Advanced Operating Systems (CS 202)

Scheduling (1)
Today: CPU Scheduling
The Process

- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling
  - It is the dynamic execution context of a program
  - A process is sometimes called a job or a task

- A process is a program in execution
  - Programs are static entities with the potential for execution
  - Process is the animated/active program
    - Starts from the program, but also includes dynamic state
Process State Graph

New -> Ready
Create Process

Ready -> Waiting
I/O Done

Waiting -> Running
Schedule Process

Running -> Ready
Unschedule Process

Running -> Terminated
Process Exit

Terminated -> New
I/O, Page Fault, etc.

UCR
Threads

- Separate dual roles of a process
  - Resource allocation unit and execution unit
  - A thread defines a sequential execution stream within a process (PC, SP, registers)
  - A process defines the address space, and resources (everything but threads of execution)

- A thread is bound to a single process
  - Processes, however, can have multiple threads

- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities
Threads in a Process

Stack (T1)

Stack (T2)

Stack (T3)

Heap

Static Data

Code

PC (T1)

PC (T2)

PC (T3)

Thread 1

Thread 2

Thread 3
Thread Design Space

- **One Thread/Process, One Address Space** (MSDOS)
- **Many Threads/Process, One Address Space** (Pilot, Java)
- **One Thread/Process, Many Address Spaces** (Early Unix)
- **Many Threads/Process, Many Address Spaces** (Mac OS, Unix, Windows)
Today: CPU Scheduling

- Scheduler runs when we context switching among processes/threads on the ready queue
  - What should it do? Does it matter?

- Making the decision on what thread to run is called scheduling
  - What are the goals of scheduling?
  - What are common scheduling algorithms?
  - Lottery scheduling

- Scheduling activations
  - User level vs. Kernel level scheduling of threads
Scheduling

- Right from the start of multiprogramming, scheduling was identified as a big issue
  - CCTS and Multics developed much of the classical algorithms

- Scheduling is a form of resource allocation
  - CPU is the resource
  - Resource allocation needed for other resources too; sometimes similar algorithms apply

- Requires mechanisms and policy
  - Mechanisms: Context switching, Timers, process queues, process state information, …
  - Scheduling looks at the policies: i.e., when to switch and which process/thread to run next
Preemptive vs. Non-preemptive scheduling

- In *preemptive* systems where we can interrupt a running job (involuntary context switch)
  - We’re interested in such schedulers…

- In *non-preemptive* systems, the scheduler waits for a running job to give up CPU (voluntary context switch)
  - Was interesting in the days of batch multiprogramming
  - Some systems continue to use cooperative scheduling

- Example algorithms:
  - RR, FCFS, Shortest Job First (how to determine shortest), Priority Scheduling
Scheduling Goals

- What are some reasonable goals for a scheduler?
- Scheduling algorithms can have many different goals:
  - CPU utilization
  - Job throughput (# jobs/unit time)
  - Response time ($\text{Avg}(T_{\text{ready}})$: avg time spent on ready queue)
  - Fairness (or weighted fairness)
  - Other?

- Non-interactive applications:
  - Strive for job throughput, turnaround time (supercomputers)
- Interactive systems
  - Strive to minimize response time for interactive jobs
- Mix?
Goals II: Avoid Resource allocation pathologies

- **Starvation** no progress due to no access to resources
  - E.g., a high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock

- **Priority inversion**
  - A low priority process running before a high priority one
  - Could be a real problem, especially in real time systems
    - Mars pathfinder: http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html

- **Other**
  - Deadlock, livelock, …
Non-preemptive approaches

- Introduced just to have a baseline
- FIFO: schedule the processes in order of arrival
  - Comments?

- Shortest Job first
  - Comments?
Preemptive scheduling: 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?
Priority Scheduling

- Choose next job based on priority
  - Airline check-in 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
Combining Algorithms

- Scheduling algorithms can be combined
  - Have multiple queues
  - Use a different algorithm for each queue
  - Move processes 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
Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)

- Not perfect at any of them!
  - Used in Unix (and Windows and MacOS)
<table>
<thead>
<tr>
<th>Priority</th>
<th>Time Slice (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
</tr>
</tbody>
</table>

Round Robin Queues

- new or I/O bound task
- time slice expiration
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
Linux scheduler

» Went through several iterations

» Currently CFS
  » Fair scheduler, like stride scheduling
  » Supersedes O(1) scheduler: emphasis on constant time scheduling regardless of overhead
  » CFS is O(log(N)) because of red-black tree
  » Is it really fair?

» What to do with multi-core scheduling?