1 Class #8

As has been mentioned, alternatives to deadlock detection fall into two classes. The first approach attempts to prevent all the 4 conditions necessary for a deadlock from being true at the same time. The second approach works by getting information from processes before the request time and then deciding about actual allocation at the time of the resource allocation on case-by-case basis.

1.1 Deadlock Prevention

Deadlock prevention can be achieved by ensuring that at least one of the conditions necessary for deadlocks to occur must be false at all times. A case by case analysis for the methods available to do this for each of the four cases follows.

1.1.1 Mutual Exclusion

The mutual-exclusion condition **must** hold for resources that are inherently non-shareable. For example, printing on a printer is an exclusive activity. A more subtle instance is when a terminal is being used for output by multiple processes. Typically, output by each process is atomic to the extent of a newline. At the same time, shareable resources such as read-only files cannot be involved in a deadlock.

1.1.2 Hold and Wait

The *Hold-and-wait* condition can be prevented from being true in a bunch of ways. A process can be made to request all its resources before it ever starts execution. For example, a process using two output devices can be allowed to start only after both of them are free. At that time, those units are assigned to the process and are held by the process until the process terminates. Other protocols are also possible. The disdvantage of such protocols is that resource utilization can be quite low, since a process could be unnecessarily conservative about asking for resources.

1.1.3 Preemption

Adding preemption to the operating system is another way to prevent all the four conditions being satisfied simultaneously. For example, a simple protocal for doing preemption can be the following. Let a process make a request for certain resources. If any of the resources is unavailable for some reason, then all the resources already acquired by the process are preempted and taken away from the process. The preempted resources are added to the list of resources that the process is waiting for. An alternative strategy for preemption could be to take away resources that a process asks for from another holding process if the holding process is not running. In other words, resources can be taken away from blocked processes to enable other processes to run. Preemption is not a very useful method in general because of the enormous overhead associated with it.

1.1.4 Circular waiting

This condition can be prevented from holding using a variety of approaches as well. One particular way would be to assign a unique positive integer to each type of resource. A process would be required to request for resources according to increasing *resource number*. It is easy to see (using the *wait-for graph* that this approach ensures that there cannot be a circular wait condition holding at any time.

1.2 Deadlock Avoidance

The deadlock prevention algorithms do not make any assumptions regarding the particular needs of individual processes. The potential drawbacks of such approaches are low system utilization and throughput. However, there exist another class of algorithms, *deadlock avoidance* which require information from a process at startup time about the resources that they need. Armed with this information, an avoidance algorithm can make a case-by-case decision about whether a particular request for a resource can be granted.

1.2.1 Safe state

The avoidance algorithms follow a basic strategy similar to the following: There is a notion of a safe state, which is guarenteed not to lead to a deadlock. Conversely there is a notion of an unsafe state. It is guarenteed that every deadlock state **must** be an unsafe state. The algorithms ensure that no resource allocation is ever made at a safe state, so that a transition is made to an unsafe state.

A system is defined to be safe if there exists a way to allocate resources to each process and still avoid a deadlock. Formally, given a set of n processes, and a state S, S is safe if and only if there is a sequence of the processes, denoted $< P_1, P_2, \ldots, P_n >$ which satisfies the condition that for any P_i, all its resources can be satisfied by the sum of resources held in state S plus all the resources held by processes P_1 thru P_{i-1}. The intuition behind this definition is that by making the processes execute and terminate according to the order they have in the sequence, it is possible for any process to eventually receive all the resources it has requested. If in a given state S, for the current set of processes, such a sequence cannot be satisfied, S is defined to be unsafe. The next sections describe two deadlock avoidance algorithms which ensure a transition from a safe state to a safe state. To summarize

> The set of deadlocked states is a subset of the set of unsafe states
>
> A safe state is not a deadlock state

1.2.2 Holt Graph algorithm

If there is only one instance of each resource type, then a modified *Holt* graph algorithm can be used to ensure that the system is always in a safe state. First an *augmented Holt* graph(V,E) is defined as follows:

> V is partitioned into two types of nodes: $P_1 \ldots P_n$ representing the processes in the system and $R_1 \ldots R_n$ representing the resources in the system.

- The edges are one of three types: (the first two are already present in the *Holt* graph and the third class of edges are the augmentation)

  > *Allocation edges* of the form $R_i -> P_j$ to indicate that resource i is allocated to process j

wait edges of the form $P_i- > R_j$ to indicate that process i has requested for resource j and is waiting for it.

claim edges of the form $P_i- > R_j$ is used to indicate that process i **may** request resource j some time in the future.

When process i actually requests resource j, the corresponding *claim edge* is converted to a *wait edge*. Resources must be claimed *a priori* in the system. In other words, before P_i starts executing, all its claim edges must appear in the *augmented Holt* graph. When a process wants to start up, it must put all its claim edges to the *augmented Holt* graph. If this does not lead to a cycle formation, then P_i is allowed to start up. If on the other hand, adding the claim edges of P_i leads to the formation of a cycle, then P_i is not started up and must wait until the time when the addition of its claim edges does not lead to a cycle.

1.2.3 Banker's Algorithm

A modified Holt scheme does not apply in a system where there are multiple instances of a particular resource. This leads to the necessity of using a less efficient deadlock avoidance algorithm called *Banker's algorithm*. The intuitive idea behind this algorithm is the following: When a process enters the system, it must declare the maximum amount of each resource type that it needs. When a particular set of resources is requested, the system determines if the allocation of those resources will lead to deadlock. If so, the resources are not allocated and the requesting process must wait. If the allocation of the resources leads to a safe state, then the resources can be allocated however.

Several data structures will be used to illustrate how the banker's algorithm. Let n be the number of processes and m be the number of resource types available. We'll use the following data structures:

- available[m]: available is an array of m elements where the jth element indicates the amount of resources of type j available.

- max[n][m]: is a matrix that defines what is the maximum number of resources any process can request. For example, the entry max[i][j] stores the maximum number of reources of type j that process i can request.

- alloc[n][m]: is a matrix that defines what are the amount of resources that are presently allocated: alloc[i][j] stores how many units of resource j are allocated to process i.

- need[n][m]: is a matrix that defines the remaining amount of resources that a process can request subject to the initial maximum specification. need[i][j] is the amount of resources of type j that process i may request.

It is easy to see that need[i][j] = max[i][j] - alloc[i][j];

Before we proceed further, we introduce some notation. For any two vectors, X and Y with the same number n of elements, we say that X $\leq$ Y, iff $\forall i, 1 \dots n$, X[i] ... Y[i]. Also, given a matrix N, we use N[i] to indicate the *ith* row of matrix N. For each of the matrices defined above, the ith row of that matrix is a vector consisting of m elements. Also, for two vectors X and Y with n elements each, X+Y is defined as setting each of X[i] to X[i] + Y[i].

The banker's algorithm can be defined as follows:

```
i = 1;
while (i <= n)
```

```
find a j such that (need[j] <= available)
if no such j exists,
  declare present state to be unsafe and break
else
  available = available + alloc[j]
```