Deadlock

Goal

To examine what causes deadlock, and what to do about it.

Deadlock

The *resource manager* is that part of the kernel responsible for managing resources. Its process interface has two functions:

- *request*: asks that a process be given a resource; and
- *release*: informs the resource manager that the process no longer needs a resource it has been allocated.

example: A system has two tape drives *s* and *t*; processes *p* and *q* will each need both. The following occurs:

p requests tape drive *s*; the resource manager allocates it
q requests tape drive *t*; the resource manager allocates it
p requests tape drive *t*; the resource manager blocks *p* until that drive becomes available
*q*requests tape drive *s*; the resource manager blocks *q* until that drive becomes available

Now *p* and *q* are *deadlocked* (cite Kansas train crossing law here)

Deadlock vs. starvation:

deadlock occurs when a needed resource is never available for reassignment.

starvation occurs when a needed resource is available for reassignment but is never assigned to the process requesting i t

example: forks in the dining philosophers problem.

Approaches to the problem

There are three main approaches:

- *liberal*: Whenever possible, grant the request; if it cannot be granted, block the requestor until it can.
- *conservative*: Be willing to deny an available resource on occasion to prevent deadlock.
- *serialization*: processes cannot hold resources concurrently; so if one process requests and is granted a resource, no other process can acquire another resource.
  example: in the earlier example, *q*'s request for *t* would have been denied.

Resource types

- *Reusable* (also called *serially reusuable*) resources have a fixed total inventory: none are created, and none destroyed. Units are requested and acquired from a pool of available units and after use are returned to the pool where other processes can get them.
  examples: processors, memory, tape drives, *etc*.
- *Consumable* resources have no fixed number of units, but are created (produced) or acquired (consumed) as needed. An unblocked producer may release any number of units which become immediately available; once acquired, units cease to exist.
  examples: messages, information in I/O buffers, *etc*.
  We will not discuss deadlock analysis of consumable resources.

How to Deal with Deadlock (Policies)

(1) *Ignore it* (Tanenbaum calls this the “ostrich approach”): used by UNIX; okay if deadlocks rare and users know how to recover.
(2) *detection and recovery*: determine when the system is deadlocked, and recover; okay if deadlocks are infrequent and cost of recovery is low;
(3) *prevention*: ensure deadlock can **never** occur; if granting a request could cause deadlock later on, deny the request. This means ensuring one of the following four conditions fails (all must hold for deadlock to occur):
   - *mutual exclusion*: when a process is using a resource, no other process can use it.
   - *no preemption*: resources will not be taken from a process holding them.
   - *circular wait* or *resource waiting*: blocked processes form a circular chain, with each holding a resource requested by another member of the chain and requesting a resource held by another member of the chain.
   - *hold and wait* or *partial allocation*: a process may hold resources while requesting others.

   This policy degrades utilization of resources, but is acceptable if deadlocks are unacceptable.
(4) *avoidance*: use knowledge of the process' future behavior to constrain the pattern of resource allocation.

Deadlock Prevention

These schemes use the idea that, as a safe state is one that can never lead to deadlock, the system should be restricted so that *all* states are safe. Typical designs:

(1) only 1 process at a time may hold resources, which leads to a single-programming environment;

(2) each process must request and acquire all the resources it may need at one time. But this means that things may be requested unnecessarily, or allocated *long* before used.

(3) resources are ordered, and constraints are placed upon requesting resources in different classes of the ordering (this is called *hierarchical ordering* or an *ordered resource policy*):

- divide resources into k classes; a process can request allocations from class Ki if and only if it f has no allocations from classes Ki+1, ..., Kk.

As with (2), some resources must be allocated in advance of their need.

Deadlock Avoidance

Banker's Algorithm

This determines if the system is in a safe or unsafe state by trying to finish.

example: There are 10 resource units and 3 processes. P wants to acquire another resource unit. If the request is granted, the following will be the state:

P has 4 units and needs 4 more
Q has 2 units and needs 1 more
R has 2 units and needs 7 more
2 units are available
(1) satisfy Q:
P has 4 units and needs 4 more
R has 2 units and needs 7 more
4 units are available
(2) satisfy P:
R has 2 units and needs 7 more
8 units are available
(3) satisfy R; all processes finish.

Therefore the initial state is safe and the request can be granted.

example: Same request, but if granted the state would be:

P has 4 units and needs 4 more
Q has 2 units and needs 1 more
R has 3 units and needs 6 more
1 unit is available
(1) satisfy Q:
P has 4 units and needs 4 more
R has 2 units and needs 7 more
3 units are available

P and R cannot finish, therefore the initial state is unsafe. The request will be denied.

Problems: Five big ones:

(1) the Banker's algorithm requires a fixed number of resources
If something goes off line for repair or maintenance, the system may be put into an unsafe state without any action by the processes;
(2) the Banker's algorithm requires a fixed number of processes
This is unreasonable, especially in time sharing systems.
(3) the Banker's algorithm guarantees all requests will be granted in a finite time

But printing your program (due today) next year grants your request in a finite time. You need a better guarantee than that!

(4) the Banker's algorithm requires jobs to release their resources in a finite time
Suppose a process grabs a resource and then blocks indefinitely, waiting for an external event to occur. Again, you need a better guarantee that that!

(5) the Banker's algorithm requires users to know and state process needs in advance.