CS 111: Operating System Principles
Lecture 7

Scheduling

2.0.1

Jon Eyolfson
July 13, 2021

There are Preemptible and Non-preemptible Resources

A preemptible resource can be taken away and used for something else

e.g. a CPU

The resource is shared through scheduling

A non-preemptible resource can not be taken away without acknowledgment

e.g. disk space

The resource is shared through allocations and deallocations

Note: Parallel and distributed systems may allow you to allocate a CPU

A Dispatcher and Scheduler Work Together

A dispatcher is a low-level mechanism

- Responsible for context switching

A scheduler is a high-level policy

- Responsible for deciding which processes to run

The Scheduler Runs Whenever a Process Changes State

First let's consider non-preemptable processes

Once the process starts, it runs until completion

In this case, the scheduler will only make a decision when the process terminates

Preemptive allows the operating system to run the scheduler at will

Check `uname -v`, your kernel should tell you it's preemptable

Metrics

Minimize waiting time and response time

- Don't have a process waiting too long (or too long to start)

Maximize CPU utilization

- Don't have the CPU idle

Maximize throughput

- Complete as many processes as possible

Fairness

- Try to give each process the same percentage of the CPU

First Come First Served (FCFS)

The most basic form of scheduling

The first process that arrives gets the CPU

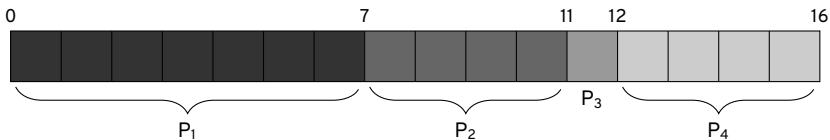
Processes are stored in a FIFO queue in arrival order

A Gantt Chart Illustrates the Schedule

Consider the following processes:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	0	4
P_3	0	1
P_4	0	4

Assume, $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$. For FCFS, our schedule is:



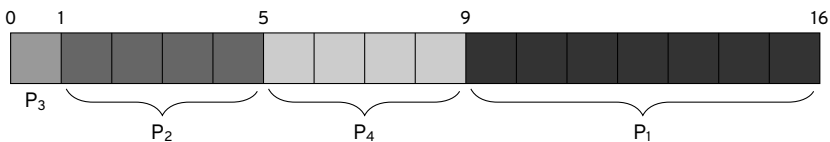
What is the average waiting time?

What Happens to Our Waiting Time with a Different Arrival Order

Consider the same processes:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	0	4
P_3	0	1
P_4	0	4

Assume, $P_3 \rightarrow P_2 \rightarrow P_4 \rightarrow P_1$. For FCFS, our schedule is:



What is the average waiting time now?

Shortest Job First (SJF)

A slight tweak to FCFS, we always schedule the job with the shortest burst time first

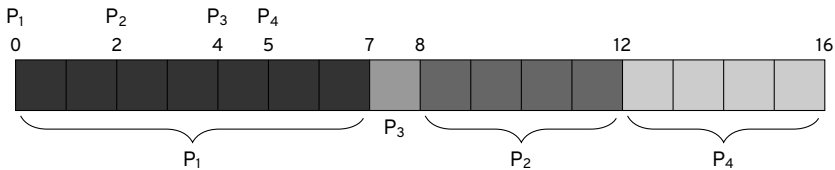
We're still assuming no preemption

SJF Minimizes the Average Wait Time over FCFS

Consider the same processes with different arrival times:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For SJF, our schedule is (arrival on top):



Average waiting time: $\frac{0+6+3+7}{4} = 4$

SJF is Not Practical

It is provably optimal at minimizing average wait time (if no preemption)

You will not know the burst times of each process

- You could use the past to predict future executions

You may starve long jobs (they may never execute)

Shortest Remaining Time First (SRTF)

Changing SJF to run with preemptions requires another tweak

We'll assume that our minimum execution time is one unit

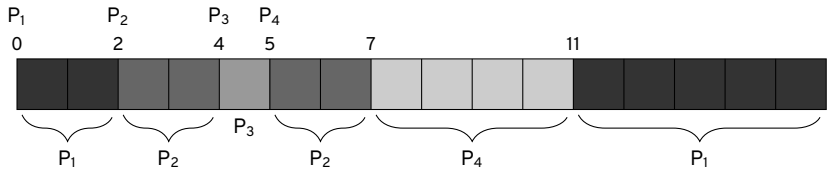
Similar to SJF, this optimizes the average waiting time

