COSC 301: Mon-Weds, 17-19 Sept 2012

Learning goals

- Describe and provide examples of scheduling algorithms such as FIFO, SJF, multilevel feedback queue scheduling, lottery scheduling, and the Linux scheduler, and compare their relative advantages and disadvantages.
- Distinguish between I/O-bound and CPU-bound processes; provide examples of each.

CPU Virtualization

Mechanisms Limited direct execution. At boot, set up trap/interrupt handlers, then run jobs. Three situations in which OS gets involved: #. System calls (context switches; change in privilege) #. Timer interrupts (hardware-assisted context switch) #. Process finishes

Policies Virtualization based on the notion of *time sharing*

Preemptible vs Non-preemptible Resources

Non-preemptible Once given to a process, resource cannot be taken away until voluntarily relinquished. Example: disk storage (disk block) OS must be a *manager* and decide which process gets which resource

Preemptible resources Resource can be taken away (ownership can be preempted). Typically few of these resources Example: CPU OS must perform *scheduling*

Main OS subsystems

Dispatcher Handles the dispatch/context switch mechanism

Scheduler Policy to determine which process gets CPU, and when

Allocator Policy to determine which processes compete for which CPU Needed for multiprocessor, parallel, and distributed systems

Initial assumptions

1. Set of jobs all arrive (essentially) at once
2. Each just uses CPU (no I/O, yet)
3. Each runs for the same length of time
4. Amount of required CPU time for a process is known *a priori*
5. We only care about one metric: **turnaround time**

Turnaround time Tcompletion - Tarrival

Algorithm 1: First-come first-served (FCFS) (or FIFO)

- Simple policy: jobs are scheduled in order of arrival. They run to completion once started.
  - FCFS is a *non-preemptive* policy
- Example: three jobs (A, B, C) requiring 10 sec CPU; what is average turnaround time? (goal is to minimize average turnaround time...)
  - Should be 20 sec.
- Relax assumption #3. What if Job B requires 100 sec CPU time?
  - Average turnaround time now goes up to 80 sec
  - This happens in real life! The *convoy effect*
- How to improve on FCFS?

Algorithm 2: Shortest job first (SJF)

- Another fairly simple policy, also *non-preemptive*
- What is average turnaround time for convoy example w/FCFS?
  - Should be 50 sec - much better
  - SJF is *optimal* given the initial assumptions. Optimal, as in it's impossible to do better.
- What if we relax assumption #1? A: 10 sec and B: 100 sec arrive at time 0; Job C for 10 sec arrives at time 20.
  - Bad for SJF, since it's a non-preemptive policy
- What's another problem with SJF? (Totally unrealistic! Most importantly, it's impossible to know CPU requirements in advance.)

Algorithm 3: STCF (Shortest time to completion first; SJF with preemption)

- Improve on SJF by making it preemptive
- This is much better!
- What if we now relax assumption #5 and use additional metrics to evaluate scheduling policy performance?
  - Response time: Time when job is first run on the CPU minus the time of arrival. Measures how long it takes to get a "response" from the CPU.
  - Why might we care about response time?
  - Do any of the above policies provide good response time?

Algorithm 4: Round-robin

- Need the notion of a time slice (use our lovely hardware timer interrupt)
  - Timer interrupt every x milliseconds; time slice for a process is n * x, where n >= 1.k
- Idea is to run each process for a time slice on the CPU, giving each process a chance to run on the CPU for a short amount of time
  - Geared specifically toward improving response time!
  - What happens if we try to minimize response time with RR by reducing time slice?
    - Tradeoff with context switching time
    - With (much) longer time slices, can degenerate to FCFS
- RR is great for response time, but is one of the worst policies for turnaround time!
  - RR is, arguably, *fair*
  - It's a tradeoff; to improve turnaround time, you must introduce some unfairness

Incorporating I/O

- Still have assumption of processes only requiring CPU - totally unrealistic
- Typical mix
    1. Interactive jobs with short CPU bursts then long pauses (blocked on I/O)
    2. Batch jobs: long-running CPU-intensive processes

  - Draw process alternating between CPU and I/O; want to overlap I/O with CPU among competing processes
    - I/O-bound versus CPU-bound processes
    - Typical approach is to treat each CPU burst as a separate job from the scheduling standpoint

- Still also have assumption that we have prior knowledge of how long processes need to run on CPU
  - How to get around this assumption?
    - Incorporate history!

