CPU Scheduling

Thomas Plagemann

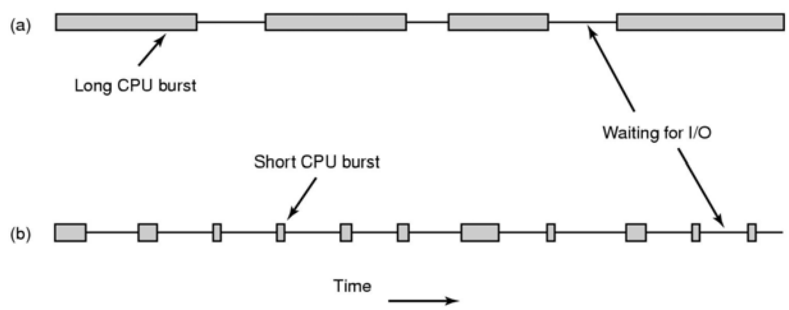
(with slides from Otto J. Anshus,
Kai Li, Pål Halvorsen and Andrew S. Tanenbaum)

Outline

- Goals of scheduling
- Scheduling algorithms:
  - FCFS/FIFO, RR, STCF/SRTCF
  - Priority (CTSS, UNIX, WINDOWS, LINUX)
  - Lottery
  - Fair share
  - Real-time: EDF and RM

Why Spend Time on Scheduling?

- Optimize the system to the given goals
- Example: CPU-Bound vs. I/O-Bound Processes:



- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O bound process

Scheduling Performance Criteria

- **CPU (resource) utilization**
  - 100%, but 40-90% normal
- **Throughput**
  - Number of "jobs" per time unit
  - Minimize overhead of context switches
  - Efficient utilization (CPU, memory, disk etc)
- **Turnaround time**
  - = $\text{time}_{\text{process arrives}}$ - $\text{time}_{\text{process exits}}$
  - = sum of all waiting times (memory, R_Q, execution, I/O, etc)
  - How fast a single job got through
- **Response time**
  - = $\text{time}_{\text{request starts}}$ - $\text{time}_{\text{response starts}}$
  - Having a small variance in Response Time is good (predictability)
  - Short response time: type on a keyboard
- **Waiting time**
  - in the Ready_Queue, for memory, for I/O, etc.
- **Fairness**
  - no starvation

Scheduling Algorithm Goals

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

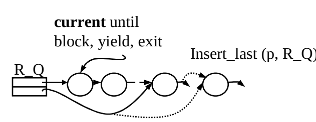
Batch systems

Interactive systems

Real-time systems

Non-Preemptive: FIFO (FCFS) Policy

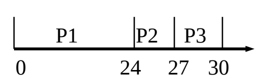
- Run to
  - to completion (old days)
  - until blocked, yield, or exit
- Advantages
- Disadvantage



Average Turnaround Time for CPU bursts:

