CS3281 / CS5281

Process Scheduling

CS3281 / CS5281

Fall 2025

Some lecture slides borrowed and adapted from
"Operating Systems: Three Easy Pieces"

VANDERBILT UNIVERSITY

Overview

- The scheduler is the part of the OS kernel responsible for deciding which process runs and for how long
- There are many types of scheduling algorithms
    - Which one is "best" depends on many things, such as the expected workload and metrics
- These slides discuss scheduling basics
    - Subsequent slides discuss particular types of schedulers, such as those in Linux
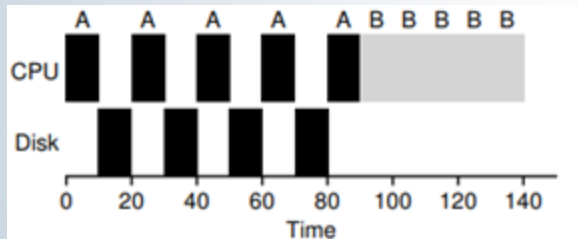
Scheduling

- A scheduling algorithm is responsible for deciding which process to run
- Some simple (impractical) scheduling algorithms (chapter 7 in your book). These algorithms schedule the *batch* (non-interactive) processes

  - First In First Out (FIFO): run each process to completion in the order they arrive. If multiple processes have the same arrival time, a process is selected randomly
    - Problems: what if a process runs "indefinitely"? What if a long running process happens to arrive first?
  - Shortest Job First (SJF): select the process that will run for the shortest time. If multiple processes have the same time, select a process with the earliest arrival time
    - Problem: we don't know how long a process will run ahead of time!
  - Shortest Time-to-Completion First (STCF): *preempt* the currently running process if another process has a shorter time to completion. If multiple processes have the same time, select a process with the earliest arrival time
    - Problem: long running processes will be *starved* if short running processes keep arriving

VANDERBILT
UNIVERSITY

Scheduling Metrics

- Metrics can be used to measure how good a scheduler is
- Examples:
  - Turnaround time: Time of completion - Time of arrival
    - A scheduler with poor turnaround time might be bad for CPU bound processes
    - Maximize the number of processes completed per unit of time
  - Response time: Time of first response - Time of arrival
    - Time between pressing a key and seeing a character on screen

    - A scheduler with a poor response time will feel "laggy" to interactive users

  - Throughput: Number of processes completed per unit time

  - Waiting time: The total amount of time that a process waits before getting scheduled on a CPU.
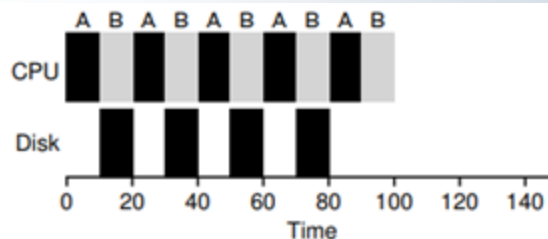   - ➢ Algorithms that optimize the turnaround time would reduce the waiting time

VANDERBILT
UNIVERSITY

What about I/O?

- All "interesting" programs do some I/O
- What should the scheduler do when a process is waiting for I/O?
  - Run another process instead! It's a bad idea to have a process spin while waiting for I/O
- Example from the book:

Figure 7.8: **Poor Use Of Resources**

vs

Figure 7.9: **Overlap Allows Better Use Of Resources**

More Realistic Scheduling: Round-Robin

- Round-Robin (RR) scheduling: instead of running a process to completion, run a process for a *time slice* (or time *quantum*), *preempt* the process, and context switch to the next process in the run queue; do this repeatedly
  - In other words: everybody gets a turn eventually!
  - Processes are in a single queue. The priority of a process is based on its position in the queue. (The process at the head of the queue has the highest priority.) When the time slice expires, the process moves to the tail of queue
  - RR is used in many real-world schedulers
- What triggers the preemption of the currently running process?
  - Simple case: the periodic timer interrupt (whose rate is configurable)
    - Timer interrupt occurs
    - OS handles this interrupt by executing the timer interrupt handler
    - Time interrupt handler checks how long current process has been running
    - If current process has run longer than its allotted time slice, the scheduler is invoked
    - The scheduler saves the context of the current process and selects a new process to run

