Deadlock

Definitions

The operating system manages resources that are (generally) *non-preemptable*.

However, managed resources are *serially reusable*. They can be reused by another process after the current process finishes (e.g., tape drives).

Resources are *discrete* and *bounded*, meaning that they are allocated in integer units (e.g., 2 tape drives, not 1.2 drives).

Identical resources belong to *resource classes*. For instance, most processes don't care which tape drive they get out of the pool of identical drives.

The *resource manager* fields requests from user processes to allocate and release resources. The manager has two options:

1. if the requested resource is not available, it can block the requesting process until the request can be granted
2. it can deny the request and let the user decide how to handle the situation

Typical Usage

Typically, processes use resources as follows:

```
Get Resource; /* may block */
Use Resource;
Release Resource;
```

Premature Termination

If the process terminates before it has released its resources, the resource manager can:

1. return the resources to the pool of unused resources. Some clean up operation may be needed, such as rewinding a tape drive.
2. refuse allocate the resource to another process until notified that it is safe to do so. In particular, some resources may be left in an inconsistent state (e.g., database inconsistencies).

Deadlock

Canonical example: Two processes A & B need two tape drives sometime during their execution. The system has only two drives. The resource manager could allocate resources as follows:

1. A requests and gets one drive
2. B requests and gets one drive
3. A requests another drive, and resource manager blocks request
4. B requests another drive and blocks as well

A *deadlock* has occurred, because neither process can proceed.

Of course, the resource manager could grant requests differently. For instance:

1. A requests and gets one drive
2. B requests a drive, but the resource manager blocks the request
3. A requests and gets a second drive
4. A finishes execution and releases both drives
5. resource manager grants allows B's request to proceed

Necessary Conditions for Deadlock

Deadlock can occur only if the following conditions hold:

1. Mutual exclusion condition. Each resource is assigned to a process or is available.
2. Hold and wait condition. Processes holding resources can request new resources.
3. No preemption condition. Resources cannot be taken from a process; that is, a resource given to a process cannot be taken back
4. Circular wait condition. A set of processes are in a circular wait; that is, there is a set of processes $\{ p_0, p_1, \ldots, p_n \}$ where p_i is waiting for a resource held by p_{i+1}, and p_n is waiting for a resource held by p_0.

Designing algorithms that prevent deadlock involves preventing one or more of the above conditions.

Resource Graph

When considering deadlock, it is useful to draw a *resource graph* that shows processes and resources. In a resource graph:

- circles denote processes
- boxes denote resources
- an arrow connects a process to a resource if the process is waiting on the resource
- an arrow connects a resource to a process if the resource is allocated to the process

A deadlock has occurred if the resource graph contains a cycle.

Note: The resource manager designer is faced with a classic problem: we want to maximize concurrency, yet minimize the occurrence of deadlock.

Conservative policies deny resources even when they are available. However, they reduce the likelyhood of deadlock at the expense of reduced concurrency

Liberal policies allocate resources when they are available at the risk of entering a deadlock state later.

Dining Philosophers Problem

Classic problem to demonstrate both deadlock and starvation.

Five philosophers seated around a table with ten chopsticks and five rice bowls. Need two chopsticks to eat rice. Philosophers alternate between thinking and eating. Want to grant chopsticks while avoiding both deadlock and starvation.

What happens with the following straightforward solution:

```
Philospher(which)
int which;        /* index of philospher */
Resource LeftChopstick, RightChopstick;
{
    while (1) {
        Think;        /* for some amount of time */
        GetResource(LeftChopstick);
        GetResource(RightChopstick);
        Eat;
        ReleaseResource(LeftChopstick);
        ReleaseResource(RightChopstick);
    }
}
```

Serialization

The most conservative approach is called *serialization*. No two processes are allowed to allocate *any* resources concurrently.

Serialization works because all processes must wait for the one process holding resources to complete.

Example: Suppose two processes both need two tape drives. Once one drive has been allocated to the first process, the second process would not be granted any request until the first process releases all the resources it holds.

Disadvantage:

- sharply reduced concurrency. Resources may include tape drives, files, additional memory, etc.
- may lead to *starvation*. Blocked process may never be allowed to continue!

One-Shot Allocation

To avoid cycles in the resource graph, require processes to acquire all resources at one time. That way, the resource manager has sufficient information to determine if granting the request can lead to a deadlock.

Consider the Dining Philosophers problem. Philosophers would ask for both chopsticks at the same time, and return them both at the same time:

- deadlock cannot occur
- however, starvation can. Philosophers 1 and 3 might alternate between eating and thinking, but if they never think at the same time, philosopher 2 goes hungry.

Disadvantage:

- too conservative to be practical; indeed, many processes cannot predict in advance what resources they will need. For instance, a process may not know how many tape drives it needs until it reads data from a tape.

Hierarchical Allocation

Observation: we can eliminate cycles in a resource graph by assigning increasing numbers to resources, and disallowing arcs from higher numbered resources to lower numbered ones.