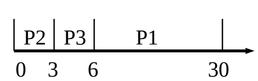
Process	Burst time
1	24
2	3
3	3

Arrival order: 1 - 2 - 3



$TT_{average} = (24+27+30)/3=\mathbf{27}$

Arrival order: 2 - 3 - 1



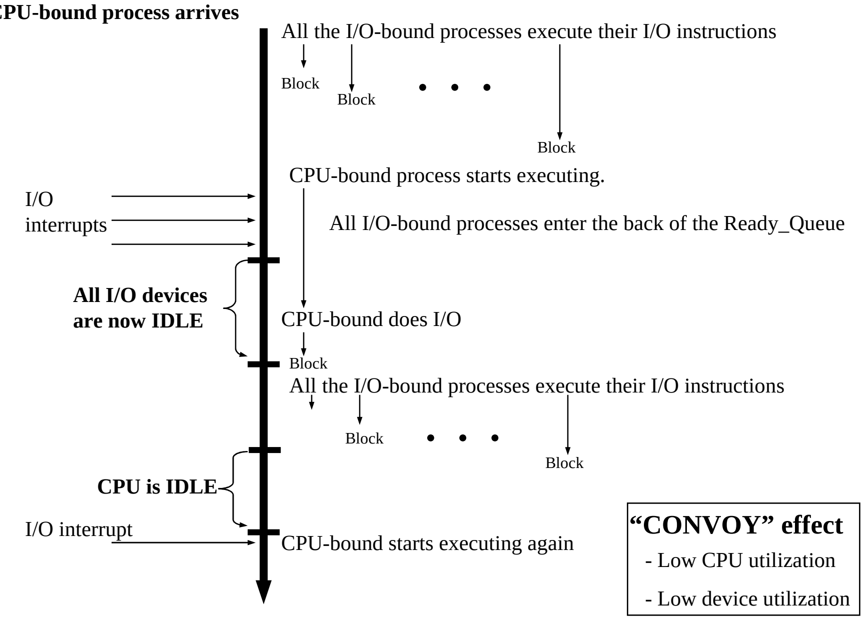
$TT_{average} = (3+6+30)/3=\mathbf{13}$

How well will FCFS handle:

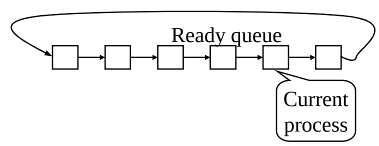
•Many processes doing I/O arrives

•One CPU-bound process arrives

Discussion topic FCFS



Round Robin



- FIFO queue
- n processes, each runs a time slice or quantum, q
  - each process gets 1/n of the CPU in max q time units at a time
- Max waiting time in Ready_Queue per process: (n-1) * q
- How do you choose the time slice?
  - Overhead vs. throughputs
  - Overhead is typically about 1% or less
    - interrupt handler + scheduler + dispatch
    - 2 context switches: going down, and up into new process
  - CPU vs. I/O bound processes

FIFO vs. Round Robin

- 10 jobs and each takes 100 seconds
- FIFO
- Round Robin
  - time slice 1s and no overhead
- Comparisons

Case: Time Slice Size

- Resource utilization example
  - **A** and **B** each uses 100% CPU
  - **C** loops forever (1ms CPU and 10ms disk)
- Large or small time slices?
  - nearly 100% of CPU utilization regardless of size
  - Time slice 100ms: nearly 5% of disk utilization with Round Robin
  - Time slice 1ms: nearly 85% of disk utilization with Round Robin
- What do we learn from this example?
  - The right (shorter) time slice can improve overall utilization
  - CPU bound: benefits from having longer time slices (>100 ms)
  - I/O bound: benefits from having shorter time slices (≤10ms)

Shortest Time to Completion First (STCF) (a.k.a. Shortest Job First)

- Non-preemptive
- Run the process having smallest service time
- Random, FCFS, … for “equal” processes

- Problem to establish what the running time of a job is
- Suggestions on how to do this?
  - Length of next CPU-burst
    - Assuming next burst = previous burst
    - Can integrate over time using a formula taking into account old and new history of CPU burst lengths
  - But mix of CPU and I/O, so be careful

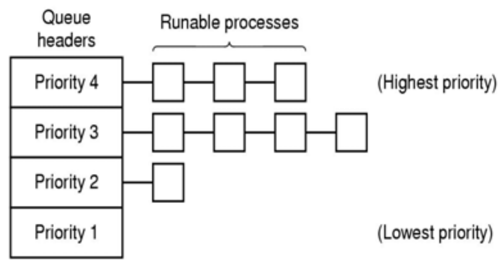
Shortest Remaining Time to Completion First (SRTCF) (a.k.a. Shortest Remaining Time First)

- Preemptive, dynamic version of STCF
- If a shorter job arrives, PREEMPT current, and do STCF again

- Advantage: high throughput, average turnaround is low
  (Running a short job before a long decreases the waiting time MORE for the short than it increases for the long!)
- Disadvantage: starvation possible, must know execution time

