Deadlock

Introduction

We now want to abstract the concept of deadlock, and apply it to all resources that a computer system may have. We will see that in some cases the resources can be shared without any chance of deadlock. However, in other cases it is possible for several processes to be waiting for resources that they will never be able to allocate.

Resources

For the purposes of our discussion a **resource** is an object that a process makes use of. A resource can be a piece of hardware such as

- tape drive
- disk drive
- printer
- etc.

or a piece of information such as

- a file
- a record within a file
- a shared variable
- a critical section
- etc.

A computer typically has many different resources. In some cases there may be many instances of a resource of a given type (e.g. buffers), all of which are equivalent. A process needing one of these resources can use any one of them. In other cases there may be only one instance of a resource (e.g. CD-ROM drive with a particular CD).

Resources come in two flavors: preemptable and nonpreemptable. A **preemptable resource** is one which can be allocated to a given process for a period of time, then be allocated to another process and then be reallocated to the first process without any ill effects. Examples of preemptable resources include

- memory
- buffers
- CPU
- array processor
- etc.

A **nonpreemptable resource** cannot be taken from one process and given to another without side effects. One obvious example is a printer: certainly we would not want to take the printer away from one process and give it to another in the middle of a print job. (Actually, the THE operating system treated printers as preemptable resources -- the operators would have to sort the printed output to reassemble print jobs.)

As we shall see, deadlocks usually involve nonpreemptable resources. The usual sequence of events that occur

as a resource is used is

1. **Request the resource**. One of two things can happen when a resource is requested: the request can be granted immediately (if the resource is available) or it can be postponed (or blocked) until a later time.

2. **Use the resource**. Once the resource has been acquired, it can be used.

3. **Release the resource**. When the process no longer needs the resource it releases it. Usually it is released as soon as possible but in most systems there is nothing to enforce this policy.

Definition

Tanenbaum and Woodhull define deadlock 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.*

Notice that it is possible for a single process to become deadlocked if it is waiting for an event that only it can cause. However, it is usually the case that one process is waiting for a resource that is held by another process. If the other process is itself waiting for a resource held by the first then deadlock occurs.

Note that deadlock is different from **starvation**. Starvation is the problem that occurs when a process is waiting for a resource that is allocated to other processes, released, and then allocated again to some process other than the one that is starving.

Conditions for Deadlock

In 1971 Coffman et al. showed that four conditions must hold in order for deadlock to occur:

1. **Mutual exclusions**: Each resource can be assigned to only one process at a time and is either assigned or available.

2. **Hold and wait**: Processes that currently hold resources can request and wait for additional resources.

3. **No preemption**: Resources previously granted cannot be taken away from the processes that hold them; they must be released by the holding process.

4. **Circular wait**: There must be a circular chain of two or more processes, each waiting for a resource held by the previous member of the chain.

All four of these conditions must hold in order for deadlock to occur.

Diagramming Deadlocks

We can use a directed graph called a **resource-allocation graph** to characterize a deadlock situation.

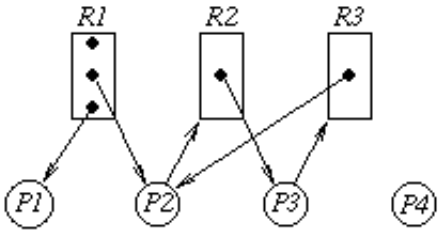
- Vertices represent resources and processes. We will use **rectangular vertices** to represent **resource**

classes, with dots within the vertex to represent instances of the resource. **Circular vertices** will be used for **processes**.

- A directed edge from a process vertex to a resource vertex indicates that the process has requested an instance of the resource. This is called a **request edge**.

  A directed edge from a dot within a resource vertex (an instance of the resource) to a process vertex indicates that the resource has been allocated to the process. This is called a **assignment edge**

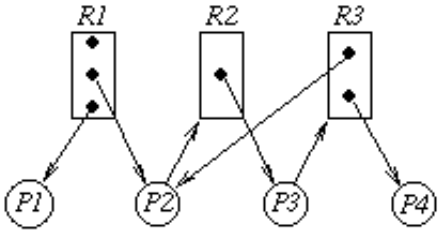
For example, consider the following graph:



