CS 153
Design of Operating Systems

Fall 20

Lecture 7: Scheduling

Instructor: Chengyu Song

Multiprogramming

- Increase CPU utilization and job throughput by overlapping I/O and CPU activities
  - Doing this requires a combination of mechanisms and policy
- We have covered part of the mechanisms
  - Context switching, how and when it happens
  - Process queues and process states
  - Now meet the scheduler
- Now we'll look at the policies
  - Who, when, goals, …
- We'll refer to schedulable entities as jobs (standard usage) – could be processes, threads, people, etc.

Scheduling Overview

- Scheduler runs when we context switching among jobs to pick who runs next
  - Under what situation does this occur?
  - What should it do? Does it matter?
- Making this decision is called scheduling
- Now, we'll look at:
  - The goals of scheduling
  - Starvation
  - Various well-known scheduling algorithms
  - Standard Unix scheduling algorithm

Scheduler

- The scheduler is the OS module that manipulates the process queues, moving jobs to and from them
- The scheduling algorithm determines which jobs are chosen to run next, for how long, and what queues they wait on
- In general, the scheduler runs:
  - When a job switches from running to waiting
  - When an event occurs
    - Why?
  - When a job is created or terminated
  - When a job voluntarily yield the CPU

Scheduling Levels

- Scheduling works at two levels in an operating system
  1. Control multiprogramming level – number of jobs loaded into memory
     - » Moving jobs to/from memory is often called swapping
     - » Long term scheduler: infrequent
  2. To decide what job to run next
     - » Does it matter? What criteria?
     - » Short term scheduler: frequent
     - » We are concerned with this level of scheduling

Scheduling Styles

- Scheduler works differently in different systems
  - In preemptive systems the scheduler can interrupt a running job (involuntary context switch)
  - In non-preemptive systems, the scheduler waits for a running job to explicitly block (voluntary context switch)
  - What about preemptive kernel?
    - » Non-preemptive kernel disables maskable interrupts during event handling
    - » Preemptive kernel allows interrupts to be delivered during event handling → more responsive

Scheduling Goals

- What are some reasonable goals for a scheduler?
- Scheduling algorithms can have many different goals:
  - CPU utilization
  - Job throughput: # jobs/unit time
  - Turnaround time: $T_{finish} - T_{start}$
    - Normalized turnaround time = Turnaround time/process length
  - Avg Response time: avg time spent on ready queue
  - Avg Waiting time: avg time spent on wait queues (in sync)
- Batch systems
  - Strive for job throughput, turnaround time (supercomputers)
- Interactive systems
  - Strive to minimize response time for interactive jobs (PC)

Starvation

Starvation is a scheduling “non-goal”:

- Starvation: a job is prevented from making progress because other jobs have the resource it requires
  - Resource could be the CPU, or a lock (later in synchronization)
- Starvation usually a side effect of the scheduling
  - E.g., a high priority process always prevents a low priority process from running on the CPU
- Starvation can be a side effect of synchronization
  - E.g., one thread always beats another when acquiring a lock
  - E.g., constant supply of readers always blocks out writers

Job Characteristics

- Achieving the goals may require knowing the job
  - Past
    - When it arrived, how much progress has it made, how long has it run, has it been behaving nicely, …
  - Current
    - How many resources it uses, how many are left, …
  - Future
    - How much work is left? …
    - Important for some scheduling algorithm, but can we really know?
  - Type
    - GUI, I/O, realtime, …
  - Priority
  - etc

First In First Out (FIFO)

