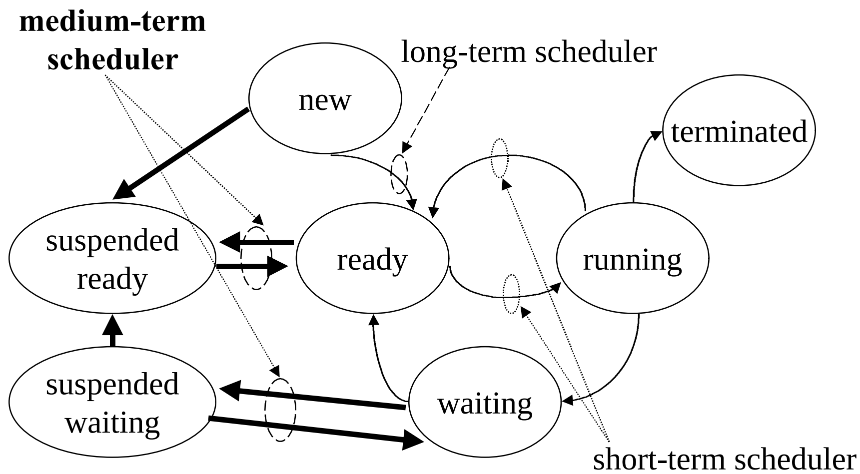
Silberschatz, et al.

Topics based on Chapter 6
CPU Scheduling

Topics covered

- Basic concepts/Scheduling criteria
- Non-preemptive and Preemptive scheduling
- Scheduling algorithms
- Algorithm evaluation

Process State Diagram



Short-term Scheduling

- Runs frequently--efficiency very important
- Critical to system’s performance--effectiveness
- Extensively studied--many interesting comparisons, theoretically-valid results
- Terminology:
  - **preemptive scheduling**: processes that are logically runnable can be temporarily suspended
  - **nonpreemptive scheduling**: processes permitted to run to completion or until they block

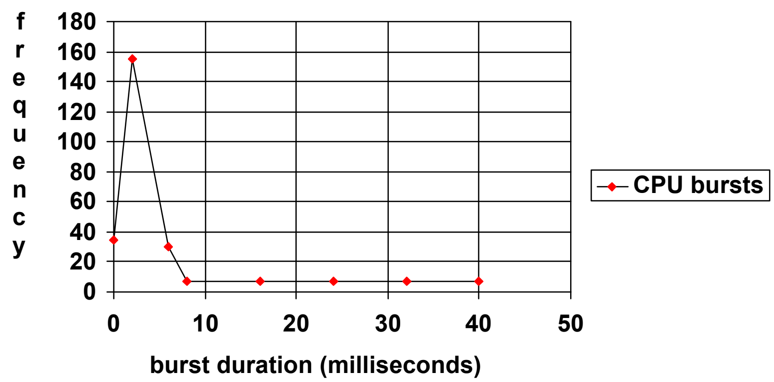
Short-term Scheduling Algorithms

- Nonpreemptive
  - First-Come First Serve (FCFS)
  - Shortest Job First (SJF)
- Preemptive
  - Shortest remaining time first (SRTF)
  - Round Robin Scheduling (RR)
  - Multilevel Queue Scheduling
  - Multilevel Feedback Queue Scheduling (MLF)

Why does Scheduling Work?

- Process behavior: CPU--I/O Burst Cycle
  - processes alternate between CPU execution and I/O waits
  - Lengths of CPU bursts exhibit predictable distribution
  - Large number of short CPU bursts
  - Small number of long CPU bursts
  - I/O bound--many very short CPU bursts
  - CPU bound--few very long CPU bursts

CPU-burst times histogram



CPU Scheduler

- Job: select from among the processes in memory that are ready to execute, and allocate the CPU to one of them
- CPU scheduling decisions can take place when a process
  - Switches from running to waiting state (nonpreemptive)
  - Switches from running to ready (preemptive)
  - Switches from waiting to ready (preemptive)
  - Terminates (nonpreemptive)

Dispatcher

- Dispatcher gives control of CPU to the selected process. This involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- Dispatch latency--time it takes for the dispatcher to stop one process and start another running.

