawbacks of the preventative measures described above, but introduces others. One way to detect deadlocks is to have the OS maintain a resource graph that keeps track of requests and releases. If the OS detects a cycle, it simply kills one of the processes in the cycle.

Or, even simpler, the OS can simply check every now and then to see if a process has waited an inordinate amount of time for a resource. If so, the process is killed. Despite the fact that this type of strategy is often used in large mainframes and batch systems, it has some major problems. Simply killing processes, while fun, is not usually desirable, especially if the process is yours. Also, to maintain system integrity, when a process is killed, all files modified by that process must be restored to their original state. In order to do this there is a tremendous amount of overhead and caching created.

The last strategy is not to physically prevent deadlocks, or allow them to happen, but to do one’s best to avoid them through carefully monitoring resource requests. Monitoring means employing some type of tested algorithm that can predict deadlock based on the current state of things. Presumably, the algorithm then avoids deadlock by only granting resource requests when it is prudent to do so. One well known algorithm for deadlock avoidance is the Banker’s algorithm. No algorithm for deadlock avoidance is perfect, however, so considerations about their usage still exist.