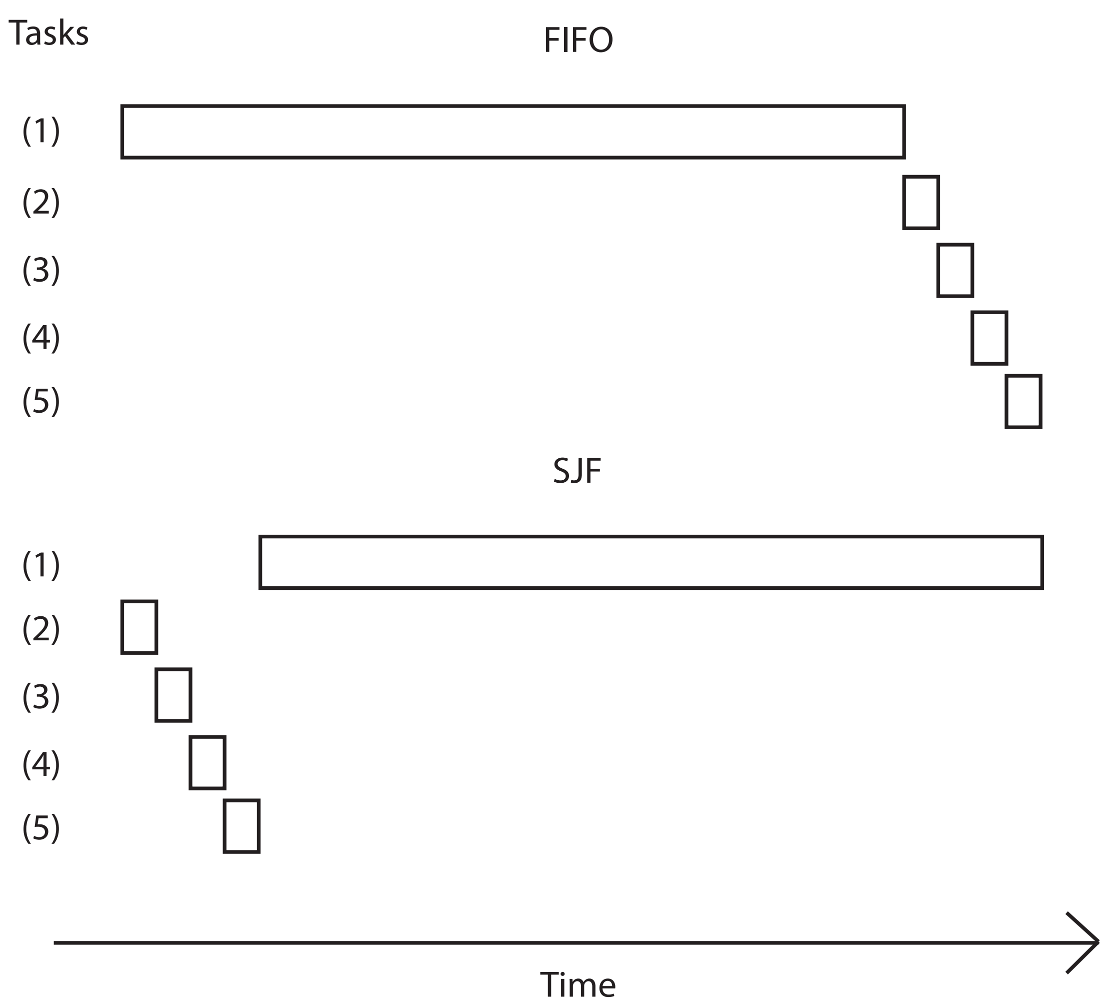
- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor
- Example: queues
  - Supermarket, banks, drive-through, …
- On what workloads is FIFO particularly bad?
  - Imagine being at supermarket to buy a drink of water, but get stuck behind someone with a huge cart (or two!)
    - …and who pays in pennies!
  - Can we do better?

Shortest Job First (SJF)

- Always do the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)

- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO? Next?
  - Which completes first in SJF? Next?

