9/27/10

CPU Scheduling

Task: Which thread gets the CPU task - & when do they get it

Policy: Rules

Mechanism: ~~Implementation~~: How we implement the rules

Metaphor

Policy: Traffic laws
Mechanism: Police

5 Possible Goals for Schedulers

Fairness: Every job gets the same amount of CPU time

Efficiency: Keep as many ~~as~~ of the computer's resources as busy as possible

Response Time: Minimize / Maximize interactive response time

Turnaround Time: Give CPU preference (or not) to background tasks

Throughput: Complete as many jobs as possible in some amount of time

Issue: Do jobs run to completion, or not.

non-preemptive

preemptive

⇓

How long is a time slice?

too long: interactive use will suffer

too short: too many unnecessary context switches

Scheduling Policies

First-Come First-Served

- Similar to "standing in line"
- Non-preemptive
- The job that has waited the longest gets the CPU next

\+ Fair (No starvation)

\+ Simple

− Short jobs can get "stuck" behind long jobs

Round Robin

Preemptive FCFS

Jobs run until their time slice has expired, ~~or~~ they are blocked on a "slow operation", or they voluntarily give up the CPU.

+ Fair
+ Short jobs don't get stuck behind long jobs

- Unnecessary context switches when all jobs are about the same length (not short)

Shortest Job First

Maximizes throughput

Run the job requiring the least ~~complet~~ CPU time to complete

Non-preemptive.

- Not Fair (Starvation)

- Not implementable

Shortest Remaining Time to Complete First

- Preemptive

The job ~~requiring~~ "we" predict will use the smallest time slice percentage - based on its ~~past~~ recent history

- Can still have starvation

- The scheduler must track & compute the time slice usage

Priority-Based Scheduling

Priority: Some jobs are favored over other jobs

Policy: "Higher" priority jobs run before lower priority jobs

Issue: Preemptive or not?
New Issue: Do priorities change over time?

Static priorities: Priorites are fixed
- − Starvation
- + Easier on O.S.

Dynamic priorities: Priorities can change
- up or down
  - wait time
  - cpu time

+ No starvation
− Extra overhead, during context switch, in recomputing priorities

Final Issue: How many priorities are needed?

Somewhere around 4 priorities are typical

Most effective Ready Queue organization, is to have a separate Ready Queue for each priority

- This allows for different scheduling ~~for~~ policies for each priority
- The time slice interval can be different for each priority queue

Deadlock

Occurs because of competition
for resources
- hardware
- files
- synchronization primitives

Ex 1

lock1->A
⋮
lock2->A
lock1-R

lock1-> Acquire();
lock2-> Acquire();
⋮
cv2->Wait(lock2);

lock1->Acquire
lock2->Acquire
cv2->Signal(lock2

Ex 2

lock1->Acquire()
→ ⋮
lock2->Acquire()

lock 2-> Acquire()

lock1-> Acquire()

Define Deadlock

Two, or more, jobs each waiting on an event that can only be produced by one of the waiting jobs.

Using Resource Sequence

① User program requests resource access from the O.S.

② Once access is granted, the user's job uses the resource

③ When user program is done with the resource, it is given back to the O.S.

What if a requested resource is not available?

① Fail the request

Application has the responsibility to handle failed requests

\+ No deadlock possible

② Queue the request

O.S. has responsibility to manage queued requests

\- Deadlock is possible

4 Conditions for Deadlock

1. Mutual Exclusion

   Some resources that cannot be shared

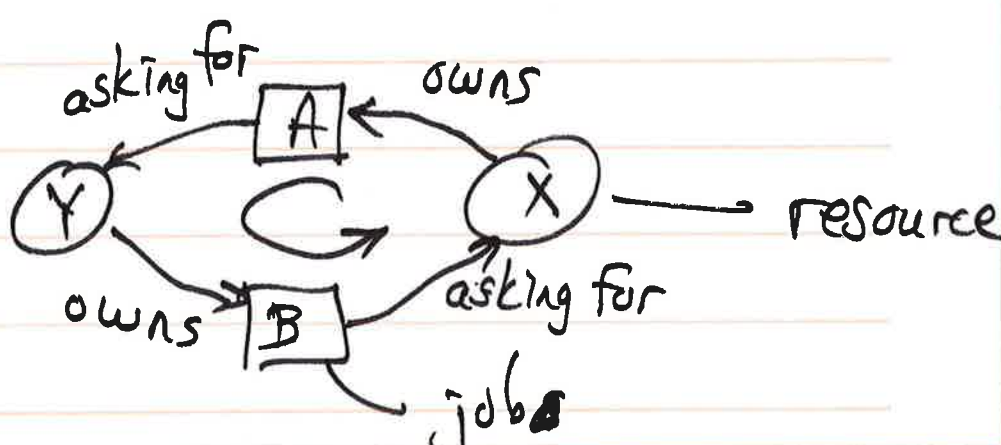
2. Hold & Wait

   Jobs can own resources & request other resources, without giving up what they already own.

3. No preemption

   Once a resource is given to a job, it cannot be "safely" borrowed

4. Circular Wait



Deadlock Solution Strategies ($3\frac{1}{2}$)

1. Do nothing

2. Detection & Recovery

   Detection: have a way to detect deadlock has occurred

   Recovery: kill a job

3. Dynamic Avoidance

   We try to keep deadlock from happening by carefully allocating resources.

4. Prevention

   Eliminate 1 of 4 required conditions for deadlock