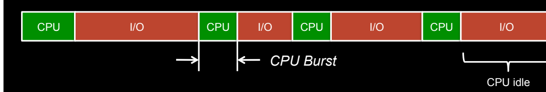
Operating Systems Design
7. Process Scheduling

Paul Krzyzanowski
pxk@cs.rutgers.edu

1

Process Behavior

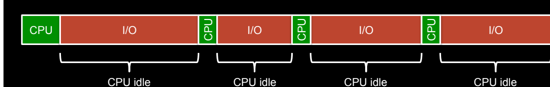
Most processes exhibit:
- Large # of short CPU bursts between I/O requests
- Small # of long CPU bursts between I/O requests



2

Process Behavior

Interactive process: mostly short CPU bursts

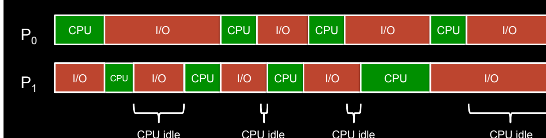


Compute process: mostly long CPU bursts

3

Process Scheduling

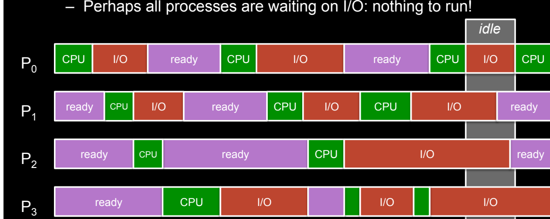
Goal:
- Maximize use of CPU & improve throughput
- Let another process run when the current one is waiting on I/O



4

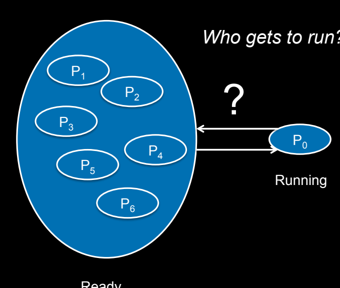
Process Scheduling

Reality:
- Some processes will use long stretches of CPU time
  - Preempt them and let another process run
- More processes may want the CPU: keep them in the *ready* list
- Perhaps all processes are waiting on I/O: nothing to run!



5

Process Scheduler



6

Switching processes

- Scheduling algorithm:
  - Policy: Makes the decision of who gets to run
- Dispatcher:
  - Mechanism to do the context switch

7

When does the scheduler make decisions?

Four events affect the decision:

1. Current process goes from *running* to *waiting* state
2. Current process terminates
3. Interrupt causes the scheduler to move a process from *running* to *ready: scheduler decides it's time for someone else to run*
4. Current process goes from *waiting* to *ready*
   I/O (including blocking events, such as semaphores) is complete

- Preemptive scheduler vs.
- Cooperative (non-preemptive) scheduler
  - CPU cannot be taken away
- Run-to-completion scheduler (old batch systems)

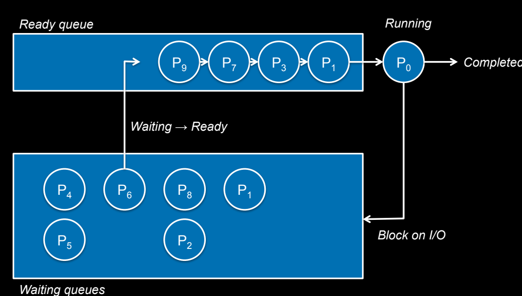
8

Scheduling algorithm goals

- Be fair (to processes? To users?)
- Be efficient: Keep CPU busy … and don't spend a lot of time deciding!
- Maximize throughput: minimize time users must wait
- Minimize response time
- Be predictable: jobs should take about the same time to run when run multiple times
- Minimize overhead
- Maximize resource use: try to keep devices busy!
- Avoid starvation
- Enforce priorities
- Degrade gracefully

9

First-Come, First-Served (FCFS)