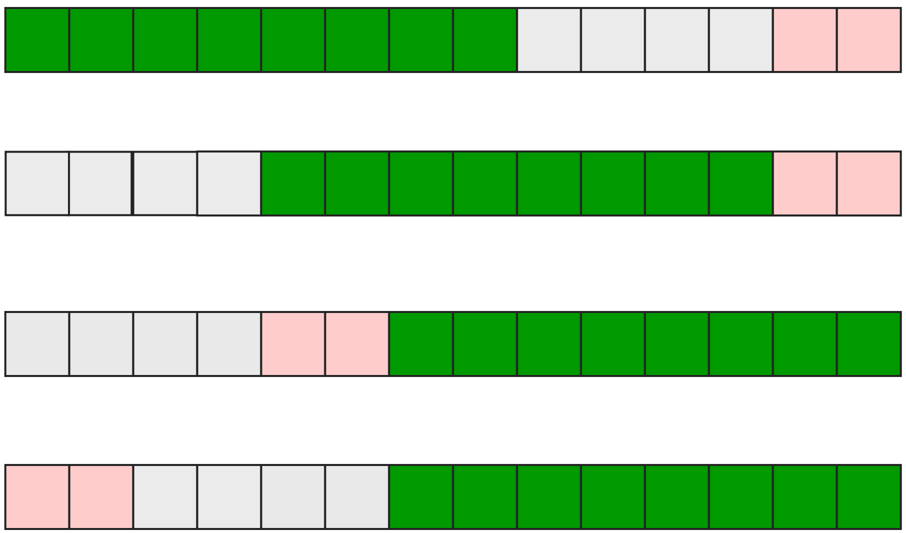
FIFO vs. SJF



What's the big deal? Don't they finish at the same time?

FIFO vs. SJF

- Assuming jobs arrives almost at the same time



ATT = (8+(8+4)+(8+4+2))/3 = 11.33

ATT = (4+(4+8)+(4+8+2))/3 = 10

ATT = (4+(4+2)+(4+2+8))/3 = 8

ATT = (2+(2+4)+(2+4+8))/3 = 7.33

FIFO vs. SJF

- Claim: SJF is optimal for average response time
  - Why?
- For what workloads is FIFO optimal?
  - For what is it pessimal (i.e., worst)?
- Does SJF have any downsides?

SJF

- Problems?
  - Impossible to know size of CPU burst
    - Like choosing person in line without looking inside basket/cart
  - How can you make a reasonable guess?
  - Can potentially starve

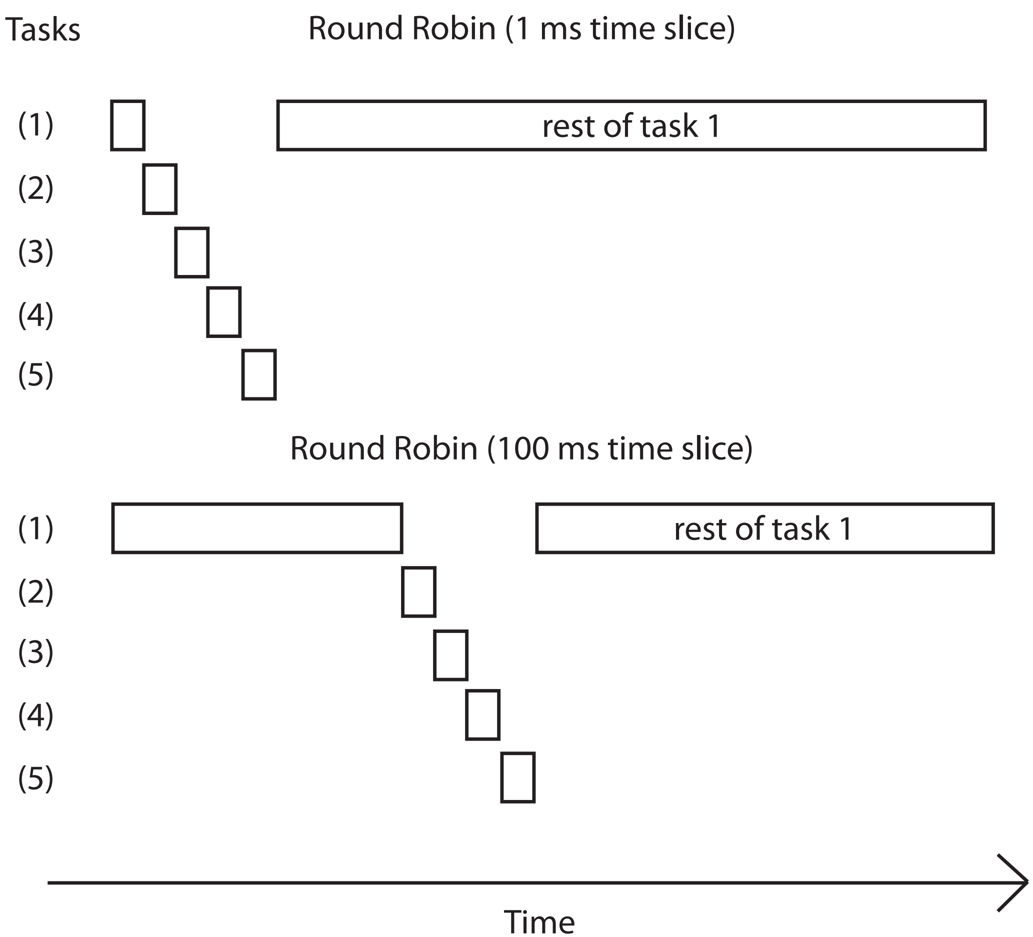
- Flavors
  - Can be either preemptive or non-preemptive
  - Preemptive SJF is called shortest remaining time first (SRTF)

Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn’t complete, it goes back in line

- Need to pick a time quantum
  - What if time quantum is too long?
    - » Infinite?
  - What if time quantum is too short?
    - » One instruction?

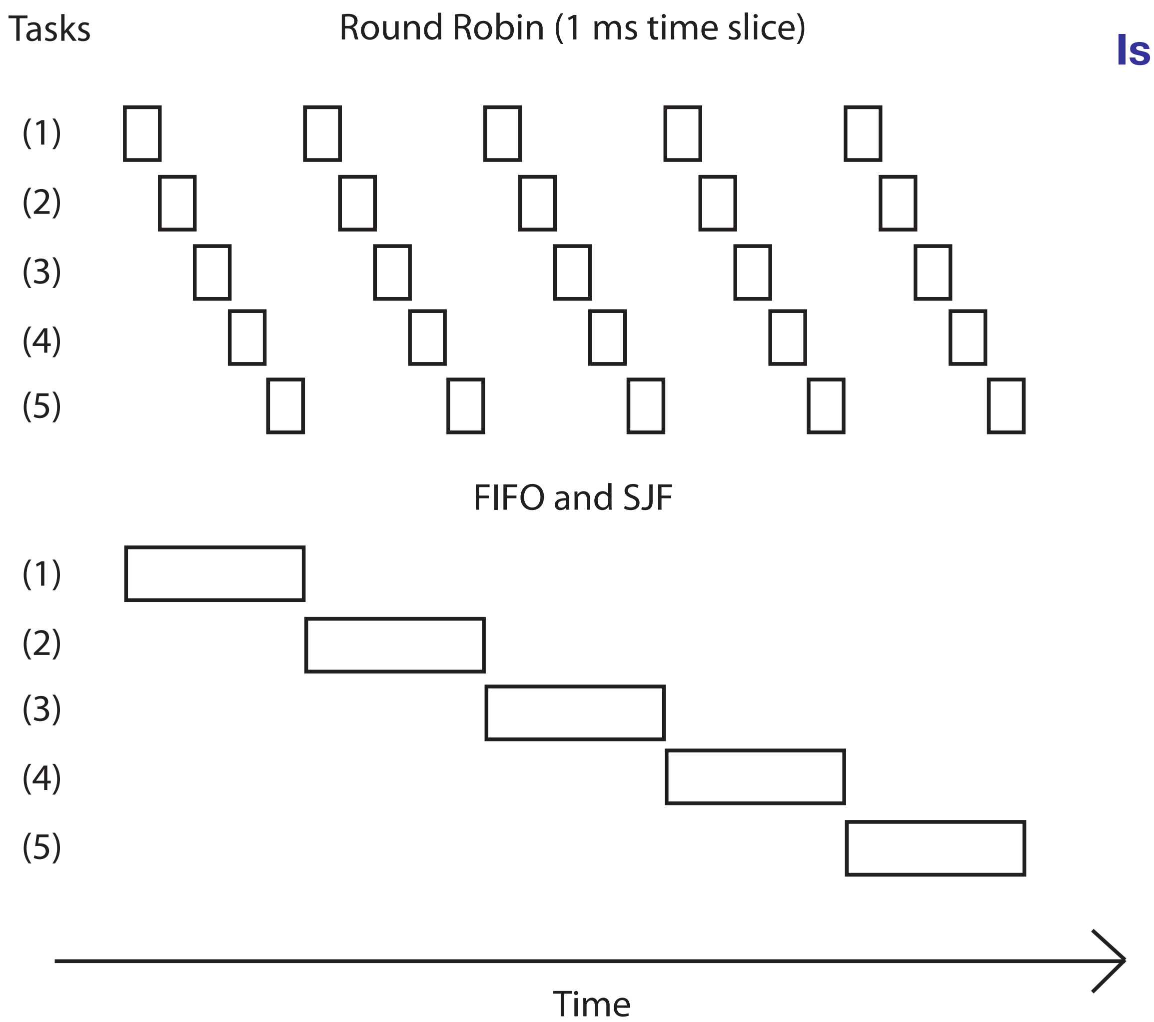
Round Robin


Tasks
Round Robin (1 ms time slice)
(1)
rest of task 1
(2)
(3)
(4)
(5)
Round Robin (100 ms time slice)
(1)
rest of task 1
(2)
(3)
(4)
(5)
Time