ISIS

VANDERBILT
UNIVERSITY

CPU Bound vs. I/O Bound

- Processes are often *CPU bound* or *I/O bound*
  - CPU bound: when they run, they use the CPU continuously
    - Examples: MATLAB simulations, training a machine learning model, video encoding
  - I/O bound: when they run, they spend most of their time waiting for I/O
    - Examples: text editors, PowerPoint, shell
- Processes can switch between being I/O bound and CPU bound (and vice versa)
  - Example: when you do a "spell check" on a document, the editor goes from being I/O bound to CPU bound
- Question: how can a scheduler let CPU bound processes use the CPU but also quickly switch to an I/O bound process when input or output arrives?
  - Something the Linux CFS (covered later) aims to address

VANDERBILT
UNIVERSITY

Exercise

Schedule these processes with FIFO, SJF, STCF, and RR algorithms

- The time slice is 1 unit of time for RR
- If two or more processes are eligible to run at an instant, the one with the lowest number is scheduled

Process	Runtime	Arrival time
P1	10	0
P2	1	3
P3	2	1
P4	1	2
P5	5	0

VANDERBILT
UNIVERSITY

Exercise

FIFO

0	10	15	17	18	19
P1	P5	P3	P4	P2	

SJF

0	5	6	7	9	19
P5	P2	P4	P3	P1	

STCF

0	1	3	4	5	9	19
P5	P3	P2	P4	P5	P1	

RR

| 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	19
P1	P5	P1	P3	P5	P4	P1	P2	P3	P5	P1	P5	P1	P5	P1	

RR: When a new process arrives into system, it is placed at the tail of queue

Multi-Level Feedback Queues (MLFQ)

- MLFQ is a scheduling algorithm that tries to do two things:
  - Optimize turnaround time
    - This can be done by running shorter processes first, but we don't know how long a process will run!
  - Minimize response time
    - This makes the system feel responsive to interactive users
    - Round-robin is good for response time, but is bad for turnaround time
- MLFQ is cool because it addresses these two goals without knowing anything about the running times a priori

VANDERBILT
UNIVERSITY

MLFQ Basic Rules

- MLFQ has:
  - A number of distinct queues, each at a different priority level
  - At any given time, a process that is ready to run is on a single queue
  - Processes with higher priority are run first; processes with equal priority are run RR
- Rule 1: priority(A) > priority(B) => A runs
- Rule 2: priority(A) == priority(B), A and B run in RR
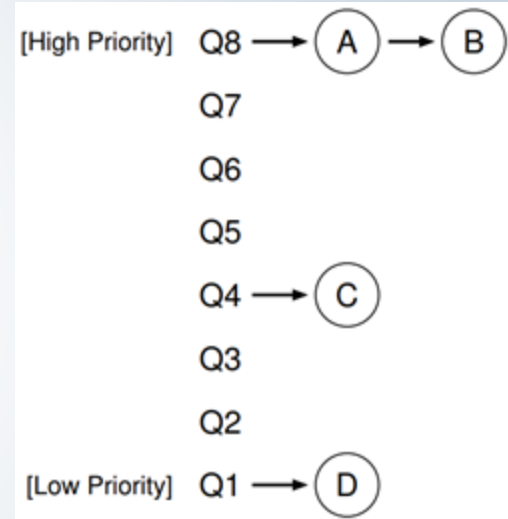
Changing Priority

- Rule 3: When process enters system, it is placed at highest priority (top queue)
- Rule 4a: If a process uses up an entire time allotment while running, its priority is reduced (it moves down to the tail of a queue with a lower priority)
  - The lower priority queues have longer time allotment
- Rule 4b: If a process gives up the CPU before its time allotment is up, it stays at the same priority level

