Processor Scheduling

*Chapters 9 and 10 of [OS4e]*,

*Chapter 6 of [OSC]:*

- Queues
- Scheduling Criteria
- Cooperative versus Preemptive Scheduling
- Scheduling Algorithms
- Multi-level Queues
- Multiprocessor and Real-Time Scheduling
- Java Thread Scheduling

Queues

Recall OS maintains various queues

- linked lists of process control blocks or thread control blocks
- not necessarily FIFO

**ready queue**

- short-term scheduler selects process/thread from this queue for dispatching

**job queue (batch queue)**

- long-term scheduler selects job from this queue for admission as process to system

**blocked, ready-suspend, blocked-suspend queues**
**Scheduling Criteria**

**User-oriented, Performance-related**
- provide acceptable *turnaround time* for batch jobs
- maximise number of interactive users receiving acceptable *response time*
- maximise percentage of *deadlines* met for *real-time* tasks

**User-oriented, Other**
- *predictability* of turnaround time or response time
System-oriented, Performance-related
- maximise throughput
  i.e., number of processes completed per unit time
- maximise processor utilisation

System-oriented, Other
- fairness
- enforce priorities
- balance resource utilisation

Cooperative versus Preemptive Scheduling
Short-term scheduler might be invoked after interrupt handled
- because running process terminated or blocked
- to give opportunity for preemption
  e.g., another process arrives at ready queue
  e.g., allocated time-slice expired
  – running process returns to ready queue
N.B. Some OS, e.g., Windows 3.x, do not allow preemption
Scheduling Algorithms

When processor becomes free, short-term scheduler uses an algorithm to select process from ready queue and dispatch it.

Scheduling algorithms may make decisions based upon:
- waiting time $w$ (= current time - arrival time)
- estimate of service time $s$ (= execution time)
  - e.g., by user (job aborts if over runs)
  - e.g., exponential averaging of previous actual values $t(k)$ for that job
    
    $s(k+1) = \alpha t(k) + (1 - \alpha) s(k)$, where $0 < \alpha < 1$

of each process in ready queue.

Effectiveness of algorithms can be calculated (after $n$ processes have completed) from their waiting times $w_i$ and actual service times $t_i$:
- mean (i.e., average) absolute waiting time, i.e., $(w_1 + \ldots + w_n) / n$
- mean normalised waiting time, i.e., $(w_1/t_1 + \ldots + w_n/t_n) / n$

Alternatively:
- effectiveness can be expressed in terms of turnaround time $T = w + t$
- mean normalised turnaround time = mean normalised waiting time + 1

First-Come, First-Served
- select process with maximum $w$
- nonpreemptive
- cheap to implement (FIFO queue)
- performance of short processes (i.e., with short service time) may be unacceptable
  - e.g., process with service time 100ms comes before one with service time 1ms
  - e.g., convoy effect when I/O-bound processes repeatedly have to wait in ready queue for CPU-bound process to finish executing
Round-Robin
- preemptive (at time quantum) version of FCFS
- needs hardware support, i.e., timer (time-slicing)
- fair, in that all processes treated equally
- good for short processes
- overhead from process switching versus size of quantum
N.B. RR degenerates to FCFS if quantum is made too large

Shortest-Process-Next (Shortest-Job-First)
- select process with minimum s
- nonpreemptive
- implemented using priority queue, where shortest service time has highest priority
- optimal, in that it minimises mean absolute waiting time
- long processes are penalised and may even starve
Shortest-Remaining-Time
- preemptive (at arrival of another process) version of SPN
- no additional interrupts
- elapsed service time of preempted process must be recorded (or deducted from estimated service time)
- shortest process never has to wait
- same disadvantages as SPN
  i.e., expensive to implement, penalises long processes

Highest-Response-Ratio-Next
- select process with maximum $\frac{w}{s}$
- nonpreemptive
- fair to all processes
  - automatic aging
- expensive to implement
Multi-level Queues

- ready queue partitioned into several separate queues
  - e.g., one queue for interactive processes and another for batch jobs
  - e.g., one queue for each priority
- each queue operates own scheduling algorithm
- queues themselves are scheduled
  - e.g., process arriving at higher-priority queue preempts process previously dispatched from lower-priority one
  - danger of starvation for processes in lowest-priority queues
  - e.g., time-slicing with different quanta for each queue
Multi-level Feedback-Queues

- upon arrival, process may be placed in lower queue than dictated by its priority, if it has already used too much CPU time
  - CPU-bound processes thus give way to I/O-bound and interactive processes
- process waiting too long is moved to higher-priority queue (*aging*)

Figure 9.4 Priority Queuing
Multiprocessor and Real-Time Scheduling

With multiple CPU’s scheduling is more complex

- **symmetric multiprocessing**
  - ready queue shared by all processors, each running own short-term scheduler
  - cooperation necessary to ensure system data structures updated consistently, process neither lost nor selected by different processors

- **asymmetric multiprocessing**
  - one processor runs scheduler and dispatches processes to others

**Hard real-time systems**

- critical tasks must be completed within deadlines
  - secondary storage (and therefore virtual memory) cannot be used
  - require special-purpose software running on dedicated processor
**Soft real-time systems**
- critical processes must be given priority
e.g., multimedia, high-speed interactive graphics
- low *dispatch latency*
  - insert *preemption points* in long-duration system calls
  at safe places in kernel (i.e., system data structures
  must be in consistent state), or
  - make entire kernel preemptible (e.g., Solaris), relying
    on high-priority process not to access inconsistent
    data structures

**priority inversion**
- high-priority process needs resource already in use
- low-priority process allowed to inherit its priority in order
  to finish using resource

---

**Java Thread Scheduling**

How does JVM schedule threads?
- preemptive
  - time-slicing is optional
- `Thread.yield()` *(cooperative multitasking)*
- prioritised
  - `Thread.MIN_PRIORITY`
  - `Thread.NORM_PRIORITY`
  - `Thread.MAX_PRIORITY`
  - may be changed dynamically with `setPriority(n)`