# Deadlock

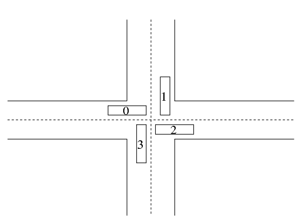
CS 537 - Introduction to Operating Systems

## Defining Deadlock

- Deadlock is a situation where 2 or more processes are unable to proceed because they are waiting for shared resources.
- Three necessary conditions for deadlock
  - able to hold more than one resource at a time
  - unwilling to give up resources
  - cycle
- Break any one of these three conditions and deadlock is avoided

## Example

- Imagine 4 cars at an intersection



## Example

- Lanes are resources.
- Deadlock has occurred because
  - each car is holding 2 resources (lanes)
  - none of the cars is willing to backup
  - car 0 waits for car 1 which waits for car 2 which waits for car 3 which waits for car 0
    - this is a cycle
- If any ONE of the above conditions can be broken, deadlock would be broken

## Dealing with Deadlock

- Three ways to deal with deadlock
  - never allow it to occur
  - allow it to occur, detect it, and break it
  - ignore it
    - this is the most common solution
    - requires programmers to write programs that don't allow deadlock to occur

## Not Allowing Deadlock to Occur

- Don't allow cycles to happen
- Force requests in specific order
  - for example, must requests resources in ascending order
  - Process A may have to wait for B, but B will never have to wait for A
- Must know in advance what resources are going to be used
  - or be willing and able to give up higher numbered resources to get a lower one

## Detecting Deadlock

- Basic idea
  - examine the system for cycles
  - find any job that can satisfy all of its requests
  - assume it finishes and gives its resources back to the system
  - repeat the process until
    - all processes can be shown to finish - no deadlock
    - two or more processes can't finish – deadlocked

## Detecting Deadlock

- Very expensive to check for deadlock
  - system has to stop all useful work to run an algorithm
- There are several deadlock detection algorithms
  - not used very often
  - we won't cover them

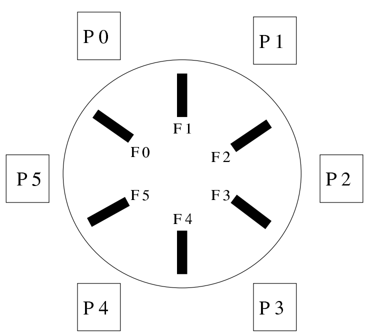
## Deadlock Recovery

- So what to do if deadlock is discovered?
  - OS can start deactivating processes
  - OS can revoke resources from processes
- Both of the above solutions will *eventually* end a deadlock
  - which processes to deactivate?
  - which resources to revoke?

## Dining Philosophers

- Philosophers sitting around a dining table
- Philosophers only eat and think
- Need two forks to eat
- Exactly as many forks as philosophers
- Before eating, a philosopher must pick up the fork to his right and left
- When done eating, each philosopher sets down both forks and goes back to thinking

## Dining Philosophers




## Dining Philosophers

- Only one philosopher can hold a fork at a time
- One major problem
- what if all philosophers decide to eat at once?
  - if they all pick up the right fork first, none of them can get the second fork to eat
  - deadlock

## Philosopher Deadlock Solutions

- Make every even numbered philosopher pick up the right fork first and every odd numbered philosopher pick up the left fork first
- Don't let them all eat at once
  - a philosopher has to enter a monitor to check if it is safe to eat
    - can only get into the monitor if no one else in it
  - each philosopher checks and sets some state indicating their condition

## Philosopher Deadlock Solution

```
enum { THINKING, HUNGRY, EATING };
monitor diningPhilosopher {
    int state[5];
    condition self[5];
    diningPhilosphers { for(int i=0; i<5; i++)   state[i] = THINKING; }
    pickup(int i) {
            state[i] = HUNGRY;
            test(i);
            if (state[i] != EATING)   self[i].wait;
    }
    putDown(int i) {
            state[i] = THINKING;
            test((i+5) % 6);
            test((i+1) % 6);
    }
    test(int i) {
            if( (state[(i + 5) % 6] != EATING) && (state[i] == HUNGRY)
                && (state[(i + 1) % 6] != EATING) ) {
                        state[i] = EATING;
                        self[i].signal;
            }
    }
}
```