Scheduling

Computer Operating Systems, Fall 2025

Instructors: Joel Ramirez

Head TAs: Maya Huizar, Akash Kaukuntla, Vedansh Goenka, Joy Liu

TAs:

Eric Zou	Joseph Dattilo	Aniket Ghorpade	Shriya Sane
Zihao Zhou	Eric Lee	Shruti Agarwal	Yemisi Jones
Connor Cummings	Shreya Mukunthan	Alexander Mehta	Raymond Feng
Bo Sun	Steven Chang	Rania Souissi	Rashi Agrawal
Sana Manesh			

Administrivia

- Penn-shell is due this ~~Friday, October 3rd~~ Monday, October 6th
  - If you have any questions, please stop by office hours! We are here to help.
  - Latest time to turn it in is Friday, October 10th during fall break…don't do this to yourself.

- Midterm will be on Thursday, October 16 from 5:15 – 6:45
  - Locations: Towne 100 & Wu and Chen in Levine
  - Towne 100
    - A – M
  - Wu and Chen
    - N - Z
  - Practice exams and the official post will go out later today.

- Penn OS will come out Friday, October 17th after the midterm!

Administrivia

- PennOS:
  - Specifications and team sign-up to be posted Friday (day after exam)
  - Done in groups of 4
  - Partner signup due by end of day on Monday, **10/20**
    - Those left unassigned will be randomly assigned the next morning (Tuesday the 21st)
  - Lecture dedicated to PennOS in class on Tuesday the 21st. ***Highly recommend you go.***

- ***NO IN-PERSON LECTURE on* 10/07**
  - I will record the entire lecture on Zoom this weekend and then upload it to canvas on Tuesday. There will be a thread on Ed incase there are any questions.
  - Do not come to AGH/try to join on zoom…I will not be here nor there…

Lecture Outline

- High Level: Scheduling
- Non-Preemptive Algorithms
  - First Come First Serve (FCFS)
  - Shortest Job First
- Preemptive Algorithms
  - Round Robin
  - PennOS Round Robin
  - Round Robin Variants
  - Completely Fair Scheduling (CFS)
- "Nice" Threads
- Extra: Earliest Eligible Virtual Deadline First, Interactive Tasks, Etc.

OS as the Scheduler

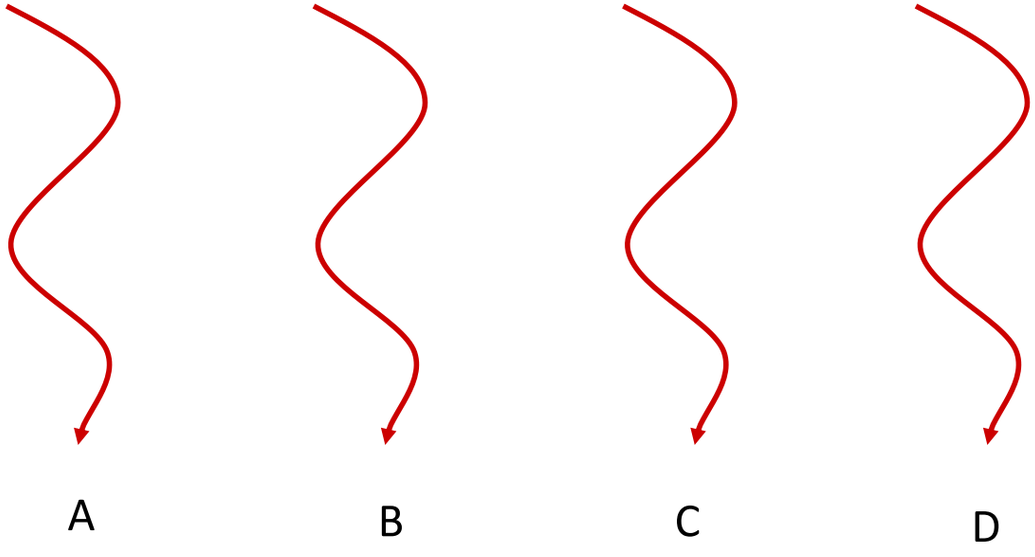
- The scheduler is code that is part of the kernel (OS)

