Scheduling and Synchronization

Questions (15 min):

Scheduling(15-20 min):

This is a post-mortem rip off of Levy's notes from last quarter. His outline is much better than mine was.

Scheduling

- The scheduler is the module that moves jobs from queue to queue
  - the scheduling algorithm determines which job(s) are chosen to run next, and which queues they should wait on
  - the scheduler is typically run when:
    - a job switches from running to waiting
    - when an interrupt occurs
      - especially a timer interrupt
    - when a job is created or terminated
- There are two major classes of scheduling systems
  - in preemptive systems, the scheduler can interrupt a job and force a context switch
  - in non-preemptive systems, the scheduler waits for the running job to explicitly (voluntarily) block

Scheduling Goals

- Scheduling algorithms can have many different goals (which sometimes conflict)
  - maximize CPU utilization
  - maximize job throughput (#jobs/s)
  - minimize job turnaround time ($T_{finish} - T_{start}$)
  - minimize job waiting time (Avg(T_{wait}): average time spent on wait queue)
  - minimize response time (Avg(T_{resp}): average time spent on ready queue)
- Goals may depend on type of system
  - batch system: strive to maximize job throughput and minimize turnaround time
  - interactive systems: minimize response time of interactive jobs (such as editors or web browsers)

Scheduler Non-goals

- Schedulers typically try to prevent starvation
  - starvation occurs when a process is prevented from making progress, because another process has a resource it needs
- A poor scheduling policy can cause starvation
  - e.g., if a high-priority process always prevents a low-priority process from running on the CPU
- Synchronization can also cause starvation
  - we'll see this in a future class
  - roughly, if somebody else always gets a lock I need, I can't make progress

Algorithm #1: FCFS/FIFO

- First-come first-served (FCFS)
  - jobs are scheduled in the order that they arrive
  - "real-world" scheduling of people in lines
    - e.g. supermarket, bank tellers, MacDonalds, …

- typically non-preemptive
    - no context switching at supermarket!
- jobs treated equally, no starvation
    - except possibly for infinitely long jobs

- Problems:
  - average response time and turnaround time can be large
    - e.g., small jobs waiting behind long ones
    - results in high turnaround time
  - may lead to poor overlap of I/O and CPU

Algorithm #2: SJF

- Shortest job first (SJF)
  - choose the job with the smallest expected CPU burst
  - can prove that this has optimal min. average waiting time
- Can be preemptive or non-preemptive
  - preemptive is called shortest remaining time first (SRTF)
- Problem: impossible to know size of future CPU burst
  - from your theory class, equivalent to the halting problem
  - can you make a reasonable guess?
    - yes, for instance looking at past as predictor of future
    - but, might lead to starvation in some cases!

Algorithm #3: Priority Scheduling

- Assign priorities to jobs
  - choose job with highest priority to run next
    - if tie, use another scheduling algorithm to break (e.g. FCFS)
  - to implement SJF, priority = expected length of CPU burst
- Abstractly modeled as multiple "priority queues"
  - put ready job on queue associated with its priority
- The problem: starvation
  - if there is an endless supply of high priority jobs, no low-priority job will ever run
- Solution: "age" processes over time
  - increase priority as a function of wait time
  - decrease priority as a function of CPU time
  - many ugly heuristics have been explored in this space

Algorithm #4: Round Robin

- Round Robin scheduling (RR)
  - ready queue is treated as a circular FIFO queue
  - each job is given a time slice, called a quantum
    - job executes for duration of quantum, or until it blocks
    - time-division multiplexing (time-slicing)
  - great for timesharing
    - no starvation
    - can be preemptive or non-preemptive
- Problems:
  - what do you set the quantum to be?
    - no setting is "correct"

    - if small, then context switch often, incurring high overhead
    - if large, then response time drops
  - treats all jobs equally
    - if I run 100 copies of SETI@home, it degrades your service
    - how can I fix this?

Synchronization example (15-20 min):

Problem 7.8 from Silberschatz, Galvin, and Gagne:

We have one counted semaphore, `barber_shop`, with its count initialized to the number of chairs in the barber shop. There are 3 binary semaphores, `mutex`, `barber_snooze`, and `hair_done`. `mutex` is initially unlocked, and the other two are initially locked. The shared variables are the boolean `barber_asleep`, which is initially false, and the integer `chairs_full`, which is initially 0.

```
barber(){
   while(1){
      P(mutex)
      if(chairs_full==0){
         //sleep
         barber_asleep=true
         V(mutex)
         P(barber_snooze)
         P(mutex)
         barber_asleep=false
      } else {
         // cut someones hair
         V(hair_done)
      }
   V(mutex)
   }
}

client(){
   // enter barber shop
   P(barber_shop)
   P(mutex)
   if(barber_asleep){
      V(barber_snooze)
   }
   //sit in chair
   chairs_full++
   V(mutex)
   P(hair_done)
   P(mutex)
   //get up and leave
```

```
    chairs_full--
    V(mutex)
    V(barber_shop)
}
```