Deadlocks

Module 21

Resources

- Computers are really just a collection of resources…
  - The Operating System allocates / grants access to resources
  - Resources might be hardware
  - Resources might be software (the file system, a mutex, etc.)

Resource Acquisition

- Through system calls:
  - Request the resource
  - Use the resource
  - Release the resource

```
semaphore r1;

// work with resource
down(&r1);
use_resource();
up(&r2);
```

```
file_descriptor fd;

// use device
fd = open(device_num);
use_device();
close(fd);
```

Deadlocks

- A deadlock occurs when:
  - Each process in a set is waiting for an event
  - Those event can only be caused by another process in the set.
- Deadlocks occur within a set of process, and at least two shared resources

Deadlock - basic example

Process A:

Give me Resource 1

Give me Resource 2

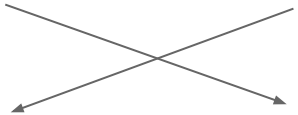
Work on resource 1 & 2

Process A:

Give me Resource 2

Give me Resource 1

Work on resource 1 & 2



- If Process A gets Resource 1, and is then interrupted, Process B may acquire Resource 2
- Neither can run… ever again.

Requirements

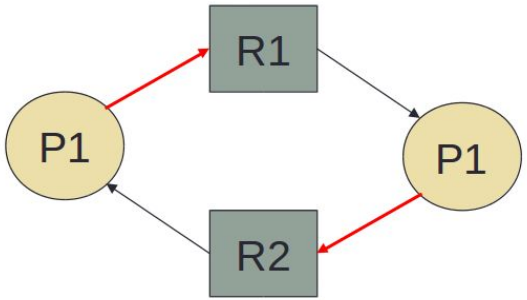
A set of process, and a set of policies, must meet the following conditions:

1) Mutual Exclusion is Guaranteed
2) Hold & Wait is Permitted
3) No Preemption of Resources is allowed
4) Circular Wait is Permitted

Modeling Resource Allocation

- Deadlocks can be defined as *cycles* in a graph
  - Resources & Processes are Nodes
  - Assignment Edges
  - Request Edges

Computers are good at finding cycles in graphs!



OS Responsibility

What can an OS do?

Option 1: Pretend this will never happen.

Most general purpose OS's actually do this

Handling Deadlocks

- In some systems (long running, critical uptime), we may need to handle deadlocks
- Often individual programs handle deadlock within their own threads/resources

- Eliminate them:
  - Prevention algorithms (attack requirements as matter of policy)
  - Avoid them (careful planning)
- Detect and Recover

Attacking Requirements

- Do not allow mutual exclusion

or

- Do not allow a process to ask for a resource while holding another

or

- Allow resources to be taken away at any time

or

- Detect circular wait and fail on resource request

Avoidance

- We can model “hypothetical” situations:
  - If I grant access to this resource, will there be a deadlock?
  - If so, cause the “open” call to fail
- We can hold “reserve” resources to avoid unsafe situations:
  - Banker’s algorithm - don’t allocate if there is no way of recovering from “worst case” - which is that everyone opens resources

Banker's Algorithm

12 Tape Drives, 2 currently free

Process	Max resource possible to request	Current resources acquired
P0	10	5
P1	4	2
P2	9	3

The system is "safe" - if each process asks for ALL resources it could ask for, the situation can resolve itself (P1)

Banker's Algorithm

12 Tape Drives, 1 currently free

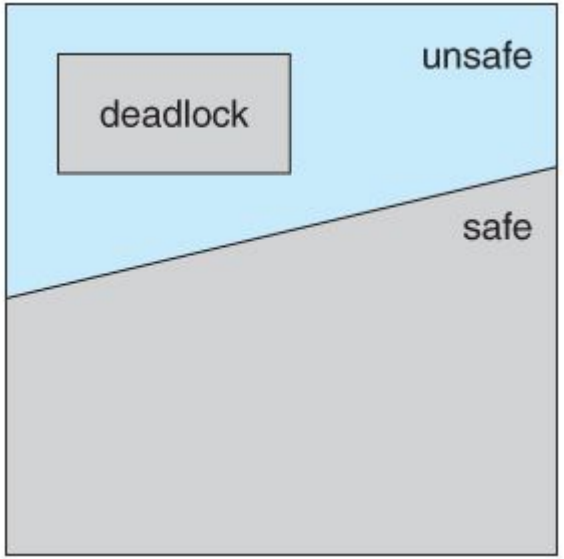
Process	Max resource possible to request	Current resources acquired
P0	10	5
P1	4	2
P2	9	**4**

If P2 asks for (and gets) 1 more drive, it would put us in an **unsafe** state

Disallow P2 from acquiring the resource

Banker's Algorithm

- It is a bit "conservative"
  - *Just like banks, most OS's throw caution to the wind and charge towards the unsafe assuming they won't be around to clean up the mess…*
- Algorithm is flawed - requires us to know how many total resources a process needs, in advance!



Recovery

- Avoiding Deadlocks usually put onerous restrictions (too conservative) on system
- How about we let them happen, and then recover?
  - Terminate offending process?
  - Preempt the resource?

Why do we care?

- If most OS's don't deal with deadlocks… why are we studying them?
- Deadlocks most often occur between threads ***within the same process***.
- As a user programmer, you probably **do know** how many resources each thread may need, and you probably can accurately implement avoidance algorithms!