- The scheduler runs when a thread:
  - starts ("arrives to be scheduled"),
  - Finishes
  - Blocks (e.g., waiting on something, usually some form of I/O)
  - Has run for a certain amount of time

- It is responsible for scheduling threads
  - Choosing ***which*** one to run
  - Deciding ***how*** long to run it

Scheduling Threads

- Scheduler: It is responsible for scheduling threads
- From 4 threads, **which do we select to run**?



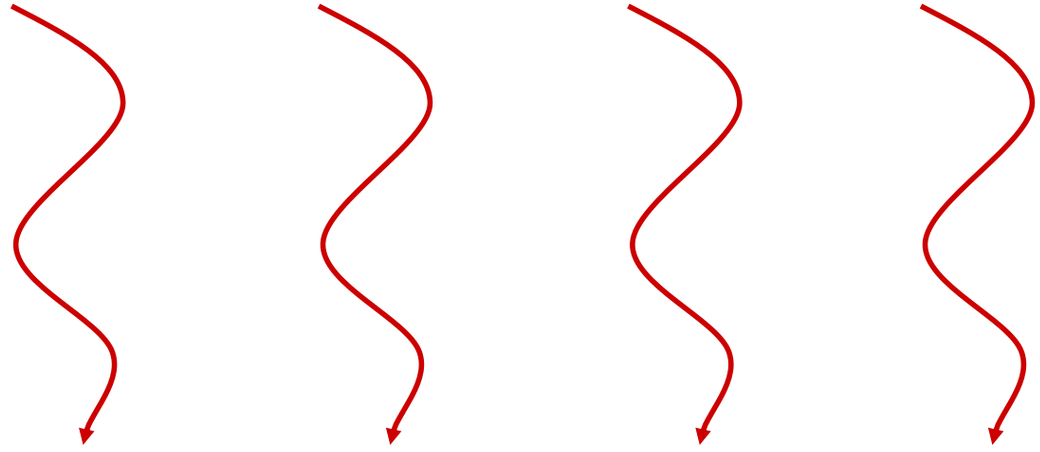
Time Needed at Minimum To Complete Task

A	**5**
B	**5**
C	**5**
D	**5**

Seems like we could just choose any; no preference.

Scheduling Threads

- Scheduler: It is responsible for scheduling threads
- From 4 threads, **which do we select to run**?



A
WIFI daemon

B
Bluetooth daemon

C
Discord

D
Spotify

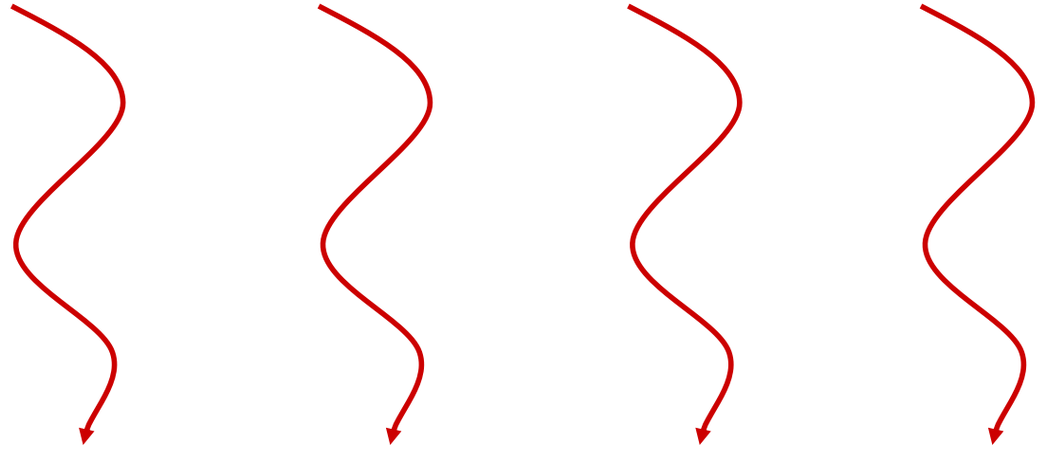
Time Needed at Minimum To Complete Task

A	**5**
B	**5**
C	**5**
D	**5**

Now, the question doesn't seem as straightforward.