Requirement: a process may only request resources at a higher level than any resources it currently holds.

Example: A system with tape drives, printers, and plotters might assign each resource at level 1, 2, and 3 respectively.

A process that has one tape drive may request either printers or plotters (but not another tape drive).

A process that has one plotter may not request any additional resources.

Hierarchical allocation solves the deadlock problem because deadlock can only occur if a cycle appears in the resource graph. However, a cycle can occur only if an arc goes from a higher numbered resource to a lower numbered one.

Disadvantage:

- process must allocate resources in predefined order, but the order may not reflect the order in which the process wants to use them

Advance-Claim Algorithm/Banker's Algorithm

Processes make an advance *claim* before acquiring resources.

- the claim is an upper bound on the types and numbers of resources the process will ever use.
- the resource manager will reject any future requests that would raise the process's resource allocations above its claim

The resource manager records the current *allocation state* for each resource class.

Realizable State

An allocation state is called *realizable* if:

- no one claim is for more that the total resources available
- no process is holding more than its claim
- the sum of the currently allocated resources (within a class) does not exceed the total available resources

The resource manager wants to insure that *unrealizable* states are never entered.

Safe State

A realizable state is called *safe* if there is a sequence of processes, called a *safe sequence* that satisfy:

- the first process in the sequence can finish, even if it requests all the resources its claim allows
- the second process in the sequence can finish, if the first finishes and releases all resources it currently has
- the ith process can finish if all previous processes complete

As long as we are in a safe state, we can always run the processes in the order given by the sequence, thereby preventing deadlock. (Hopefully, we can do even better!)

Example:

Process	Holding	Claims
A	4	6
B	2	7
C	4	11
Unallocated	2	

A can finish (although it can request 2 more drives, they are unallocated)

B can finish. If it asks for 5 more, it must wait until A completes.

C can finish. If it asks for 7 more, it must wait until A & B complete.

Example:

Process	Holding	Claims
A	4	6
B	2	9
C	4	11
Unallocated	2	

A can finish, but B will not, if it requests 7 drives.

Advance-Claim Algorithm: never allocate a request if it causes the current allocation state to become unsafe (also called the *banker's algorithm*).

Observation: if the state is safe, and if some process A can finish given currently available resources, there is a safe sequence starting with A.

Algorithm:

```
S = { set of all processes };
while (S != EMPTY_SET) {
   Find A, an element in S that can finish
   if (no A exists)
       state is unsafe
   remove A from S;
   add A's resources to the unallocated pool
}
state is safe
```

Note: most liberal policy we have seen that prevents deadlock.

Disadvantages:

- processes must make advance claims, but may not know in advance what their resource requirements are.
- runs in time proportional to $m * n^2$, where m is the number of resources and n is the number of processes

Deadlock Detection

So far, we have considered *deadlock avoidance* algorithms. The strategy of discovering deadlocks after they have occurred is called *deadlock detection*.

If we suspect a deadlock has occurred, build a resource graph and look for cycles. However, our definition of a resource graph must be modified slightly:

- all instances of a resource in a resource class are placed in one node
- an arrow from a process to a resource indicates that the process is waiting on any one of the resources within the class
- an arrow from a resource to a process indicates that the process has obtained one (or more) units of the resource.
- deadlocks no longer given by cycles in graph
- presence of a *knot* in the graph indicates deadlock

A knot is a set of vertices (processes and resources) such that when starting at any vertex in the knot, paths lead to all vertices in the knot, but to no vertices outside the knot.

Detecting knots:

- start with the generalized resource graph
- remove processes that are not waiting an anything
- remove processes that are waiting on resources that are not fully allocated
- repeat process until no more processes can be removed
- if algorithm is able to remove all processes, no knot is present

When would be a good time to check for deadlock?

- when a process requests a resource
- VMS solution: start timer whenever a process blocks waiting for a resource. If the request is still blocked 10 seconds later, start deadlock detection algorithm
- UNIX takes the path of least resistance. It does not detect deadlock, and it simply denies requests that cannot be satisfied given the resources on hand. Ostrich algorithm. Stick your head in the sand and don't worry about it. Mathematicians don't like it all—could have deadlock. Engineers—what are the chances of it happening.

Once deadlock has been detected, *deadlock recovery* choses a victim process that will break the deadlock when terminated.

Issues:

- terminating a process might injure data. Such processes could indicate to the operating system that they should be terminated only if no better victim exists.
- resource manager might take into account process priorities, amount of CPU time the processes have used, etc.

Starvation

A process *bigjob* needing many resources may starve:

- granting resources to *bigjob* may lead to an unsafe state
- to improve concurrency, however, the resource manager allocates the resources to smaller jobs that keep jumping ahead of *bigjob*

Possible solution:

- use timer to detect presence of starving jobs, and boost their priority in such a way that the resource manager favors them (e.g., by denying resources to the smaller jobs)