10

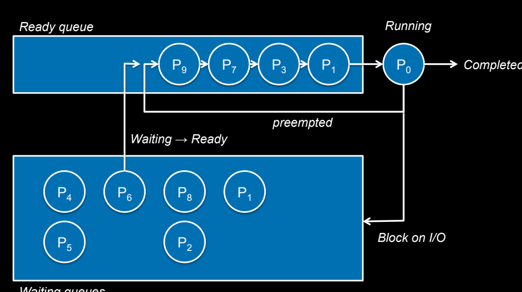
First-Come, First-Served (FCFS)

- Non-preemptive
- A process with a long CPU burst will hold up other processes
  - I/O bound jobs may have completed I/O and are ready to run: poor device utilization
  - Poor average response time

11

Round-Robin Scheduling

Preemptive: Process can not run for longer than a quantum (time slice)



12

Round-Robin Scheduling

- Performance depends on the time slice
  - Long time slice makes this similar to FCFS
  - Short time slice increases overhead % of context switching
- Advantages
  - Every process gets an equal share of the CPU
  - Easy to implement
  - Easy to compute average response time: *f*(# processes on list)
- Disadvantage
  - Giving every process an equal share isn't necessarily good
  - Highly interactive processes will get scheduled the same as CPU-bound processes

13

Shortest Remaining Time First Scheduling

- Sort jobs by anticipated CPU burst time
- Schedule shortest ones first
- Optimize average response time

Burst time	2	2	10	3	8	Total time = 25
Process	*E*	*D*	*C*	*B*	*A*	
Total run time	25	23	21	11	8	Mean time = 17.6

last ← *first*

Burst time	10	8	3	2	2	Total time = 25
Process	*C*	*A*	*B*	*D*	*E*	
Total run time	25	15	7	4	2	Mean time = 10.6

Mean completion time for a process falls by almost 40%!

14

Shortest Remaining Time First Scheduling

- Biggest problem: *we're optimizing with data we don't have!*
- All we can do is estimate
- Exponential average:

  $e_{n+1} = \alpha t_n + (1 - \alpha)e_n$

  α is a weight factor to balance the weight of the last burst period vs. historic periods ($0 \le \alpha \le 1$)

  If $\alpha = 0$: $e_{n+1} = e_n$ (recent history has no effect)
  If $\alpha = 1$: $e_{n+1} = \alpha t_n$ (use only the last burst time)

- Algorithm can be preemptive or non-preemptive
- Preemptive version is:
  Shortest remaining time first scheduling (vs. SJF)

15

Shortest Remaining Time First Scheduling

- Advantage
  - Short-burst jobs run fast
- Disadvantages
  - Long-burst (CPU intensive) jobs get a long mean waiting time
  - Rely on ability to estimate CPU burst length

16

Priority Scheduling

Round Robin assumes all processes are equally important

- Not true
  - Interactive jobs need high priority for good response
  - Long non-interactive jobs can worse treatment (get the CPU less frequently): *this goal led us to SRTF*
  - Users may have different status (e.g., administrator)
- Priority scheduling algorithm:
  - Each process has a priority number assigned to it
  - Pick the process with the highest priority
  - Processes with the same priority are scheduled round-robin

17

Priority Scheduling

- Priority assignments:
  - Internal: time limits, memory requirements, I/O:CPU ratio, …
  - External: assigned by administrators
- Static & dynamic priorities
  - Static priority: priority never changes
  - Dynamic priority: scheduler changes the priority during execution
    - Increase priority if it's I/O bound for better interactive performance or to increase device utilization
    - Decrease a priority to let lower-priority processes run
    - Example: use priorities to drive SJF/SRTF scheduling

18

Priority Scheduling: dealing with starvation

- Starvation
  - Process is blocked indefinitely
  - Steady stream of higher-priority processes keeps it from being scheduled
- Dealing with starvation: Process aging
  - Gradually increase the priority of a process so that eventually its priority will be high enough so it will be scheduled to run
  - Then bring it down again