Scheduling Threads

- Scheduler: It is responsible for scheduling threads
- From 4 threads, **how long do we select each thread to run**?



A
WIFI daemon

B
Bluetooth daemon

C
Discord

D
Spotify

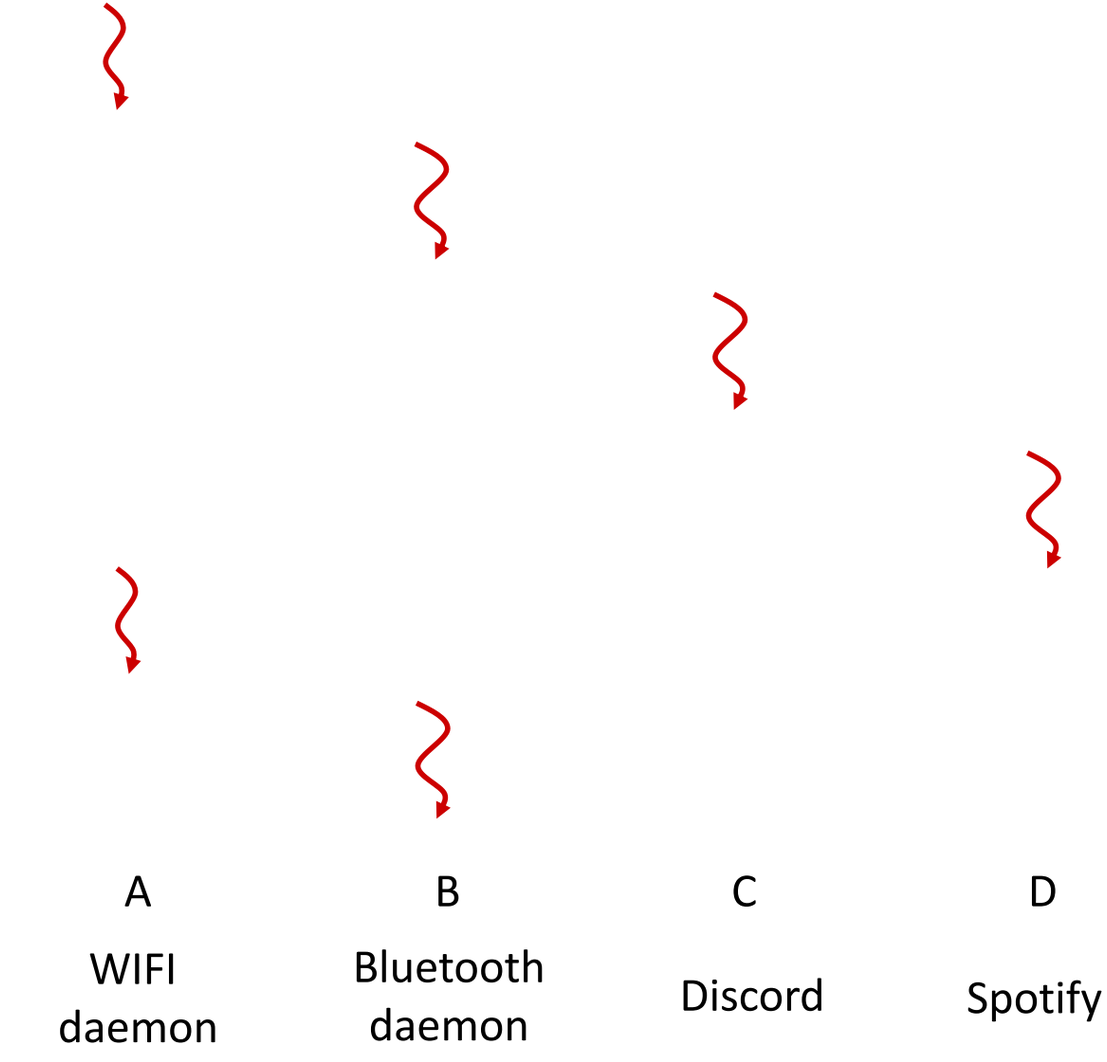
Time Needed at Minimum To Complete Task

A	**5**
B	**5**
C	**5**
D	**5**

We don’t need to run threads until they finish.

Scheduling Threads

- An Example: Smaller Unit of Time



Smaller Unit of Time:

- Allows for finer interleaving patterns of threads.
- Might result in quicker response times.
  - e.g. Need to handle network responses asap so that we don't lose packets….

Much is still missing from the conversation here….

Goals

- Not as straightforward, schedular needs to balance a number of things…
  - **Minimizing wait time**
    - Starting Threads as soon as possible.
  - **Minimizing latency**
    - Quick response times and task completions are preferred
  - **Maximizing throughput**
    - Do as much work as possible per unit of time
  - **Maximizing fairness**
    - Make sure every thread can execute fairly

- These goals depend on the system and can conflict with each other…

Scheduler Terminology

- The scheduler utilizes a scheduling algorithm to decide what runs next.