Priority Scheduling

- Assign each process a priority
- Run the process with highest priority in the ready queue first
- Multiple queues
- Advantage
  - (Fairness)
  - Different priorities according to importance
- Disadvantage
  - Users can hit keyboard frequently
  - Starvation: so should use dynamic priorities
- Special cases (RR in each queue)
  - FCFS (all equal priorities, non-preemptive)
  - STCF/SRTCF (the shortest jobs are assigned the highest priority)

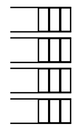


Multiple Queue

- Good for classes of jobs
  - real-time vs. system jobs vs. user jobs vs. batch jobs
- Multi level feedback queues
  - Adjust priority dynamically
    - Aging
    - I/O wait raises the priority
    - Memory demands, #open files, CPU:I/O bursts
  - Scheduling **between** the queues
    - Time slice (and cycle through the queues)
    - Priority typical:
      - Jobs start at highest priority queue
      - If timeout expires (used current time slices), drop one level
      - If timeout doesn't expires, stay or pushup one level
  - Can use different scheduling per queue
  - A job using doing much I/O is moved to an "I/O bound queue"

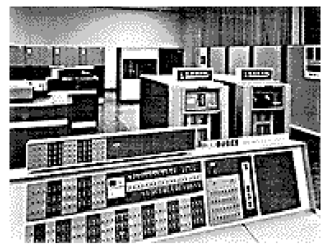
Compatible Time-Sharing System (CTSS)

- One of the first (1962) priority schedulers using multiple feedback queues (moving processes between queues)
- One process in memory at a time (high switch costs)
- Large slices vs. response time → priority classes



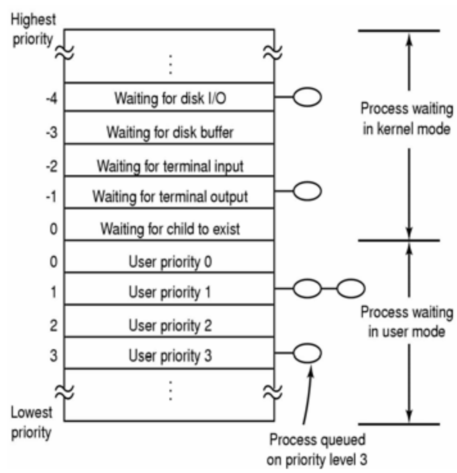
"Priority"	Time slices
0	1
1	2
2	4
3	8

- Each time the quantum was used, the process dropped one priority class (larger slice, less frequent)
- Interaction → back to highest priority class
- Short, interactive should run more often



Scheduling in UNIX

- Many versions



- User processes have positive priorities, kernel negative
- Schedule lowest priority first
- If a process uses the whole time slice, it is put back at the end of the queue (RR)
- Each second the priorities are recalculated:
  priority =
  CPU_usage (average #ticks)
  \+ nice (+- 20)
  \+ base (priority of last corresponding kernel process)

Scheduling in UNIX (4.4BSD)

- Similar to last slide
- Time slices of 100 ms
- Priorities is updated every 4th tick (40 ms)

  p_usrpri = PUSER + [p_estcpu x ¼] + 2 x p_nice

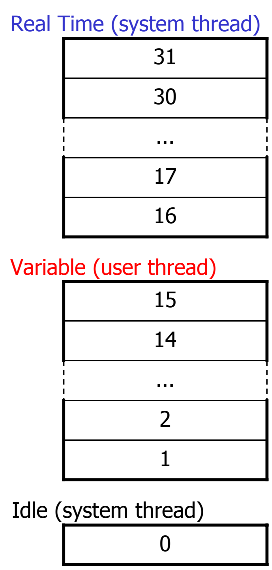
  – PUSER defaults to 50 (min), may be changed but here one uses only values between 50 and 127