Possible scheduling criteria

- CPU use: keep the CPU as busy as possible
- Throughput: number of processes that complete their execution per time unit
- Turnaround time: amount of time to execute a particular process
- Waiting time: amount of time a process has been waiting in the ready queue
- Response time: amount of time it takes from when a request was submitted until the first response is produced (not the time it takes to output that response as it is possible that output overlaps subsequent computation)

Scheduling criteria

- Maximize CPU use and throughput; minimize turnaround time, waiting time, and response time
- Perhaps minimize the average; but it may be desirable to optimize the minimum or maximum times rather than the average (e.g., good response time in an interactive system)
- Interactive systems may prefer *predictable* output times (i.e., limit the *variance*), but little work has been done on this

First-Come, First-Served Scheduling (FCFS)

First process needing CPU gets it allocated (FIFO queue)
Nonpreemptive

Example: p1 (burst time 24); p2 (3); p3 (3)/arrive at t=0 in order

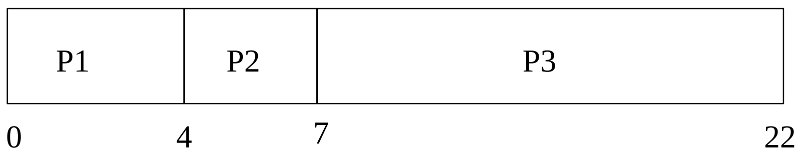
Gantt chart

average turnaround time = (24+27+30)/3 = 27

average wait time = (0+24+27)/3 = 17

First-Come, First-Served Scheduling (FCFS)

Example: p1 (burst time 4); p2 (3); p3 (15)/arrive at t=0



average turnaround time = (4+7+22)/3 = 11

average wait time = (0+4+7)/3 = 3 2/3

What happens if we reverse the order of arrival?

First-Come, First-Served Scheduling (FCFS)

Example: p3 (burst time 15); p2 (3); p1 (4)/arrive at t=0

average turnaround time = (15+18+22)/3 = 18 1/3

average wait time = (0+15+18)/3 = 11

average turnaround time was 11 is 18 1/3
average wait time was 3 2/3 is 11

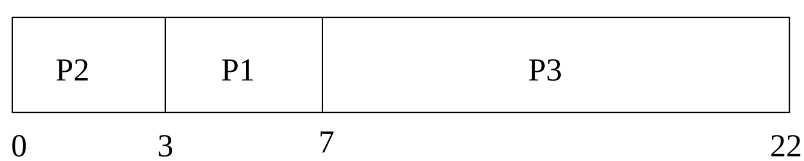
FCFS Scheduling

- Very simple to implement. Very quick to execute.
- Average wait time can be quite long and subject to variation depending on arrival time.
- Wait time not necessarily minimal (as seen by reordering processes).
- *Convoy effect:* short I/O bound processes wait behind CPU-bound process then execute quickly. CPU idle. Better device use possible with mix (e.g., shorter processes first).
- Nonpreemptive algorithm, so problematic for timesharing system (CPU bound holds up others)

Shortest Job First Scheduling (SJF)

- Give the CPU to the process with the smallest next CPU burst
- FCFS breaks ties

Example: as before, p1(4); p2(3); p3(15)



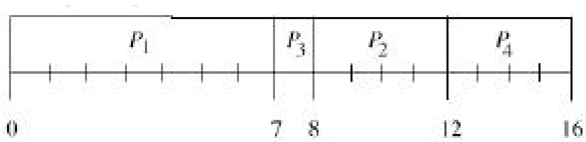
average turnaround time = (3+7+22)/3 = 10 2/3
average wait time = (0+3+7)/3 = 3 1/3

Shortest Remaining Time First

- A preemptive version of SJF scheduling
- If a new process arrives with CPU burst length less than the remaining time of the current executing process, preempt the current executing process.

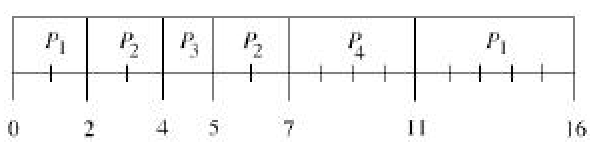
SJF/SRTF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4



SJF

Average waiting time =
(0 + 6 + 3 + 7)/4 = 4



SRTF

Average waiting time =
(9 + 1 + 0 + 2)/4 = 3

SJF Scheduling

- SJF can be proven to be optimal! Minimizes average waiting time for a given set of processes.
  - proof sketch: each process contributes to overall average waiting time so putting the one that contributes the least first decreases the average.
- But it requires that you know the future
  - cannot "know" the length of the next CPU burst
  - must predict future behavior (see following)
  - prediction on behavior will be wrong when process behaves inconsistently