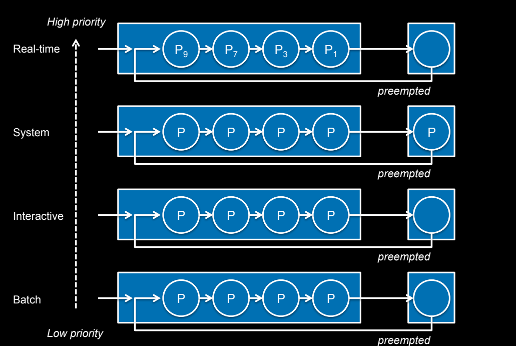
19

Multilevel Queues

- Priority classes
  - Examples: System processes, interactive processes, slow interactive processes, background non-interactive processes
  - Each priority class gets its own queue
  - Processes are permanently assigned to a specific queue
- Goals
  - Priority scheduler with queues per priority level
  - Each queue may have a different scheduling algorithm
  - Quantum is increased at each lower priority level
    - Lower-priority processes tend to be compute bound

20

Multilevel Queues



21

Multilevel Feedback Queues

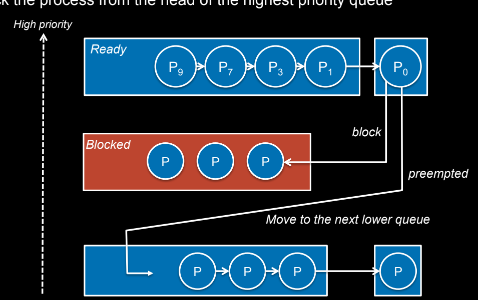
Goals

- *Allow processes to move between queues*
- Separate processes based on CPU burst behavior
  - I/O-bound processes will end up on higher-priority queues
  - If a process does not finish its quantum then it will stay at the same level, otherwise it moves to a lower level
- Quantum is increased at each lower priority level
  - Lower-priority processes tend to be compute bound
- Processes that wait too long may be moved to a higher-priority queue (aging)
- Each queue may have a different scheduling algorithm

22

Multilevel Feedback Queues

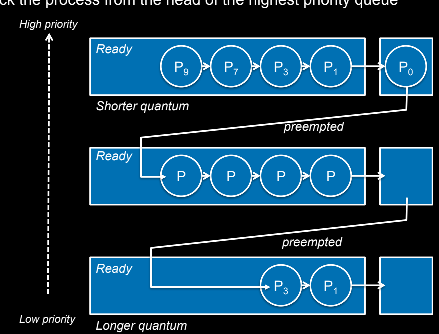
Pick the process from the head of the highest priority queue



23

Multilevel Feedback Queues

Pick the process from the head of the highest priority queue



24

Multilevel Feedback Queues

- Advantage
  - Good for separating processes based on CPU burst needs
  - Let I/O bound processes run often
  - Give CPU-bound processes longer chunks of CPU
  - No need to estimate interactivity! (Estimates were often flawed)
- Disadvantages
  - Priorities get controlled by the system.
    A process is considered important because it uses a lot of I/O
  - Processes whose behavior changes may be poorly scheduled
  - System can be gamed by scheduling bogus I/O

25

Symmetric multiprocessor scheduling

- Processor affinity
  - Try to reschedule a process onto the same CPU
  - Cached memory may be present on the CPU's cache
- Types of affinity
  - Hard : force a process to stay on the same CPU
  - Soft affinity: best effort, but the process may be rescheduled on a different CPU
    - Load balancing: ensure that CPUs are busy
    - It's better to run a job on another CPU than wait
    - If the run queue for a CPU is empty, get a job from another CPU's run queue: *pull migration*
    - Check load periodically: if not balanced, move jobs. *Push migration*

26

Hierarchy of symmetric multiprocessors

- Multiple processors
- Multiple cores
  - Shared caches among cores (e.g., Intel i7 cores share L3 cache)
- Hyperthreading
  - Presented as two cores to the operating system
  - Memory stall: CPU has to wait (e.g., to get data on a cache miss)
  - When the issuing logic can no longer schedule instructions from one thread and there are idle functional units in the CPU core, it will try to schedule a suitable instruction from the other thread.
- Good schedulers will know the difference