- Algorithms are designed to consider many factors:
  - Fairness: Every program gets to run
  - Liveness: That “something” will eventually happen
  - Throughput: amount of work completed over an interval of time
  - Wait time: Average time a “task” is “alive” but not running
  - Turnaround time: time between task being ready and completing
  - Response time: time it takes between task being ready and when it can take user input
  - Etc…

Scheduling: Other Considerations

- It takes time to context switch between threads
  - Could get more work done if thread switching is minimized

It takes ~1-2ns to context switch but really depends on machine.

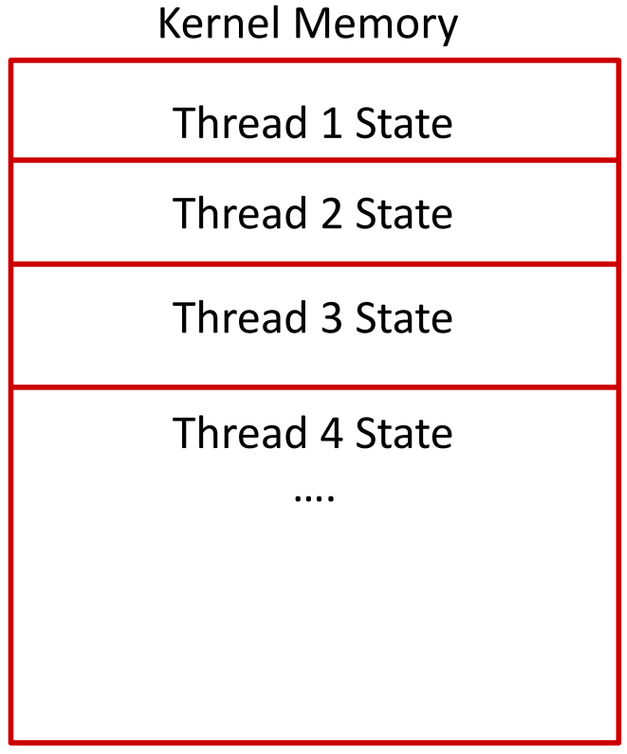
Save Thread A's registers to memory

Load Thread B's registers from memory

These are the General Purpose Registers in RISC-V, but this extends to all architectures….

Scheduling: Other Considerations

- Scheduling takes resources
  - It takes time to decide which thread to run next
  - It takes space to hold the required data structures



You don't want to ***waste*** valuable time looping through all Threads to find the most optimal; selecting the next thread should be a O(1) operation!