Algorithm 5: Multilevel Feedback Queues

- Goal: provide as good a balance as possible between minimizing turnaround time and response time
  - Want the system to "feel" responsive for interactive jobs
  - Want to provide good turnaround time for CPU-bound processes
- Basic idea: assume each new job entering the system is interactive. As the jobs runs, figure out whether that's true or not
  - Start with round-robin, but with many queues (we'll assume 3)
    - Q2 -> A,B (high priority)
    - Q1 -> C
    - Q0 -> D,E (low priority

- Each job has a priority at a given time, may change

Base rules

1. If P(A) > P(B), then A runs
2. If P(A) == P(B), then do round robin (A,B)

Key question: how to change priorities over time?

3. Start jobs at highest priority
    a. If I/O issued before time slice used up, stay at same priority
    b. If time slice used up, move down one level in priority

Examples

1. One long-running job --- what happens?
2. A long-running job arrives first, then a short job arrives some time later (and survives for 2 time slices)
3. Two long-running jobs: one I/O intensive, one CPU intensive

Problems

1. Starvation
   - If too many interactive (high prio) jobs, low prio jobs may never get scheduled
   - After some time period S, move all jobs back to highest priority
     - Forcibly avoid starving processes
     - Process behavior may change over time, anyway!
2. Gaming the scheduler
   - If you're a CPU-bound job and know the time slice length, just issue a dummy I/O operation before the end of the time slice, and retain high priority.
   - Fix: Better accounting. Keep track of how much CPU time a process has used at a given level. Once used up, reduce process priority.

Parameterizing MLFQ

- How long should S be (priority boost)?
- How long should time slices be? Should they be the same at each priority?
  - How much CPU time at each priority level?
- Solaris: 60 queues, highest priority around 20 millisec time slice, down to multiple hundreds of milliseconds; priority boost every 1 second.

Algorithm 6: Lottery Scheduling

- Cool scheduling algorithm; not going to spend much time on this
- Basic idea: proportional share of the CPU; try to allow each process to gain some percentage of CPU time
  1. Give processes n lottery tickets for resources
  2. When a process needs to be scheduled, choose a lottery ticket at random; process holding that ticket is selected
  3. No starvation as long as a job has at least one ticket
- Benefits
  - Responsive
  - Cooperative processes may trade tickets
    - Example: client can loan tickets to a server so the server can quickly process the client's request
  - Can achieve proportional fairness through ticket allocation

Algorithm 7: The Linux Scheduler

- Linux kernel schedules *tasks*
  - Task abstraction can act as process (no shared address space with another task)
  - ... Or as a thread (shared addr space with another task)
- Tasks can be specified as *real-time* or *conventional*
  - Real-time tasks have static priority 0-99
  - Conventional tasks have priority 100-139
- Basic algorithm
  - Find highest priority queue with a ready process
  - Compute its time slice (quantum) length

  - Run it
- Separate ready queues for each processor
  - Ready queue is actually 140 separate queues
    * Two arrays[140] of linked lists of tasks: active and expired
  - Easy to select next task of highest priority by scanning active array
    * O(1)
  - As tasks use their time slices, they're added to a list on the expired array
  - When there are no more tasks in the active array, the arrays are swapped
  - Algorithms to try to keep ready queues balanced
- Time slice
  - Quantum = (140 - staticprio)*20 if SP < 120; (140-staticprio)*5 otherwise
    * Static priority is normally 120, but can be set via system calls (100-139)
    * Higher priority processes get *longer* time quanta! (Idea is that important processes should run longer)
    * Processes also get a dynamic priority that is used in computing time slice (static priority + a *penalty*)
    * I/O-bound jobs have negative penality so their dynamic priority rises
    * CPU-bound jobs have positive penality so their dynamic priority decreases
- What problem does the use two arrays solve (specifically with MLFQ)?

Multiprocessor System Scheduling

- OSes typically maintain ready queues for *each* processor
- In a multiprocessor context, additional scheduling issues arise
  - What if one processor is underutilized?
  - Want to balance load across processes, but...
- Processor cache performance
  - May want to avoid "migrating" a process/thread from one processor to another to improve cache behavior
  - Can programmatically "pin" a thread to a processor (with special system calls)
  - OS may support "soft" affinity (a mere preference for a processor) or "hard" affinity (demand a specific processor!)

- Load balancing across processors
  - Job of the allocator (or an additional scheduling agent, what ever it happens to be called in a given OS)
    * Balancing may require migrating a process from one CPU to another, which may affect cache performance
    * Generally want to avoid movement if possible