  – p_estcpu =
    - running process: [(2 x load)/(2 x load + 1)] x p_estcpu + p_nice
    - blocked process: $[(2 \times load)/(2 \times load + 1)]^{p\_sleeptime} \times p\_estcpu$

  – p_nice defaults to 0

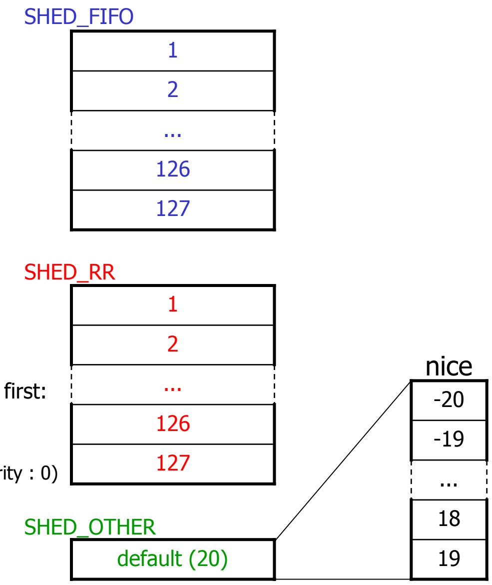
Scheduling in Windows 2000

- ✓ Preemptive kernel
- ✓ 32 priority levels - Round Robin (RR) in each
- ✓ Schedules threads individually
- ✓ Processor affinity
- ✓ Default time slices (3 quantums = 10 ms) of
  - ➢ 120 ms – Win2000 server
  - ➢ 20 ms – Win2000 professional/workstation
  - ➢ may vary between threads
- ✓ Interactive and throughput-oriented:
  - ➢ "Real time" – 16 system levels
    - fixed priority
    - may run forever
  - ➢ Variable – 15 user levels
    - priority may change – *thread priority* = process priority ± 2
    - uses much CPU cycles → drops
    - user interactions, I/O completions → increase
  - ➢ Idle/zero-page thread – 1 system level
    - runs whenever there are no other processes to run
    - clears memory pages for memory manager



Scheduling in Linux

- Preemptive kernel
- Threads and processes used to be equal, but Linux uses (in 2.6) thread scheduling
- SHED_FIFO
  - may run forever, no timeslices
  - may use it's own scheduling algorithm
- SHED_RR
  - each priority in RR
  - timeslices of 10 ms (quantums)
- SHED_OTHER
  - ordinary user processes
  - uses "nice"-values: 1≤ priority≤40
  - timeslices of 10 ms (quantums)
- Threads with highest *goodness* are selected first:
  - realtime (FIFO and RR):
    goodness = 1000 + priority
  - timesharing (OTHER):
    goodness = (quantum > 0 ? quantum + priority : 0)
- *Quantums* are reset when no *ready* process has quantums left:
  quantum = (quantum/2) + priority



Lottery Scheduling

- Motivations
  - SRTCF does well with average response time, but unfair
  - Guaranteed scheduling may be hard to implement
  - Adjust priority is a bit ad hoc. For example, at what rate?
- Lottery method
  - Give each job a number of tickets
  - Randomly pick a winning tickets
  - To approximate SRTCF, short jobs gets more tickets
  - To avoid starvation, give each job at least one ticket
  - Allows ticket exchange