SRTF Reduces the Average Wait Time Compared to SJF

Consider the same processes and arrival times as SJF:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For SRTF, our schedule is (arrival on top):



Average waiting time: $\frac{9+1+0+2}{4} = 3$

Round-Robin (RR)

So far we haven't handled fairness (it's a trade off with other metrics)

The operating system divides execution into time slices (or quanta)

- An individual time slice is called a quantum

Maintain a FIFO queue of processes similar to FCFS

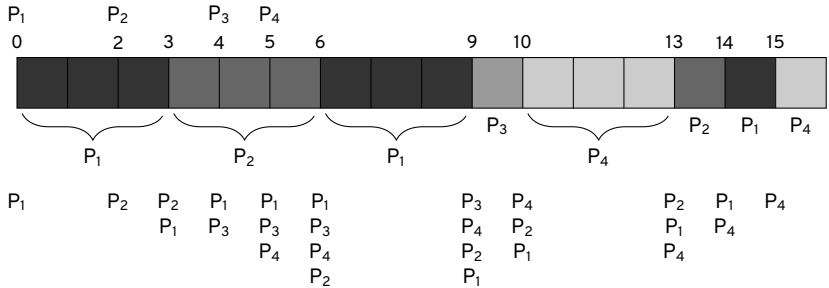
- Preempt if still running at end of quantum and re-add to queue

What are practical considerations for determining quantum length?

RR with a Quantum Length of 3 Units

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For RR, our schedule is (arrival on top, queue on bottom):



Metrics for RR (3 Unit Quantum Length)

Number of context switches: 7

Average waiting time: $\frac{8+8+5+7}{4} = 7$

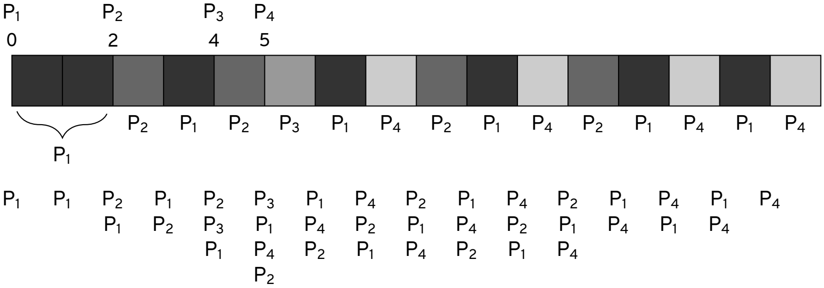
Average response time: $\frac{0+1+5+5}{4} = 2.75$

Note: on ties (a new process arrives while one is preempted), favor the new one

RR with a Quantum Length of 1 Units

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For RR, our schedule is (arrival on top, queue on bottom):



Metrics for RR (1 Unit Quantum Length)

Number of context switches: 14

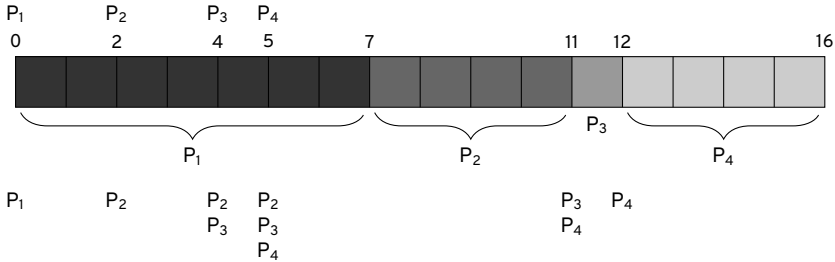
Average waiting time: $\frac{8+6+1+7}{4} = 5.5$

Average response time: $\frac{0+0+1+2}{4} = 0.75$

RR with a Quantum Length of 10 Units

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For RR, our schedule is (arrival on top, queue on bottom):



Metrics for RR (10 Unit Quantum Length)

