

CPU Scheduling

- General rule — keep the CPU busy; an idle CPU is a wasted CPU
- Major source of CPU idleness: I/O (or waiting for it)
- Many programs have a characteristic *CPU–I/O burst cycle* — alternating phases of CPU activity and I/O inactivity
 - ◆ *CPU-bound* programs have fewer, longer CPU bursts
 - ◆ *I/O-bound* programs have more, shorter CPU bursts

Core Definitions

- *CPU scheduling* (a.k.a. *short-term scheduling*) is the act of selecting the next process for the CPU to “service” once the current process leaves the CPU idle
- Many algorithms for making this selection
- Implementation-wise, CPU scheduling manipulates the operating system’s various PCB queues
- The *dispatcher* is the software that performs the dirty work of passing the CPU to the next selected process — the time to do this is called the *dispatch latency*

Preemptive vs. Cooperative Scheduling

- Two levels at which a program can relinquish the CPU
- First level = *cooperative scheduling*
 - ◊ Process explicitly goes from running to waiting (e.g., system call for I/O or awaiting child termination)
 - ◊ Process terminates
- Cooperative scheduling (or “cooperative multitasking”) was used in older personal computer operating systems (< Windows 95, < Mac OS X), primarily due to what PC hardware could do

- Second level = *preemptive scheduling*
 - ◊ Interrupt causes a process to go from running to ready state
 - ◊ Interrupt causes a process to go from waiting state to ready state (e.g., asynchronous completion of I/O)
- Key word is “interrupt” — this is a hardware-level feature that is required for preemptive scheduling
- Prevents a process from “running away” with the CPU
- But brings up new issues of its own:
 - ◊ Coordinating access to shared data
 - ◊ Can you interrupt an interrupt?

Scheduling Criteria

There is no “one, true scheduling algorithm,” because the “goodness” of an algorithm can be measured by many, sometimes contradictory, criteria:

- *CPU utilization* — Give the CPU steady work
 - *Throughput* — “Work” per unit time; one definition of work is a completed process
 - *Turnaround time* — Real time to process completion
 - *Waiting time* — Time spent in ready queue
 - *Response time* — Time from request to response
-
- Note how the purpose of a system determines the appropriateness of a criterion: throughput is applicable mainly when individual program runs correspond to completions of a task, while response time is crucial to interactive systems (e.g., graphical user interfaces)
 - Typical goals: maximize CPU utilization and throughput but minimize times
 - Other choices for optimization:
 - ◆ Optimize the average measure, or min/max (i.e., how bad can the “worse-case scenario” be?)
 - ◆ Minimize *variance* in measure, thus resulting in better *predictability* — meaningful for interactive systems, but not much work done in this area

First-Come, First-Served (FCFS) Scheduling

- Simple premise: the sooner a process asks for the CPU, the sooner it gets it; subsequent processes wait in line until the ones before them finish up (or fall into an I/O wait state)
- This is a *cooperative* algorithm — the CPU can't be taken away from a process
- Simple implementation: FIFO queue
- Analysis: order significantly affects the average waiting time; certain combinations of CPU- and I/O-bound processes decrease utilization

Shortest-Job-First (SJF) Scheduling

- Requires knowledge of CPU burst durations: give the CPU to the process with the shortest next CPU burst; provably minimizes the average waiting time
- *But* how the heck do we know the length of the next CPU burst? Better fit for batch systems (long-term scheduling), where users can assign time limits to jobs
- CPU scheduling can *approximate* SJF by *predicting* the next burst length — generally via *exponential average*
- In preemptive flavor, we change processes if the next CPU burst $<$ what's left of the current one

Priority Scheduling

- Assign a priority p to a process (typically lower p = higher priority, but this isn't set in stone), and give the CPU to the process with the highest priority
- Note how SJF is a special case of priority scheduling: p is the inverse of the next predicted CPU burst
- Processes of equal priority are scheduled FCFS
- Priorities range from internally-calculated metrics (time limits, memory requirements, open files, I/O-to-CPU burst ratio) to external factors

- Comes in both preemptive and cooperative flavors:
 - ◆ Preemptive version interrupts the currently running process when a higher-priority process comes in
 - ◆ Cooperative version puts the higher-priority process at the top of the queue and waits for the currently running process to relinquish the CPU
- Key issue: *indefinite blocking* or *starvation* of a process — low-priority processes may wait forever if higher-priority ones keep showing up
- Address starvation through *aging*: gradually increase a process's priority as waiting time increases — caps the maximum waiting time

Round-Robin (RR) Scheduling

- Necessarily preemptive: defines a *time quantum* or *time slice*; processes never have the CPU for > 1 quantum
- Implementation: maintain the ready queue as a circular FIFO, then traverse each PCB
- Two possibilities:
 - ◇ Current process relinquishes in < 1 quantum
 - ◇ 1 quantum passes, resulting in a timer interrupt
- In both cases, the “done” process is moved to the tail of the queue, and the new head becomes current