VANDERBILT UNIVERSITY

Example of Moving through Queues

- Over time, a long running process will move to the bottom queue
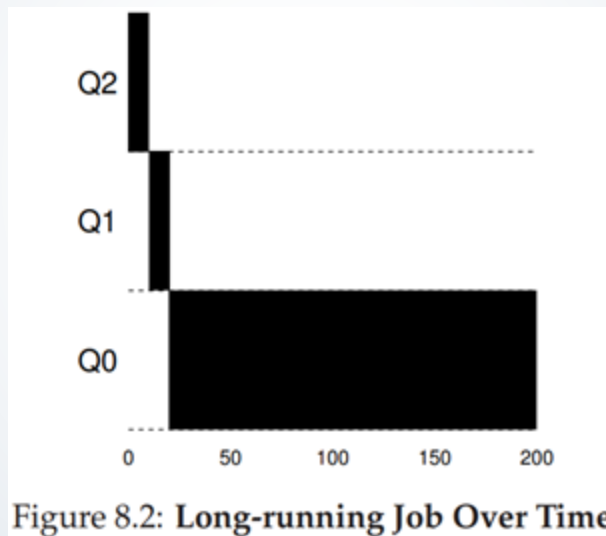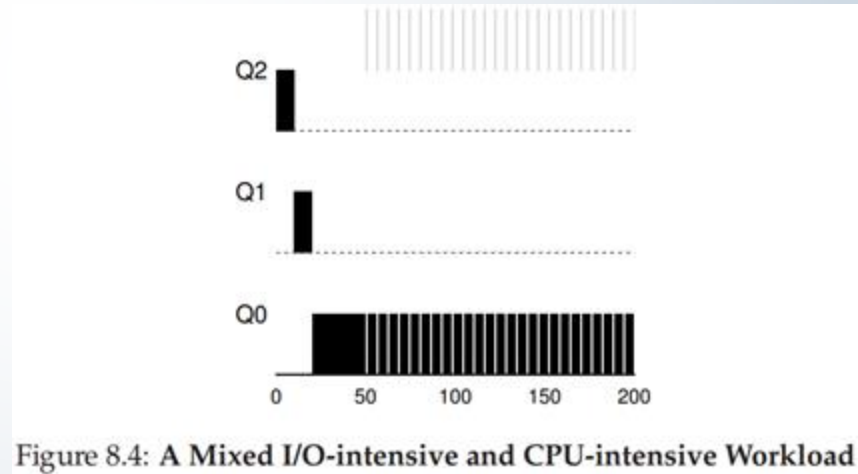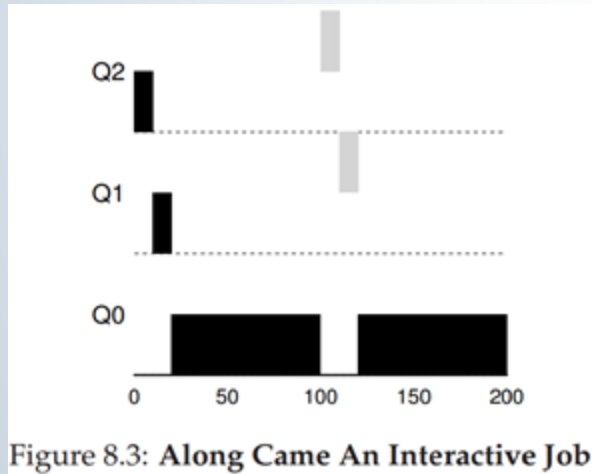- Example:

Figure 8.2: **Long-running Job Over Time**

More Examples

- Figure 8.3: a long running process and short-running job
- Figure 8.4: a long running process and an I/O bound process