Predicting the length of the next CPU burst

- Estimation based on previous behavior using exponential averaging. Let
  - t_n = actual length of n*th* CPU burst
  - τ_{n+1} = predicted value for the next CPU burst
  - α, 0 α 1
  - Define

$$\tau_{n+1} = \alpha \, t_n + (1 - \alpha) \, \tau_n$$

Exponential averaging

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- Common case: $\alpha = 0.5$

(Two cases for “flaky” CPU behavior)

Exponential averaging

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

- When we expand the formula we see that each successive term has less weight than its predecessor since α and $(1 - \alpha)$ are both between 0 and 1

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots$$
$$+ (1 - \alpha)\verb|^|j\ \alpha t_{n-j} + \ldots$$
$$+ (1 - \alpha)\verb|^|(n+1)\ \alpha \tau_0$$

Priority Scheduling (a general concept)

- The concept
  - Priority associated with each process
  - CPU allocation goes to the process with the highest priority
- can be either preemptive or nonpreemptive
- SJF is an example of (nonpreemptive) priority scheduling where the priority is based on the length of the next CPU burst

Priority Scheduling

- Preemptive priority scheduling: newly arriving process will preempt CPU if held by lower priority process
- Possible strategy to insure interactive response: process has higher priority after returning from I/O interrupt (can be abused in interactive environment--*how*?)
- One problems with priority scheduling is the possibility of <u>starvation</u> (indefinite blocking)
  - process waiting and ready to run that never gets CPU because of continuing stream of arriving higher-priority processes
  - <u>aging</u> might be one possible solution (increase priority with time)
  - Unix `nice` decreases priority as CPU use increases

Round Robin Scheduling (preemptive)

- For timesharing systems
- Define a *time quantum* (time slice): small unit of time, generally from 10 to 100 milliseconds
- Scheduling scheme
  - treat ready queue as FIFO queue
  - new processes added to tail
  - scheduler dispatches first process from head
  - if process releases CPU voluntarily, continue down queue, resetting quantum timer
  - at expiration of quantum, preempt process and return it to *tail* of ready queue

Round Robin Scheduling

- Example: p1 (burst time 15) ; p2 (3); p3 (5) quantum: 4

P1	P2	P3	P1	P3	P1	P1
0–4	4–7	7–11	11–15	15–16	16–20	20–23

0 4 7 11 15 16 20 23

p1 waits 0+7+1
p2 waits 4
p3 waits 7+4
average wait = 7 2/3

p1 ends at 23
p2 ends at 7
p3 ends at 16
average turnaround = 15 1/3

Round Robin Scheduling

- Interactive response good: if quantum *q* and *n* processes, a process must wait no longer than $(n-1)*q$ time units for CPU
- Average waiting time is quite long because of preemptions
- Performance depends heavily on size of quantum
  - If quantum infinite, same as FCFS (FCFS is special case of RR)
  - If quantum very small, appears (in theory) to users that there are *n* virtual processors, each running at $1/n$ the speed of the actual processor (given *n* processes)
  - But in reality the effects of context switching affects the performance of RR scheduling--**context switch overhead**

Round Robin Scheduling Quantum Size

- Time quantum should be large with respect to the context switch time to reduce effects of context switch overhead
- Smaller time quantum results in more context switches
- Time quantum too large, degenerates to FCFS case
- Metric from the text--80% of CPU bursts shorter than time quantum

Multilevel Queue Scheduling

- General class of algorithms involving *multiple* ready queues
- Appropriate for situations where processes are easily classified into different groups (e.g., foreground and background)
- Processes permanently assigned to one ready queue depending on some property of process (e.g., memory size, process priority, process type)
- Each queue has own scheduling algorithm (e.g., foreground could be RR while background could be FCFS)
- Scheduling as well between the queues--often a **fixed-priority preemptive scheduling**. For example, foreground queue could have absolute priority over background queue. (New foreground jobs displace running background jobs; no background until foreground queue empty).