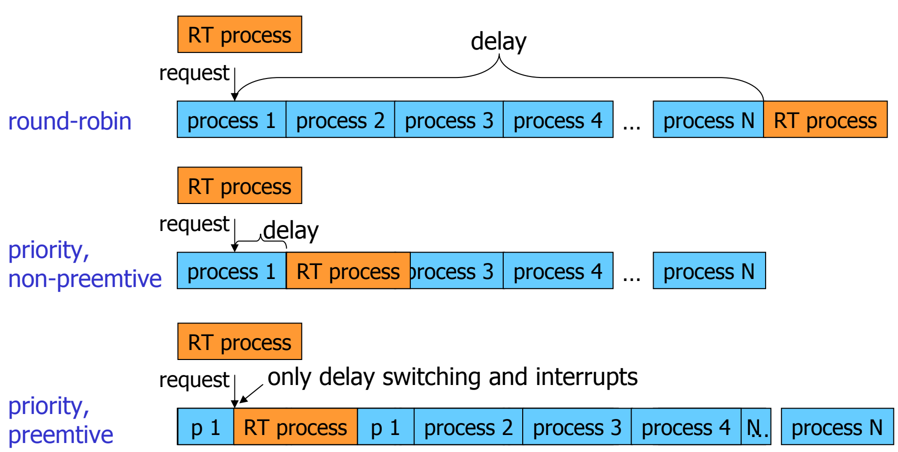
C.A. Waldspurger and W.E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management." Proc. of the 1st USENIX Symp. on Operating System Design and Implementation (OSDI). Nov 1994.

Fair Share

- Each PROCESS should have an equal share of the CPU
- History of recent CPU usage for each process
- Process with least recently used CPU time := highest priority
  - ➔ an editor gets a high priority
  - ➔ a compiler gets a low priority

- Each USER should have an equal share of the CPU
- Take into account the owner of a process
- History of recent CPU usage for each user

Real-Time Scheduling

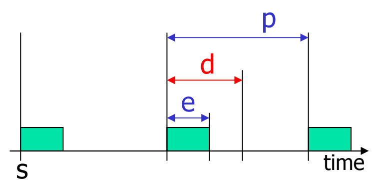


NOTE: preemption may also be limited to preemption points (fixed points where the scheduler is allowed to interrupt a running process) → giving larger delays

Real-Time Scheduling

- Real-time tasks are often periodic
(e.g., fixed frame rates and audio sample frequencies)

- Time constraints for a periodic task:
  - s – starting point
(first time the task require processing)
  - e – processing time
  - d – deadline
  - p – period
  - r – rate (r = 1/p)



  - 0 ≤ e ≤ d
(often d ≤ p: we'll use d = p – end of period, but **Σ**d ≤ **Σ**p is enough)
  - the **k**th processing of the task
    - is ready at time s + (k – 1) p
    - must be finished at time s + (k – 1) p + d
  - the scheduling algorithm must account for these properties

Schedulable Real-Time Systems

- Given
  - *m* periodic events
  - event *i* occurs within period P_i and requires C_i seconds

- Then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

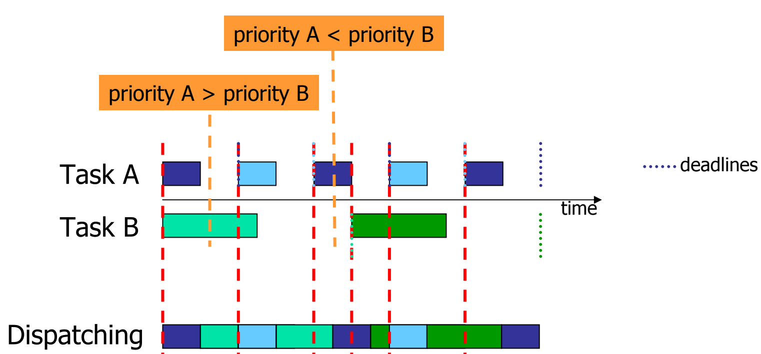
- Can we process 3 video streams, 25 fps,
each frame require 10 ms CPU time?
  - 3 * (10ms/40ms) = 3 * 25 * 0.010 = 0.75 < 1 → YES

Earliest Deadline First (EDF)

- Preemptive scheduling based on dynamic task priorities
- Task with closest deadline has highest priority
  → priorities vary with time
- Dispatcher selects the highest priority task
- Assumptions:
  - requests for all tasks with deadlines are periodic
  - the deadline of a task is equal to the end on its period (starting of next)
  - independent tasks (no precedence)
  - run-time for each task is known and constant
  - context switches can be ignored

Earliest Deadline First (EDF)