Figure 8.3: **Along Came An Interactive Job**

Figure 8.4: **A Mixed I/O-intensive and CPU-intensive Workload**

Problems

- There are two problems with the design so far:
    - Processes have no way to move back up in priority (starvation)
        - Starvation is the indefinite delay to execute a process when the other processes are allowed to proceed
    - Processes can "game" the scheduler by giving up the CPU right before using the quantum
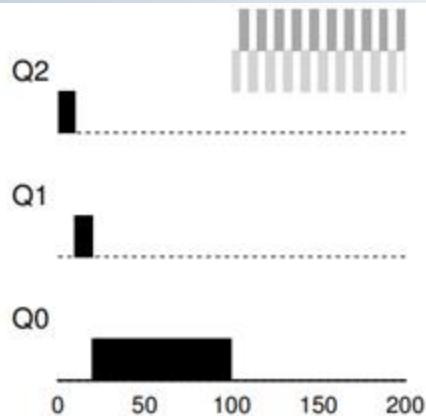

Figure 8.5: Without (Left) and With (Right) Priority Boost
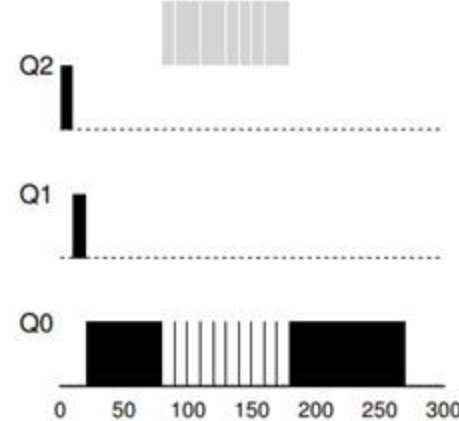
Figure 8.6: Without (Left) and With (Right) Gaming Tolerance

VANDERBILT
UNIVERSITY

Addressing the Problems

- We can keep processes from "gaming" the scheduler by moving them down a queue after they exhaust their cumulative time slice at a given level
- We can move processes up in priority periodically to keep them from starving
  - As an extension: we could "weight" processes so that some get to move up faster (but not too fast!)

VANDERBILT
UNIVERSITY

Summary of MFLQ Rules

- Rule 1: priority(A) > priority(B) => A runs
- Rule 2: priority(A) == priority(B), A and B run in RR
- Rule 3: when process enters system, it is placed at highest priority (top queue)
- Rule 4: once a process uses its allotment at a given level, its priority is reduced (it moves down to the tail of a queue with a lower priority)
- Rule 5: after some time period S, move all the jobs in the system to the topmost queue

VANDERBILT UNIVERSITY

Exercise

Consider these processes with the arrive times and runtimes.

Suppose you have 3 queues: the top queue uses RR with time slice of 8 time units, the middle queue uses RR with time slice of 16 time units and the bottom queue uses RR with time slice of 32 time units to schedule processes. How do you schedule these processes?

Process	Runtime	Arrival time
P1	36	0
P2	20	16
P3	12	20

Exercise

	0	8	16	24	32	40	52	56	68
Q2	P1		P2	P3				
Q1		P1			P1	P2	P3	
Q0								P1

VANDERBILT
UNIVERSITY

Multitasking: Preemptive Operating Systems

- Linux is a preemptive, multitasking OS
  - Preemptive: the scheduler decides when a process should stop and another should start
  - Multitasking: it can simultaneously interleave execution of more than one process
  - Windows, macOS, iOS, Android are also all multitasking, preemptive operating systems
- On a single-processor machine, this makes it seem like multiple processes are running concurrently
- On a multiprocessor (multicore) machine this allows multiple processes to run in parallel on different processors (cores)
- The *timeslice* (or quantum) is the amount of time a process runs before being preempted

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

ISIS

VANDERBILT
UNIVERSITY

Review: Preemption

- What triggers the preemption of the currently running process? Two things
- Case 1: returning from interrupt handler
  - Example: the timer interrupt occurs
  - OS handles this interrupt by executing the timer interrupt handler
  - Timer interrupt handler checks how long current process has been running
  - If current process has run longer than its allotted time slice, the scheduler is invoked
  - The scheduler saves the context of the current process and selects a new process to run
- Case 2: process makes a system call
  - Recall: process generates an intentional exception called a trap (e.g., `int 80h`)
  - OS executes a trap handler (to handle the system call)
  - When exiting the system call, the OS checks if it should invoke the scheduler
  - The scheduler, if run, checks for higher priority processes