- Average waiting time is longer with RR, in exchange for better response time with sufficiently small quantum
 - ◇ Note how RR is effectively FCFS if quantum is sufficiently large, unless CPU bursts are *really* long
- With RR, the cost of a *context switch* gains significance: we want quantum $>$ context-switch time, preferably by multiple orders of magnitude (i.e., milli- vs. μ -seconds)
 - ◇ But again, not too large! Or else we’re back at FCFS
 - ◇ Rule of thumb: choose time quantum so that 80% of CPU bursts are shorter than that quantum
- Interesting tidbit: turnaround time is not necessarily proportional (direct or inverse) to the time quantum

Multilevel Queue Scheduling

- Our first “composite” algorithm: partition processes into different queues, each with its own scheduling algorithm (as appropriate for that queue)
- Canonical example: interactive processes use RR in one queue, batch processes use FCFS in another
- Now of course we have to schedule *among* queues:
 - ◊ Priority scheduling — queues have preset priorities
 - ◊ RR scheduling — each queue is given some quantum during which its processes do work

Multilevel Feedback-Queue Scheduling

- Multilevel queues + the ability for a process to move to another queue
 - ◊ For example, track CPU burst times and move CPU-bound processes to a lower-priority queue; vice versa for I/O-bound processes
 - ◊ Use aging to prevent starvation: long wait times for a process move it to a higher-priority queue
- Lots of parameters to play with: number of queues, scheduling algorithms per queue, queue selection rules

Multiple-Processor Scheduling

- Potential for better overall performance
- Focus on *homogeneous* multiple processors: allows any available processor to run any available process
- Two approaches — most modern OSes do SMP:
 - ◆ *Asymmetric multiprocessing* has a single master processor (i.e., the one running OS code), relegating other processors to run user code only
 - ◆ *Symmetric multiprocessing (SMP)* allows each processor to make decisions for itself (“self-scheduling”)

- *Processor affinity* — Processes tend to perform better if they stay on the same CPU, particularly due to caching
 - ◆ *Soft affinity* tries to keep processes on the CPU, but doesn’t absolutely guarantee it
 - ◆ *Hard affinity* disallows process migration
- *Load balancing* — Ideally, all CPUs have about the same amount of work at any given time; counteracts processor affinity somewhat
 - ◆ Process migration may be *push* (overloaded CPUs dump on idle ones) or *pull* (idle CPUs yank someone else’s work)
 - ◆ Many systems do both (Linux, ULE in BSD)
- *Symmetric multithreading (SMT)* — Hardware ability to present a single physical CPU as multiple *logical* CPUs: the OS doesn’t really need to know, but it may help to be aware of which physical CPU has which logical CPU

Thread Scheduling

- *Contention scope* — determines a thread’s “competition” for the CPU: either with other threads within the same process (*process-contention scope*) or with other threads in the entire system (*system-contention scope*)
- Contention scope is typically bound to the operating system’s threading model (many-to-one and many-to-many use process-contention scope; one-to-one uses system-contention scope)

Scheduling Examples

Solaris	Priority-based with 4 classes (real time, system, interactive, time sharing); multilevel feedback queue for time sharing and interactive class; the lower the priority, the higher the time quantum
Windows XP	Priority-based with preemption; 32 priority levels split into <i>variable class</i> (1–15) and <i>real-time class</i> (16–31); priority 0 thread does memory management; special <i>idle</i> thread gets run if no ready threads are found
Linux	Priority-based with preemption; 141 priority levels split into <i>real-time</i> (0–99) and <i>nice</i> (100–140); the <i>higher</i> the priority, the higher the time quantum; for SMP, each CPU has its own <i>runqueue</i>
Mac OS X	Policy-based with preemption; priority is embedded in the scheduling policy — a <i>standard policy</i> uses a system-defined fair algorithm; a <i>time constraint policy</i> is for real-time needs; and a <i>precedence policy</i> allows externally-set priorities

Evaluating Scheduling Algorithms

- As you've seen, different algorithms have different strengths, and no single one is ideal for absolutely every situation
- So, we need techniques for quantifying the characteristics of each algorithm in order to make an informed choice
- First things first — we need to specify the quantitative metrics for a “good” algorithm for our particular need:
 - ◆ “Maximum CPU utilization with maximum response time of 1-second”
 - ◆ “Maximum throughput with turnaround time linearly proportional to execution time”

Evaluation Techniques

- *Deterministic modeling*: Real calculations on exact cases; simple and accurate, but requires exact information
- *Analytic evaluation*: Superclass of deterministic modeling — reasoned direct study of algorithm's properties
- *Queueing models*: Model system resources as servers with queueing properties, then solve *Little's formula*
- *Simulations*: Create a model of the system, then process statistical or real data (*trace tapes*)
- *Implementations*: “Just do it” — then measure