27

Scheduler Examples

28

Solaris Scheduler

- Priority-based scheduler: 170 priorities (0-169)
  - High priority → short quantum
- Six scheduling classes
  - Each class has priorities and scheduling algorithms

1. Time sharing (0-59)
   Default class. Dynamic priorities via a multilevel feedback queue *DEFAULT*
2. Interactive (0-59)
   Like TS but higher priority for in-focus windows in GUI
3. Real-time (100-159)
   Fixed priority, fixed time quantum; high priority values
4. System (60-99)
   Used to schedule kernel threads: run until they block or complete
5. Fair share (0-59)
   Processes scheduled on % of CPU
6. Fixed priority (0-59)
   Fixed priority

Highest priority (160-169): interrupt-handling threads

29

Solaris Scheduler

- Default class: time sharing
  - Multilevel feedback queue
  - Small time slice for high priority queue
  - Long time slice for low priority queue
- Interactive class: similar but gives windowing apps higher priority
- Highest priority: threads in the real-time class
- System class: runs kernel threads (scheduler & paging)
  - Not preempted
- Fair share: set of processes get a "CPU share"
- Fixed priority: like time-sharing but never adjusted

30

Windows Scheduler

- Two classes:
  - Variable class: priorities 1-15
  - Real-time class: priorities 16-31
- Each priority level has a queue
  - Pick the highest priority thread that is ready to run
- Relative priority
  - Threads have relative levels within their class
  - When a quantum expires, the thread's priority is lowered but never below the base
  - When a thread wakes from wait, the priority is increased
    - Higher increase if waiting for keyboard input
  - Priority is increased for foreground window processes

31

Linux Schedulers

- Linux 1.2: Round Robin scheduler (fast & simple)
- Linux 2.2: Scheduling classes
  - Classes: Real-time, non-real-time, non-preemptible
  - Support for symmetric multiprocessing
- Linux 2.4: O(N) scheduler
  - Iterates over every task at each scheduling event
  - If a time slice was not fully used, ½ of the remaining slice was added to the new time slice for the process.
  - "goodness" metric decided who goes next
  - One queue (in a mutex): no processor affinity

32

Linux 2.6 O(1) scheduler goals

Addressed three problems

- Scalability: O(1) instead of O(n) to not suffer under load
- Support processor affinity
- Support preemption

33

Linux 2.6 O(1) scheduler

- One runqueue per CPU: 140 priority lists serviced round robin
  - Two priority ranges: 0-99 for real-time; 100-140 for others
  - *High priority processes get a longer quantum!*
  - If a process uses its time slice, it will not get executed until all other processes exhaust their quanta
- runqueue data structure:
  - Two arrays sorted by priority value:
    - **Active**: all tasks with time remaining in their slices
    - **Expired**: all tasks that used up their time slice
  - Scheduler chooses the highest priority task from the active queue
  - When the active queue is empty, the expired queue becomes active

34

Linux 2.6 O(1) scheduler

- Real-time tasks: static priorities
- Non real-time tasks: dynamic priorities
  - I/O-bound processes get priority increased by up to 5 levels
  - CPU-bound processes get priority decreased up to 5 levels
  - Interactivity determined by %sleep : %compute time ratio
- SMP load balancing
  - Every 200ms, check if CPU loads are unbalanced
  - If so, move tasks from a loaded CPU to a less-loaded one
  - If a CPU's runqueue is empty, move from the other runqueue
- Downside of O(1) scheduler
  - A lot of code with complex heuristics

35

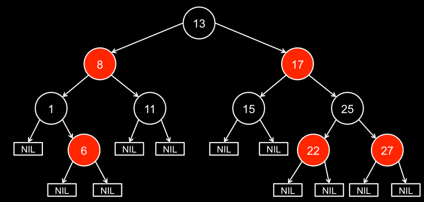
Linux Completely Fair Scheduler

- Latest scheduler (introduced in 2.6.23)
- Goal: give a "fair" amount of CPU time to tasks
- Keep track of time given to a task ("virtual runtime")
  - Also use "sleeper fairness": tasks get a "fair" share of the CPU even if they sleep from time to time
- Priorities
  - Used as a decay factor for the time a task is permitted to execute
  - Allowable time decreases for low priority tasks