VANDERBILT
UNIVERSITY

High-Level Concept: Lottery Scheduling

- Simple idea: every so often, hold a lottery to determine which process should run next
  - Processes that should run more often get more *tickets*, and thus more chances to win the lottery
  - Described in chapter 9 in the book
- Example:
  - Two processes, A and B
  - 100 tickets total. A has 75, B has 25
  - Pick a random winning ticket to see who gets to run next
  - A should get the CPU ~75% of the time, B should get it ~25% of the time
- Need a good random number generator, a data structure for tickets, and the total number of tickets!

VANDERBILT
UNIVERSITY

Completely Fair Scheduler Scenario

- The CFS in Linux is similar to lottery scheduling
- Consider the following scenario
  - Two tasks: a text editor (I/O bound) and a MATLAB simulation (CPU bound)
  - The text editor needs to respond to key presses quickly
  - MATLAB needs CPU time to perform a simulation
- In the ideal case:
  - Give a larger proportion of CPU time to text editor
    - But not because it needs it! Because we want it to have time the moment it needs it.
  - Allow the text editor to preempt MATLAB as soon as input is available (i.e., when a key is pressed)
    - This will make it responsive and give good interactive performance

VANDERBILT
UNIVERSITY

Virtual Runtime

- So how is the timeslice calculated?
  - Too low: increases "fairness" but also increases overhead (due to context switching)
  - Too high: decreases overhead but also decreases fairness
- Timeslice uses a "magic value" (empirically determined) called `sched_latency` (typical value = 48ms) to calculate the timeslice
  - Timeslice = sched_latency / # of processes
- Nice value: parameter of a process controllable by user
  - Goes from -20 to +19; default of 0;
    - Negative implies higher priority (you are "less nice")

Completely Fair Scheduler Scenario (cont.)

- Divides CPU time evenly among processes
  - Instead of a "fixed" timeslice, CFS calculates how long each process should run as a function of the total number of runnable processes
    - Use the nice value to weight this proportion of processor a process receives
    - If all nice values are equal: all processes get an equal proportion of processor time
  - Uses a simple counting technique known as virtual runtime (vruntime)
    - Lower vruntime => a process hasn't had its "fair share"

Timeslice

- Use the nice value to "weight" the timeslice

$$\text{time\_slice}_k = \frac{\text{weight}_k}{\sum_{n=0}^{n-1} \text{weight}_i} \cdot \text{sched\_latency} \qquad (9.1)$$

- Also scale the `vruntime`:

prio_to_weight[20]

$$\text{vruntime}_i = \text{vruntime}_i + \frac{}{\text{weight}_i} \cdot \text{runtime}_i \qquad (9.2)$$

- Table of weights:

```
static const int prio_to_weight[40] = {
/* -20 */      88761,      71755,      56483,      46273,      36291,
/* -15 */      29154,      23254,      18705,      14949,      11916,
/* -10 */       9548,       7620,       6100,       4904,       3906,
/*  -5 */       3121,       2501,       1991,       1586,       1277,
/*   0 */       1024,        820,        655,        526,        423,
/*   5 */        335,        272,        215,        172,        137,
/*  10 */        110,         87,         70,         56,         45,
/*  15 */         36,         29,         23,         18,         15,
};
```

VANDERBILT
UNIVERSITY

Virtual Runtime (cont.)

- But what if there are too many processes?
  - Define a minimum timeslice value; never use timeslice lower than this

- But how do we give processes "priority"?
  - In CFS: give them a larger share of the CPU

- Nice value: parameter of a process controllable by user
  - Goes from -20 to +19; default of 0;
  - Negative implies higher priority (you are "less nice")

Virtual Runtime (cont.)

- As a process runs, it accumulates vruntime
- When scheduler needs to pick a new process, it picks the process with the lowest vruntime