Multilevel Queue Scheduling

- Example: five queues (highest to lowest)
  - system processes
  - interactive processes
  - interactive editing processes
  - batch processes
  - student processes
- One possibility for scheduling between the queues: each queue has absolute priority over lower-priority queues
- Another possibility: Each queue gets certain percentage of CPU time: e.g., foreground gets 80% and background gets 20%

Multilevel Feedback Queue Scheduling (MLF)

- Processes permitted to move between queues
- Needs policy about when this movement will take place
- Separate processes with different CPU burst behaviors. If CPU fails to live up to expectations it gets moved.
- Example: 3 queues
  – queue 0: quantum=8 (highest priority)
  – queue 1: quantum=16
  – queue 2: FCFS
  – New jobs enter queue 0. If don't finish in quantum move to tail of queue 1 and then to tail of queue 2
  – Higher numbered queue runs only when lower numbered queue is empty
  – Favors processes with CPU burst of 8 milliseconds or less

Multilevel Feedback Queue Scheduling (MLF)

- How many different levels? In other words, how many queues?
- Scheduling algorithm between queues.
- Scheduling algorithm for each queue.
- Method used to determine when to upgrade process to higher priority queue.
- Method used to determine when to demote process to lower priority queue.
- Method used to determine which queue a new process will enter when that process needs service.

MLF Scheduling
Example two

- (From Bic and Shaw)
- # priority levels: n+1, numbered 0 to n
- scheduling policy among levels: higher numbers have higher priority; queue n is highest and 0 lowest. All jobs at higher priority handled before any lower
- scheduling algorithm within queues: all queues use RR with a global quantum of “q”
- process upgrade: none
- process demotion: each level has associated time T_i where
  T_n = mq (m from the specifications; q quantum size)
  $0 <= i < n$, $T_i = 2^{(n-1)} * T_n$
  T_0 = infinity
  when process at level *i* has received T_i units of time, it is moved to next lower level
- New process: enters queue n (the highest level)

Special situations:
Multiple processor scheduling

- CPU scheduling more complex when multiple CPUs are available
- Limit consideration to *homogeneous* processors within a multiprocessor
- Can achieve *load sharing*
- Asymmetric multiprocessing--simpler than symmetric multiprocessing. Only one processor, the master server, handles system activities. Alleviates need for data sharing.

Special situations: Real-time scheduling

- Hard real-time systems: required to complete a critical task within a guaranteed amount of time
  - Resource reservation: statement of required resources (either accepted or rejected by system)
- Soft real-time computing: requires that critical processes receive priority over other processes
  - Must keep dispatch latency low, so real-time processes can start running faster
  - So long-running system calls may need to be preemptable. Insert preemption point into calls.

Algorithm Evaluation

- Deterministic modeling: takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queuing models: determine, and model, distribution of CPU and I/O bursts. Determine/model arrival-time distribution. Can then compute average throughput, utilization, waiting time, etc., for most algorithms.
- Simulations, perhaps using randomly-generated behaviors or perhaps using trace tapes.
- Implementation.

VAX/VMS OS Scheduling (a more complex example)

- (from Bic and Shaw)
- More complex than the strategies discussed so far but still has similar characteristics

VAX/VMS Scheduling (a real-world example)

- 32 priority levels. Divided into 2 groups of 16. Level 31 is highest priority.

31 to 16	Real-time processes
15 to 0	“Regular” processes

- Real time process priority fixed for duration of process
- Regular process priority varies based on recent execution history
  - base priority: assigned to process on creation. Specifies the minimum priority level
  - current priority: varies dynamically with recent execution history

VAX/VMS Scheduling

- Setting the current priority
  - Each system event has assigned *priority increment* to reflect the characteristics of the event
    - for example: terminal read > terminal write > disk i/o completion
  - When process is awakened due to one of these events, the priority increment is added to the current process priority with a maximum possible current priority of 15
  - Process enters appropriate level's queue
  - Process preempted after receiving its "fair share" of CPU. At this time decrement priority by 1 unless already at base priority. (Fair share is defined for the *process*, not the *level*)

VAX/VMS Scheduling

- Dispatch by current priority, hence real time processes always have priority over regular processes
- Preemption
  - real time: when (1) blocks itself., e.g., for I/O; (2) higher priority process arrives
  - regular: when (1), (2), or (3) exceeds time quantum (at which time it is demoted unless it is already at its base level)
- Compare to MLF
  - VAX/VMS has restriction of priority range between base priority and 15 (for regular processes)
  - Quantum associated with process, not global or with level. Dispatcher can discriminate among individual processes