36

Linux Completely Fair Scheduler

- No run queues
- Time-sorted read-black tree instead of a run queue
  - Self-balancing binary tree: search, insert, & delete in O(log n)



Most deserving: Used least amount of time

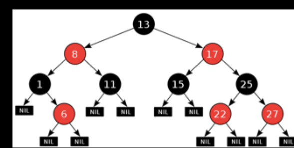
Least deserving: Used most time

From: http://en.wikipedia.org/wiki/File:Red-black_tree_example.svg

37

Linux Completely Fair Scheduler

- Goal: give a "fair" amount of CPU time to tasks
- Keep track of time given to a task ("virtual runtime")
  - Also "sleeper fairness": tasks that are waiting receive a fair share of the CPU when they are ready
- Time-sorted read-black tree instead of a run queue



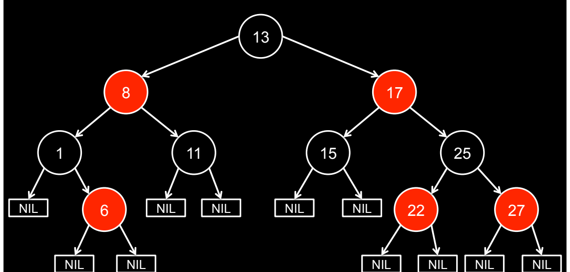
Red-black tree: self-balancing binary tree O(log n)

Most deserving: Used least amount of time

Least deserving: Used most time

From: http://en.wikipedia.org/wiki/File:Red-black_tree_example.svg

38



39

CFS: picking a process

- Scheduling decision:
  - Pick the leftmost task
- When a process is done:
  - Add execution time to the per-task run time count
  - Insert the task back in the queue
- Heuristic: *decay factors*
  - Determine how long a task can execute
  - Higher priority tasks have lower factors of decay.
  - Avoids having run queues per priority level

40

Group Scheduling

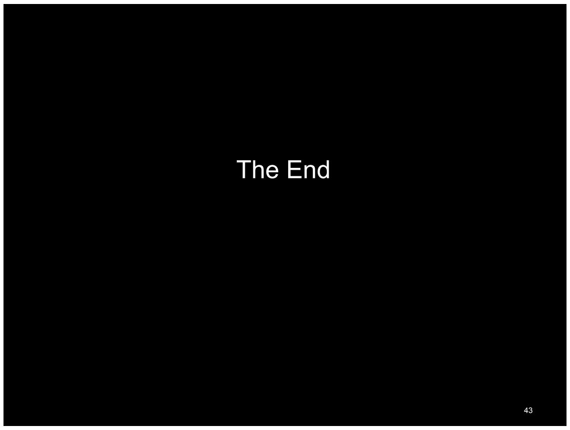
- Default operation: be fair to each task
- Assign one virtual runtime to a group of processes
  - Per user scheduling
  - cgroup pseudo file system interface for configuring groups
  - E.g., a user with 5 processes can get the same % of CPU as a user with 50 processes
- Default task group: init_task_group
- Improve interactive performance
  - A task calls *__proc_set_tty* to move to a tty task group
- /proc/sys/kernel/sched_granularity_ns
  - Tunable parameter to tune the scheduler between desktop (highly interactive) and server loads

41

More on the Linux scheduler

- Modular scheduler core: Scheduling classes
  - Scheduling class defines common set of functions that define the behavior of that scheduler
    - Add a task, remove a task, choose the next task
  - Each task belongs to a scheduling class
  - sched_fair.c
    - implements the CFS scheduler
  - sched_rt.c
    - implements a priority-based round-robin real-time scheduler
- Scheduling domains
  - Group one or more processors hierarchically
  - One or more processors can share scheduling policies

42


The End
43