Deadlock Avoidance

Introduction

We focus a great deal of attention on the four necessary conditions for deadlock, because these form the basis for deadlock prevention. If we can ensure that any of the four necessary conditions will never be met, we have created a system where deadlock (involving those resources) is impossible. There are, however, common situations in which:

- mutual exclusion is fundamental.
- hold and block are inevitable.
- preemption is unacceptable.
- the resource dependency networks are imponderable.

For many such cases, deadlock avoidance may be an easy and effective solution. Consider the following deadlock situation:

- main memory is exhausted.
- we need to swap some processes out to secondary storage to free up memory.
- swapping processes out involves the creation of new I/O requests descriptors, which must be allocated from main memory.

The problem, in this case, is not a particular resource dependency graph, but merely that we exhausted a critical resource. There are many similar situations where:

- some process will free up resources when it completes.
- but the process needs more resources in order to complete.

In situations like these, it is common to keep track of free resources, and refuse to grant requests that would put the system into a dangerously resource-depleted state. Making such case-by-case decisions to avoid deadlocks is called *deadlock avoidance*.

Reservations

Declining to grant requests that would put the system into an unsafely resource-depleted state is enough to prevent deadlock. But the failure of a random allocation request (in mid-operation) might be difficult to gracefully handle. For this reason, it is common to ask processes to reserve their resources before they actually need them.

Consider the *sbrk(2)* system call. It does not actually allocate any more memory to the process. It requests the operating system to change the size of the data segment in the process' virtual address space. The actual memory assignments will not happen until the process begins referencing those newly authorized pages. If we can determine that the requested reservation would over-tax memory, we can return an error from the *sbrk* system call, which the process can then decide how to handle. If, however, we waited until a page was referenced before we decided we did not have sufficient memory, we might have no alternative but to kill the process.

This approach is not limited to memory. For example ...

- we could refuse to create new files when file system space gets low.
- we could refuse to create new processes if we found ourselves thrashing due to pressure on main memory.
- we could refuse to create or bind sockets when network traffic saturates our service level agreement.

Unlike *malloc(3)*, these other operations do not tell us how much new resource the process wants to consume. But in each of these cases there is a request (to which we can return an error) before we reach actual resource exhaustion. And it is this failable request that gives us the opportunity to consider, and avoid a resource-exhaustion deadlock.

Over-Booking

In most situations, it is unlikely that all clients will simultaneously request their maximum resource reservations. For this reason, it is often considered *relatively safe* to grant somewhat more reservations than we actually have the resources to fulfill. The reward for over-booking is that we can get more work done with the same resources. The danger is that there might a demand that we cannot gracefully handle.

- Air lines do this all the time ... which is why they occasionally offer cash and free trips to anyone who is willing/able to take a later flight.
- Required network bandwidth is routinely estimated based on the expected traffic distribution. Such a network might be able to handle 25% more traffic while still maintaining its Service Level Agreements 99.9% of the time.
- In operating systems, the notion of killing random processes is so abhorrent that most operating systems simply refuse to over-book. In fact, it is common to under-book (e.g. reserve the last 10% for emergency/super-user use).

Dealing with Rejection

What should a process do when some resource allocation request (e.g. a call to *malloc(3)*) fails?

- A simple program might log an error message and exit.
- A stubborn program might continue retrying the request (in hope that the problem is transient).
- A more robust program might return errors for requests that cannot be processed (for want of resources), but continue trying to serve new requests (in the hope that the problem is transient).
- A more civic-minded program might attempt to reduce its resource use (and therefore the number of requests it can serve).

There are many possible responses, and different responses make sense in different situations. But the key here is that, since the allocation request failed with a clean error, the process has the opportunity to try to manage the situation in the most graceful possible way.