Chapter 11

CPU Scheduling Policies

CPU Scheduling Policies

Read:

– Silberschatz: 6

– Tanenbaum: 2.5

CPU Scheduling

- Deciding **which** process to run
- (Deciding **which** thread to run)
- Deciding **how long** the chosen process can run
- Important for system **throughput**
- Important for system **responsiveness**

CPU Scheduling Opportunities

CPU scheduling can occur after each of the following process state transitions:

- (enter) $\rightarrow$ ready: new process is created
- running $\rightarrow$ ready: e.g., timer interrupt
- running $\rightarrow$ waiting: e.g., I/O request or wait interrupt
- running $\rightarrow$ (exit): process is terminated
- waiting $\rightarrow$ ready: e.g., I/O or process completion

Kinds of CPU Scheduling

- **Simple:** Scheduling is only done after termination
  - Process runs until its program is completed
- **Cooperative** or **nonpreemptive**: Scheduling is only done after termination and after entering the waiting state
- **Preemptive:** Interrupts can be sent to processes for implementing scheduling decisions
  - A running process can be preempted by a process that arrives at the ready queue
  - Implemented by periodic clock interrupts

Scheduling Performance Measures

- **CPU utilization:** portion of time CPU is busy
- **Throughput:** number of processes completed per time unit
- **Turnaround time:** average time a process takes to complete
- **Waiting time:** average total time a process spends waiting in the ready queue
- **Response time:** average time a process takes to respond

Round Robin (RR)

Standard time-sharing algorithm

- The ready queue is an ordered circle
- The next process in the ready queue is allocated to the CPU for a fixed time period T
  - preempted if CPU burst is $> T$
- ***Advantages:*** Scheduling is **fair**
- ***Disadvantages:***
  - Waiting time may be long
  - Performance is very sensitive to the size of T

First Come, First Served (FCFS)

Simple **non-preemptive** job scheduling policy

- The ready queue is implemented as a queue and its ordering is used to schedule the processes in the queue
- Available for **real-time thread scheduling**
- Advantage: Easy to implement
- Disadvantages:
  - Waiting time is not minimal
  - No attempt to balance I/O and CPU usages

Shortest Job First (SJF)

"Prophetic" waiting-time minimization

- Process with **shortest next CPU burst** comes first — preemptive or non-preemptive
- Advantages:
  - Given the lengths of the next CPU bursts, easy to implement
  - Waiting time is **provably** minimal
- Disadvantages:
  - Difficult to predict the length of a CPU burst
  - CPU-intensive processes may have to wait

Priority

- Processes are assigned priorities; the process with the highest priority is dispatched first
- Priority assignment criteria:
  - **internal** (computational requirements, e.g. SJF), or
  - **external** (application requirements)
- ***Advantages:*** Easy to implement and flexible
- ***Disadvantages:***
  - Assigning priorities may be difficult
  - Low-priority processes may starve

Multiple Queues

- There is one queue for each process category (such as the category of system processes)
  - Each queue has is own scheduling algorithm
- The queues are scheduled as units:
  - Each queue may be assigned a priority
  - Each queue is given a portion of CPU time
- Processes can be allowed to move between queues
- ***Advantage:*** Flexibility
- ***Disadvantage:*** Hard to implement

Multiprocessor Scheduling

- ***More complex*** than uniprocessor scheduling
  - Processes must be scheduled effectively
  - Processors must be kept busy
- **Symmetric multiprocessing**: Homogeneous processors schedule themselves
  - Data sharing safety is a major issue: Processes running simultaneously on different processors may access and modify common data structures
- **Asymmetric multiprocessing**: One processor handles all system activities including processor scheduling and I/O processing

Real-Time Scheduling

- **Reduce dispatch latency:** *preemptible system calls*
  - *preemption points* in long-running system calls
  - *preemptible kernel* protecting all OS data structures with synchronization mechanisms
- **Priority inversion** resolved via priority inheritance
- **Hard Real-Time Scheduling:** needs to complete **critical** tasks within **guaranteed** time
  - resource reservation
  - predictability of duration of actions needed
  - *usually:* no virtual memory, no secondary storage
  - special-purpose software running on **dedicated** hardware
- **Soft Real-Time Scheduling:** High priority e.g. for multimedia applications or interactive graphics

Algorithm Evaluation Methods

- **Analysis of mathematical models**
  - Deterministic modelling using fixed workloads
  - Queuing theory (e.g.: Little's formula: $n = \lambda \cdot W$ with av. queue length n, arrival rate λ, and av. waiting time W)
- **Simulation**
  - Hard to get realistic workloads
- **Implementation testing**
  - Extremely expensive for experiments
  - Results may be platform dependent

CPU Scheduling Policy Examples

Section 6.7 of [Silberschatz-Galvin-Gagne-2002]:

- ***Solaris 2***
  - **Hard** real-time
    - dispatch latency with preemption enabled: 2ms
- ***Windows 2000***
  - Priority boost for "foreground" process
- ***Linux***
  - Credit-based priority system
  - Preemptible kernel still experimental

CPU Scheduling Policy Example: Solaris 2

Four scheduling classes: Real time, system, time sharing, interactive

- Priority based. Priorities within classes are converted into global priorities used for scheduling
- **Real-time class** has highest priority
  - guaranteed response within a bounded period of time
  - few real-time processes
- **System class** for kernel processes
  - ***Examples:*** scheduler, paging daemon, …
  - No time slicing
  - $\Rightarrow$ system processes run until blocked or preempted by higher-priority processes

Solaris 2 Time Sharing

Same scheduling policies for **time sharing** and **interactive**:

- Inverse relation between priorities and time slices:

  The higher the priority, the shorter the time slice

- Interactive processes, in particular windowing applications, typically have higher priority than CPU-bound processes

- $\Rightarrow$ Good interactive response time, good CPU throughput