In this example the class represented by R1 has three instances of the resource associated with it. P1 and P2 each have allocated an instance of a resource of type R1. P2 has requested an instance of resource type R2, but is waiting since the only instance is held by P3. P3 is in turn requesting a resource of type R3 but the only instance of that type is held by P2. The situation depicted in this graph is deadlocked. P2 is waiting for a resource held by P3, and P3 is waiting for a resource held by P2. This can be shown to be a cycle with

P2 -> R2 -> P3 -> R3 -> P2

Now consider the graph



This is just the same as the case before except now there are two instances of resource type R3. Both are currently allocated so that P2 and P3 are waiting just as before. This graph also has the cycle

P2 -> R2 -> P3 -> R3 -> P2

However, this is not a deadlock situation. As soon as P4 finishes with it's instance of R3 it will release it and P3 can allocate it. Even though there is a cycle, this situation does not meet the circular wait criterion for deadlock to occur: there is an instance of a requested resource that is not held by a process in the cycle (P4 holding an instance of R3).

Ways we can deal with Deadlock

There are four different approaches to dealing with the deadlock problem:

1. Ignore the problem
2. Detect deadlock when it occurs, and recover from it
3. Prevent deadlock from occurring
4. Dynamically avoiding deadlock

Ignore the Deadlock Problem

This is the approach taken by most operating systems. In fact, this is really an instance of the second approach: deadlock detection and recovery. This is because a human operator will often notice the deadlocked condition (detection) and kill the offending process(es) or restart the system (recovery). The implicit assumption here is that deadlock is unlikely to occur, and when it does it is relatively easy (and inexpensive) to recover from.

This is not quite as bad as it sounds. As we will see, handing the deadlock problem is complicated and is often inconvenient. If it is the case that a given system is likely to suffer from deadlock infrequently (once a week, once a month, etc.) but for other reasons it is restarted at least that frequently (e.g., it is restarted every morning) then it's probably not worth much effort to worry about deadlock.

Another consideration is exactly what types of processes are suffering from deadlock. If user processes are likely to be the only processes that become deadlocked, and they can be killed and restarted easily, then deadlock detection, avoidance or prevention is not mandatory. However, if it is kernel processes that are becoming deadlocked, so that the system is unusable, then it is probably worth some effort to eliminate the deadlock problem.

Deadlock Prevention

1. Conceptually, the cleanest way to deal with deadlock is to prevent it by insuring that one of the conditions for deadlock does not hold. In practice, however, deadlock prevention schemes impose restrictions that may not be tolerable - so working systems often combine deadlock prevention for some resources with one of the other two schemes for others.

2. Theoretically, deadlock can be prevented by **denying the mutual exclusion precondition**. This is usually not possible in practice, though the use of spooling to convert serially reusable devices into seemingly shareable virtual devices could be regarded as an attack at this point. (We could also regard it as a multiplication of the number of printers to the point where each process can have all it needs.)

3. Deadlock can be prevented by **denying the hold and wait precondition**.

    A. One approach is as follows: Require that a process request all the resources that it needs in one single request at process startup. The system will not grant any resource in the list until it can grant all of them.

    B. A less restrictive approach is to allow a process to request resources only when it is currently holding no resources. Thus, if a process needs a new resource, it must first yield all the resources it has and then put in its request (which might include a request for the reallocation of a resource it just gave up.)

C. Problems with this approach:

a. It can lead to processes holding resources when they don't need them, thus reducing resource utilization. This is especially serious if a process does not know what resources it will actually need for a given run until it has started working on the data. With this scheme, it must request up front all the resources it **might** need.

b. A process that needs several popular resources might starve while processes that need a smaller number of these resources keep taking them away.

4. Deadlock can be prevented by **denying the no preemption precondition**.

A. One way to implement this is to stipulate that any process that is forced to wait for some resource will have any resources it already possesses taken away from it. The wait for a single resource is then converted into a wait for a list of resources including both those it had and the one it now needs.

Example: A process holds a tape drive and requests a line printer. If a line printer is not available, the tape drive is taken away and the process is put into a state of waiting for both a tape drive and a line printer.

B. Again, this can be made somewhat less severe. A process that is waiting for some resource can hold them as long as another process does not need them. But if another process should request a resource held by the waiting process, the resource is preempted and the waiting process must now wait for both the original resource it wanted and the resource that was taken away.