Round Robin vs. FIFO

- Many context switches can be costly
- Other than that, is Round Robin always better than FIFO?

Round Robin vs. FIFO



Is Round Robin always fair?

Problems of basic algorithms

- FIFO
  - Good: fairness; bad: turnaround time, response time
- SJF
  - Good: turnaround time, response time; bad: fairness, need to estimate run-time
- RR
  - Good: fairness, response time; bad: turnaround time
- Is there a scheduler that balances these issues better?
  - Challenge: limited information about a process in the beginning
  - Challenge: how to prevent gaming the scheduler to get more run-time

MLQ: combining algorithms

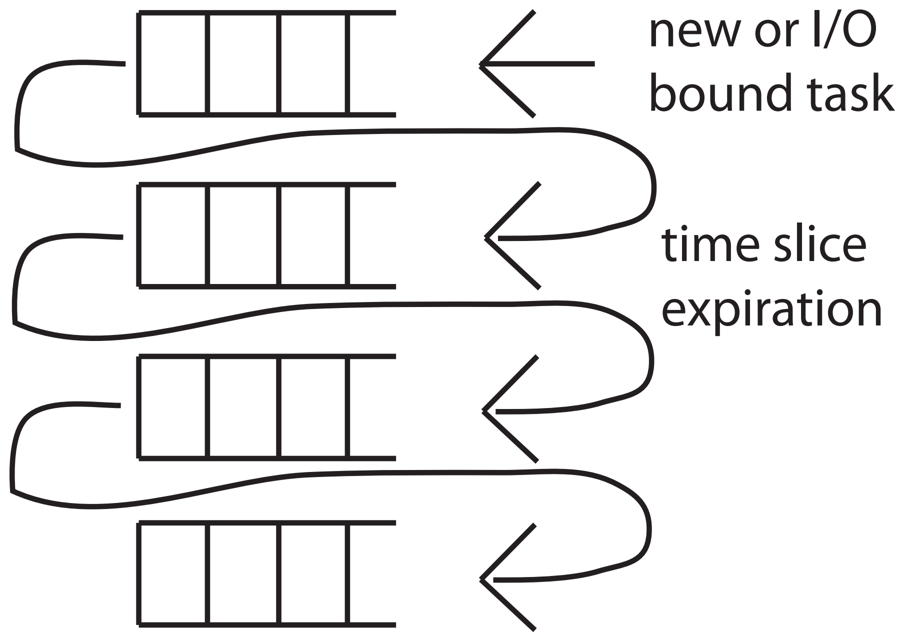
- Scheduling algorithms can be combined
  - Have multiple queues
  - Use a different algorithm for each queue
  - Move jobs among queues
- Example: Multiple-level feedback queues (MLFQ)
  - Multiple queues representing different job types
    - Interactive, CPU-bound, batch, system, etc.
  - Queues have priorities, jobs on same queue scheduled RR
  - Jobs can move among queues based upon execution history
    - Feedback: Switch from interactive to CPU-bound behavior

MLFQ

- Set of Round Robin queues
  - Each queue has a separate priority
- Higher priority queues have shorter time slices
  - Lower priority queues have longer time slices
  - Why?
- Scheduler picks first thread in highest priority queue
- Jobs start in highest priority queue
  - Assumes “good” behavior
- If a job used up the entire time slice, its priority drops one level
  - Otherwise it retains its priority

MLFQ

Priority	Time Slice (ms)	Round Robin Queues
1	10	
2	20	
3	40	
4	80	



MLFQ: Starvation and Gaming

- Wait … this design still allows starvation
  - Why?
- How to solve this issue?
  - After some period S, reset the priority by moving every job to the highest priority
- Can a job abuse the scheduler to increase its running time?
  - Jobs can deliberately relinquish the CPU before slice expires
  - Solution: using allotment time instead of one slice