Number of context switches: 3

Average waiting time: $\frac{0+5+7+7}{4} = 4.75$

Average response time: $\frac{0+5+7+7}{4} = 4.75$

Note: in this case it's the same as FCFS without preemptions

RR Performance Depends on Quantum Length and Job Length

RR has low response good interactivity

- Fair allocation of CPU
- Low average waiting time (when job lengths vary)

The performance depends on the quantum length

- Too high and it becomes FCFS
- Too low and there's too many context switches (overhead)

RR has poor average waiting time when jobs have similar lengths

Scheduling Involves Trade-Offs

We looked at few different algorithms:

- First Come First Served (FCFS) is the most basic scheduling algorithm
- Shortest Job First (SJF) is a tweak that reduces waiting time
- Shortest Remaining Time First (SRTF) uses SJF ideas with preemptions
- SRTF optimizes lowest waiting time (or turnaround time)
- Round-robin (RR) optimizes fairness and response time

We Could Add Priorities

We may favor some processes over others

- Assign each process a priority

Run higher priority processes first, round-robin processes of equal priority

- Can be preemptive or non-preemptive

Priorities Can Be Assigned an Integer

We can pick a lower, or higher number, to mean high priority

- In Linux -20 is the highest priority, 19 is the lowest

We may lead processes to *starvation* if there's a lot of higher priority processes

One solution is to have the OS dynamically change the priority

- Older processes that haven't been executed in a long time increase priority

Priority Inversion is a New Issue

We can accidentally change the priority of a low priority process to a high one

- This is caused by dependencies, e.g. a high priority depends a low priority

One solution is *priority inheritance*

- Inherit the highest priority of the waiting processes
- Chain together multiple inheritances if needed
- Revert back to the original priority after dependency

A Foregound Process Can Recieve User Input, Background Can Not

Unix background process when: process group ID differs from its terminal group ID

- You do not need to know this specific definition

The idea is to separate processes that users interact with:

- Foreground processes are interactable and need good response time
- Background processes may not need good response time, just throughput

We Can Use Multiple Queues for Other Purposes

We could create different queues for foreground and background processes:

- Foreground uses RR
- Background uses FCFS

Now we have to schedule between queues!

- RR between the queues
- Use a priority for each queue

Scheduling Can Get Complicated

There's no "right answer", only trade-offs

We haven't talked about multiprocessor scheduling yet

We'll assume symmetric multiprocessing (SMP)

- All CPUs are connected to the same physical memory
- The CPUs have their own private cache (at least the lowest levels)

One Approach is to Use the Same Scheduling for All CPUs

There's still only one scheduler

- It just keeps adding processes while there's available CPUs

Advantages

- Good CPU utilization
- Fair to all processes

Disadvantages

- Not scalable (everything blocks on global scheduler)
- Poor cache locality

This was the approach in Linux 2.4

We Can Create Per-CPU Schedulers

When there's a new process, assign it to a CPU
- One strategy is to assign it to the CPU with the lowest number of processes

Advantages
- Easy to implement
- Scalable (there's no blocking on a resource)
- Good cache locality

Disadvantages
- Load imbalance
  - Some CPUs may have less processes, or less intensive ones

We Can Compromise between Global and Per-CPU

Keep a global scheduler that can rebalance per-CPU queues

- If a CPU is idle, take a process from another CPU (work stealing)

You may want more control over which processes can switch

- Some may be more sensitive to caches

Use *processor affinity*

- The preference of a process to be scheduled on the same core

This is a simplified version of the O(1) scheduler in Linux 2.6

Another Strategy is “Gang” Scheduling

Multiple processes may need to be scheduled simultaneously

The scheduler on each CPU cannot be completely independent

“Gang Scheduling” (Coscheduling)

Allows you to run a set of processes simultaneously (acting as a unit)

This requires a global context-switch across all CPUs

Real-Time Scheduling is Yet Another Problem

Real-time means there are time constraints, either for a deadline or rate

e.g. audio, autopilot

A hard real-time system

Required to guarantee a task completes within a certain amount of time

A soft real-time system

Critical processes have a higher priority and the deadline is met in practice

Linux is an example of soft real-time

Linux Also Implements FCFS and RR Scheduling

You can search the source tree: FCFS (`SCHED_FIFO`) and RR (`SCHED_RR`)