C. This scheme also has problems:

a. It only works if the resources are preemptible. If a process has printed output on a line printer and is waiting for some other resource before it can generate more output, then the line printer really cannot be taken away without messing up the output. (This assumes the approach used by the THE system is not practical, as it often is not.)

b. This scheme can also lead to starvation for a process that needs several popular resources at the same time, since it may keep losing the resources it gets because they don't all become available at the same time.

5. Deadlock can be prevented by **making circular wait impossible**.

A. The basic approach in this scheme relies on resource ordering. With each unique type of resource (printer, tape etc.) we associate a unique number. For example, card readers might be assigned number 1, tapes number 2, and printers number 3. We require that a process request resources in increasing order of resource number; i.e. all of its requests for readers must precede any request for tapes or printers; and any request for tapes must precede any request for printers. Further, if it requests multiple resources of the same type it must request them all at once; it cannot ask for a second tape after previously requesting a tape.

B. This restriction can be loosened; if a process releases all the high-numbered resources it holds it

may be allowed to request a lower-numbered resource. It still cannot request a resource of a given type if it already holds one or more resources of that type. That is, the number of the resource it requests must be strictly greater than the number of any it holds.

Example: a process requests a reader and a printer (in that order). Ordinarily, it would not now be eligible to request any other resources. But if it releases the printer (so it only holds a reader), it may be allowed to request any number of tapes (in a single request) or any number of printers (in a single request) or both (in the order first tapes, then printers.)

C. To see that this protocol does, in fact, make circular wait impossible, we use a **proof by contradiction**:

   Assume that the resource ordering protocol is being used, and a circular wait has resulted. This means that we have a process P0 waiting on a resource held by P1, and P1 is waiting on a resource held by P2, etc.

   a. Let R0 be the resource held by P0 which PN is waiting for, R1 be the resource held by P1 that P0 is waiting for.

   b. Let F0 be the number associated with resource R0, F1 be the number associated with R1 etc.

   c. Now since P0 is waiting for a resource R1 while holding a resource R0, it must be the case that F0 < F1. In like manner, F1 < F2, etc. So we have F0 < F1 < F2 ... < FN, and therefore, by transitivity, F0 < FN.

   d. But we also have process PN requesting a resource R0 held by P0, while holding a resource RN. Therefore, it must be that F0 > FN. Since this is a contradiction, our assumption that circular wait could arise is false.

D. Of course, this scheme has problems too:

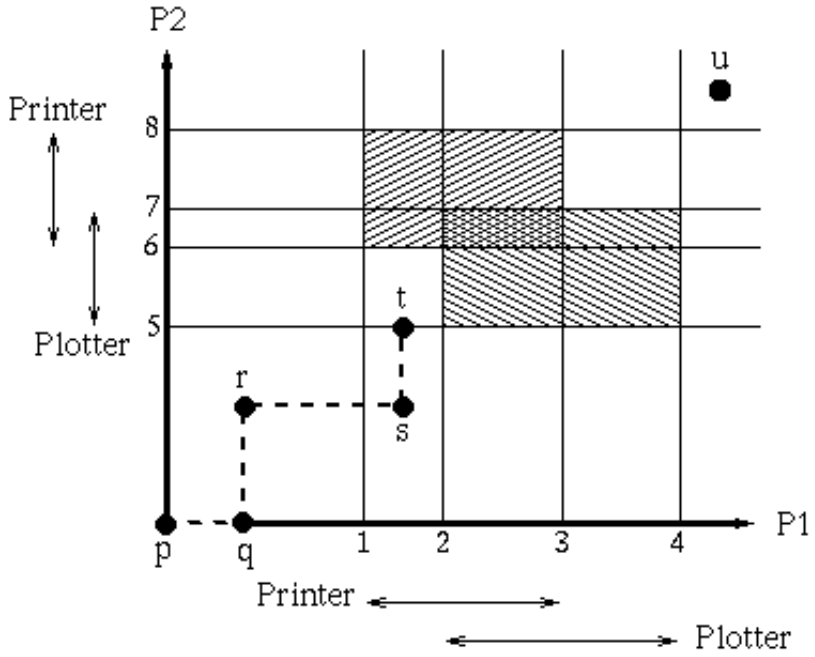
   a. The order of resource numbering may prove arbitrary and inconvenient. This is not necessarily too serious of a problem, since there are often natural ways of numbering resources. For example, processes generally use input devices (such as card readers) before output devices (such as printers). Tapes are often used for intermediate files that are created by the input phase of processing and retained through the output phase; so the example ordering above is not unreasonable.

   b. The order of numbering can force processes to request resources before they need them, thus reducing resource utilization. For example, with the above scheme a process that did some scratch work on tape before reading input from a card reader would still have to request the reader at the start of processing.