Unix Scheduler

- The canonical Unix scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels
    - Timesharing: first 60 priorities
    - System: next 40 priorities
    - Real-time: next 60 priorities
    - Interrupt: next 10 (Solaris)
- Priority scheduling across queues, RR within a queue
  - The process with the highest priority always runs
  - Processes with the same priority are scheduled RR
- Processes dynamically change priority
  - Increases over time if process blocks before end of quantum
  - Decreases over time if process uses entire quantum

Motivation of Unix Scheduler

- The idea behind the Unix scheduler is to reward interactive processes over CPU hogs
- Interactive processes (shell, editor, etc.) typically run using short CPU bursts
  - They do not finish quantum before waiting for more input
- Want to minimize response time
  - Time from keystroke (putting process on ready queue) to executing keystroke handler (process running)
  - Don't want editor to wait until CPU hog finishes quantum
- This policy delays execution of CPU-bound jobs
  - But that's ok

Other Scheduling Ideas*

- Lottery scheduler: Give processes tickets proportional to their priority
  - Linux cgroup
- Stride Scheduler (also known as proportional share): Like lottery but more predictable
  - Basis for Linux Completely Fair Scheduler

- Scheduling for heterogeneous systems
- Scheduling for distributed systems/cloud
- …

Multiprocessor Scheduling*

- This is its own topic, we wont go into it in detail
  - Could come back to it towards the end of the quarter
- What would happen if we used MLFQ on a multiprocessor?
  - Contention for scheduler spinlock
  - Multiple MFQ used – this optimization technique is called distributed locking and is common in concurrent programming
- A couple of other considerations
  - Co-scheduling for parallel programs
  - Core affinity

Scheduling Summary

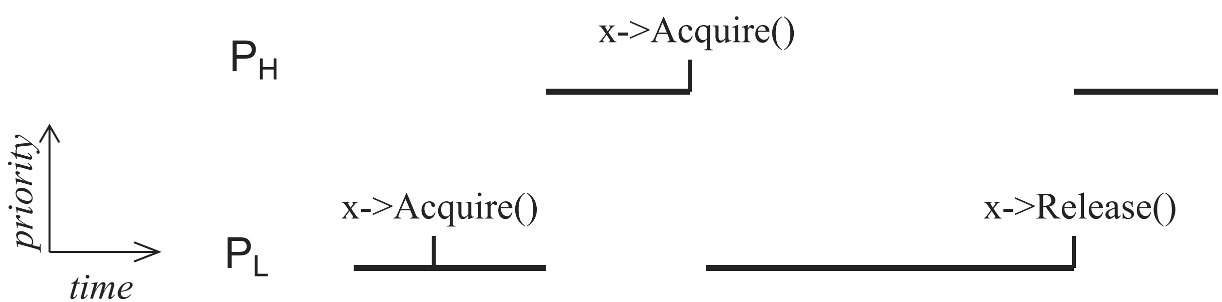
- Scheduler (dispatcher) is the module that gets invoked when a context switch needs to happen
- Scheduling algorithm determines which job runs next, where jobs are placed on queues
- Many potential goals of scheduling algorithms
  - Utilization, throughput, wait time, response time, fairness, etc.
- Various algorithms to meet these goals
  - FCFS/FIFO, SJF, RR, Priority
- Can combine algorithms
  - Multiple-level feedback queues
  - Unix example

Priority Scheduling

- Priority Scheduling
  - Choose next job based on priority
    - » Airline checkin for first class passengers
  - Can implement SJF, priority = 1/(expected CPU burst)
  - Also can be either preemptive or non-preemptive
- Problem?
  - Starvation – low priority jobs can wait indefinitely
- Solution
  - "Age" processes
    - » Increase priority as a function of waiting time
    - » Decrease priority as a function of CPU consumption

More on Priority Scheduling

- For real-time (predictable) systems, priority is often used to isolate a process from those with lower priority. *Priority inversion* is a risk unless all resources are jointly scheduled.



Priority Inheritance

- If lower priority process is being waited on by a higher priority process it inherits its priority
  - How does this help?
  - Does it prevent the previous problem?

- Priority inversion is a big problem for real-time systems
  - Mars pathfinder bug (link)