Use a multilevel queue scheduler for processes with the same priority

- Also let the OS dynamically adjust the priority

Soft real-time processes:

- Always schedule the highest priority processes first

Normal processes:

- Adjust the priority based on aging

Real-Time Processes Are Always Prioritized

The soft real-time scheduling policy will either be `SCHED_FIFO` or `SCHED_RR`

- There are 100 static priority levels (0—99)

Normal scheduling policies apply to the other processes (`SCHED_NORMAL`)

- By default the priority is 0
- Priority ranges from $[-20, 19]$

Processes can change their own priorities with system calls:

- `nice`, `sched_setscheduler`

Linux Scheduler Evolution

2.4–2.6, a $O(N)$ global queue

- Simple, but poor performance with multiprocessors and many processes

2.6–2.6.22, a per-CPU run queue, $O(1)$ scheduler

- Complex to get right, interactivity had issues
- No guarantee of fairness

2.6.23–Present, the completely fair scheduler (CFS)

- Fair, and allows for good interactivity

The O(1) Scheduler Has Issues with Modern Processes

Foreground and background processes are a good division
- Easier with a terminal, less so with GUI processes

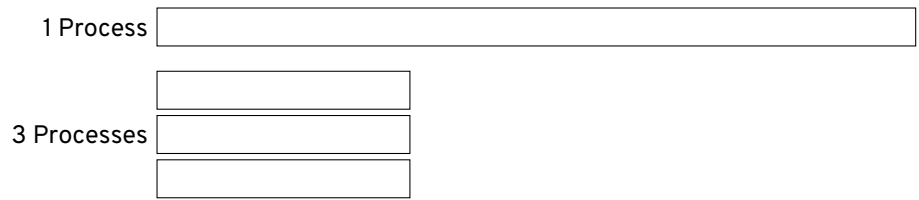
Now the kernel has to detect interactive processes with heuristics
- Processes that sleep a lot may be more interactive
  - This is ad hoc, and could be unfair

How would we introduce fairness for different priority processes?
- Use different size time slices
- The higher the priority, the larger the time slice
  - There are also situations where this ad hoc solution could be unfair

Ideal Fair Scheduling

Assume you have an infinitely small time slice

If you have n processes, each runs at $\frac{1}{n}$ rate



CPU usage is divided equally among every process

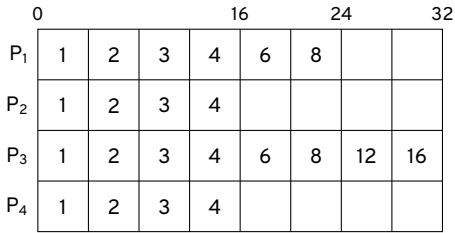
Example IFS Scheduling

Consider the following processes:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	0	4
P_3	0	16
P_4	0	4

Assume that each vertical slice can execute 4 time units.

Each box represents the time units spend executing



IFS is the Fairest but Impractical Policy

This policy is fair, every process gets an equal amount of CPU time

Boosts interactivity, has the ideal response time

However, this would perform way too many context switches

You have to constantly scan all processes, which is $O(N)$

Completely Fair Scheduler (CFS)

For each runnable process, assign it a “virtual runtime”

- At each scheduling point where the process runs for time t
  - Increase the virtual runtime by $t \times$ weight (based on priority)

The virtual runtime monotonically increases

- Scheduler selects the process based on the lowest virtual runtime
  - Compute its dynamic time slice based on the IFS

Allow the process to run, when the time slice ends repeat the process

CFS is Implemented with Red-Black Trees

A red-black tree is a self-balancing binary search tree

- Keyed by virtual runtime
  - $O(\lg N)$ insert, delete, update
  - $O(1)$ find minimum

The implementation uses a red-black tree with nanosecond granularity

- Doesn't need to guess the interactivity of a process

CFS tends to favour I/O bound processes by default

- Small CPU bursts translate to a low virtual runtime
  - It will get a larger time slice, in order to catch up to the ideal

Scheduling Gets Even More Complex

There are more solutions, and more issues:

- Introducing priority also introduces priority inversion
- Some processes need good interactivity, others not so much
- Multiprocessors may require per-CPU queues
- Real-time requires predictability
- Completely Fair Scheduler (CFS) tries to model the ideal fairness