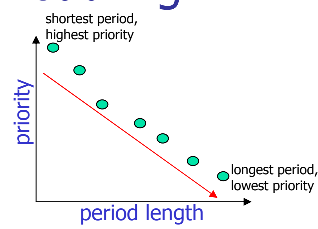
- Example:



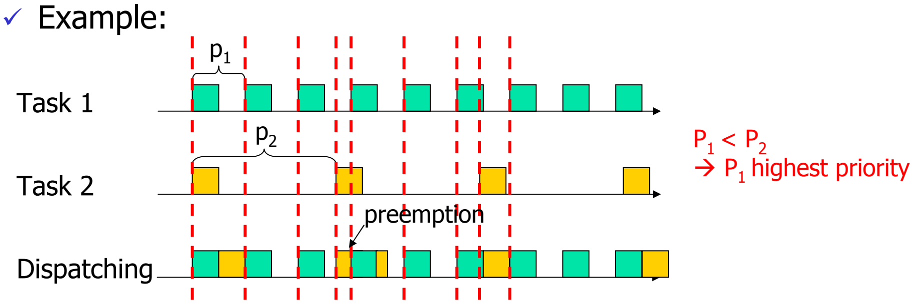
Rate Monotonic (RM) Scheduling

- ✓ Classic algorithm for hard real-time systems with one CPU
  [Liu & Layland '73]
- ✓ Pre-emptive scheduling based on static task priorities
- ✓ Optimal: no other algorithms with static task priorities can schedule tasks that cannot be scheduled by RM
- ✓ Assumptions:
  - requests for all tasks with deadlines are periodic
  - the deadline of a task is equal to the end on its period (starting of next)
  - independent tasks (no precedence)
  - run-time for each task is known and constant
  - context switches can be ignored
  - any non-periodic task has no deadline

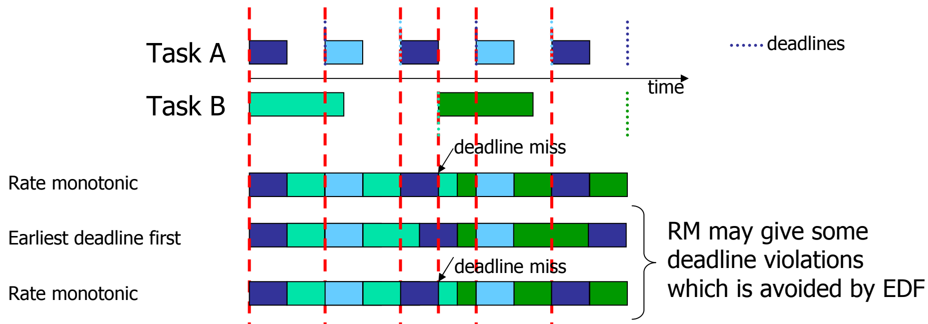
Rate Monotonic (RM) Scheduling



- ✓ Process priority based on task periods
  - task with shortest period gets highest *static* priority
  - task with longest period gets lowest *static* priority
  - dispatcher always selects task requests with highest priority
- ✓ Example:



EDF Versus RM – I



EDF Versus RM – II

- ✓ EDF
  - dynamic priorities changing in time
  - overhead in priority switching
  - QoS calculation – maximal throughput:

$$\sum_{\text{all streams } i} R_i \times P_i \le 1, \quad R - \text{rate}, P - \text{processing time}$$

- ✓ RM
  - static priorities based on periods
  - may map priority onto fixed OS priorities (like Linux)
  - QoS calculation:

$$\sum_{\text{all streams } i} R_i \times P_i \le \ln(2), \quad R - \text{rate}, P - \text{processing time}$$

Summary

- Scheduling performance criteria and goals are dependent on environment
- There exists several different algorithms targeted for various systems
- Traditional OSes like Windows, UniX, Linux, ... usually uses a priority-based algorithm
- The right time slice can improve overall utilization