Deadlock Avoidance

1. One way to avoid deadlock in certain simple circumstances is by using **resource trajectories**. Assume

that there are two processes P1 and P2 and they both need to use the printer and the plotter (a single instance of each resource is available).



The horizontal and vertical axes represent the number of instructions executed by P1 and P2 respectively. The dashed line is a *resource trajectory*; it shows when the scheduler ran each process P1 and P2. Suppose that **p** is the initial situation and the **u** is the final point. Process P1 must pass lines 1, 2, 3, and 4 and process P2 must pass lines 5, 6, 7, and 8.

The shaded areas represent "impossible" regions; areas in which both processes hold one or both of the non-shareable resources. Notice that the trajectory must either move right or up. To avoid deadlock, once the trajectory is at **t** the scheduler must schedule process P1 to run until it passes line 4 before it lets process P2 run again. This is because if we enter the box bounded by lines 1, 4, 5 and 8 the trajectory must pass through one of the shaded "impossible" areas.

2. The best-known deadlock avoidance algorithm is the **banker's algorithm** (example transparency).

   A. The advance knowledge required is the maximum number of units of each type of resource that the process will claim at any one time.

      a. This can be declared explicitly up front by the programmer; or it may be determined implicitly from the job control cards in a batch environment.

      b. Any process which requests an allocation beyond its pre-declared maximum will be aborted.

   B. The OS maintains a vector and three matrices to keep track of actual and potential allocations:

```
Available: array[1..NoResourceTypes] of integer;
Max, Allocated, Need: array[1..NoProcesses, 1..NoResourceTypes] of integer;
```

a. The vector `Available` records, for each class of resource, the number of units of that resource currently available (i.e. not allocated to any process.)

b. The matrix `Max` contains one row per process and one column per class of resource. It records the pre-declared maximum number of units of each class that the process has said it might need.

c. The matrix `Allocated` is like `Max`. It records the number of units of each class of resource currently allocated to each process.

d. The matrix `Need` is like `Max`. It records the number of additional units of each class of resource that each process might need. Note that `Need[i,j] = Max[i,j] - Allocated[i,j]`, so only two of the three matrices need actually be stored.

C. We can model a request for new resources as a vector `Request`, in which each element represents the number of additional units of some class of resource that the process is now requesting. When a request comes in from some process `p`, the system handles it as follows:

a. If `Need[p,j] < Request[j]` for any `j`, then the process has made an illegal request and must be aborted.

b. If `Available[j] < Request[j]` for any `j`, then the process must wait.

c. If neither of the above hold, then it is possible to grant the request, provided that the resultant state is safe. To determine this, the OS pretends to grant the request and then checks the resultant state for safety:

Deadlock Detection and Recovery

One way to detect deadlock is for the system to monitor the requests and releases of resources and to maintain an up-to-date resource allocation graph. If this is done then it is possible to look for cycles in the graph ($O(n^2)$ operations, where n is the number of vertices in the graph) and determine if they indicate that deadlock has occurred.

It is possible to approximate this behavior by monitoring processes and after they are observed to have been blocked for some predetermined period of time it is assumed that they are deadlocked.

Once deadlock is detected (or assumed to have been detected) some recovery action is necessary to break the deadlock. Often this is done by killing and restarting one or more of the deadlocked processes. **Note**, however, that any modifications to files or other data structures made by the processes that were terminated must be undone or accounted for in some way.

$Id: deadlock.html,v 1.3 2000/02/27 15:17:51 senning Exp $

These notes are based in part on notes written by by R. Bjork of Gordon College and on the textbooks *Operating System Concepts* by Silberschatz and Galvin, Addison-Wesley, 1998 and *Operating Systems: Design and Implementation* by Tanenbaum and Woodhull, Prentice-Hall, 1997. Some material was gleaned from

http://www.cm.cf.ac.uk/User/O.F.Rana/os/lectureos12/index.html.