Scheduling: Other Considerations

- It takes time to context switch between threads
  - Could get more work done if thread switching is minimized
- Scheduling takes resources
  - It takes time to decide which thread to run next
  - It takes space to hold the required data structures
- Different tasks have different priorities
  - Higher priority tasks should finish first

Lecture Outline

- High Level: Scheduling
- Non-Preemptive Algorithms
  - First Come First Serve (FCFS)
  - Shortest Job First
- Preemptive Algorithms
  - Round Robin
  - PennOS Round Robin
  - Round Robin Variants
  - Completely Fair Scheduling (CFS)
- "Nice" Threads
- Extra: Earliest Eligible Virtual Deadline First, Interactive Tasks, Etc.

Types of Scheduling Algorithms

- **Non-Preemptive:** if a thread is running, it continues to run until it completes or until it gives up the CPU
  - First come first serve (FCFS)
  - Shortest Job First (SJF)

- Also called "Co-Operative" threading because well, the threads co-operate with each other…☺

First Come First Serve (FCFS)

- Idea: Whenever a thread is ready, schedule it to run until it is finished (or blocks).

- Maintain a queue of ready threads
  - Threads go to the back of the queue when it arrives or becomes unblocked
  - The thread at the front of the queue is the next to run

Example of FCFS

- Example workload with three “jobs”:

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

	Time to Finish
Job 1	**24**
Job 2	**3**
Job 3	**3**

- FCFS schedule:

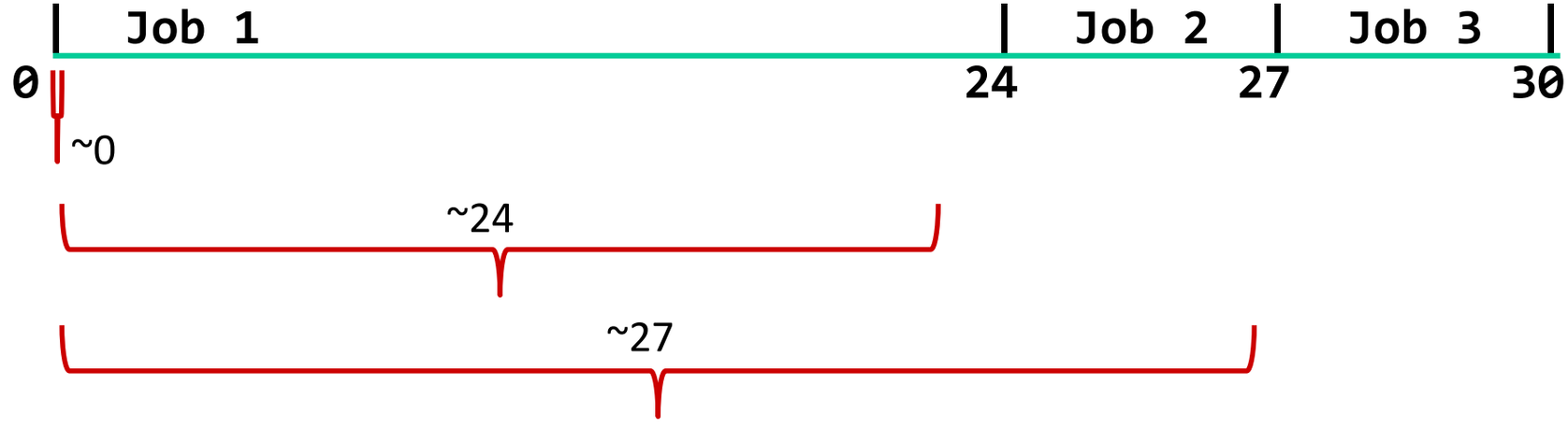
Example of FCFS

- Waiting Time: From Ready To Running

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

	Time to Finish
Job 1	**24**
Job 2	**3**
Job 3	**3**

- FCFS schedule:



Total waiting time: 0 + 24 + 27 = 51
Average waiting time: 51/3 = 17

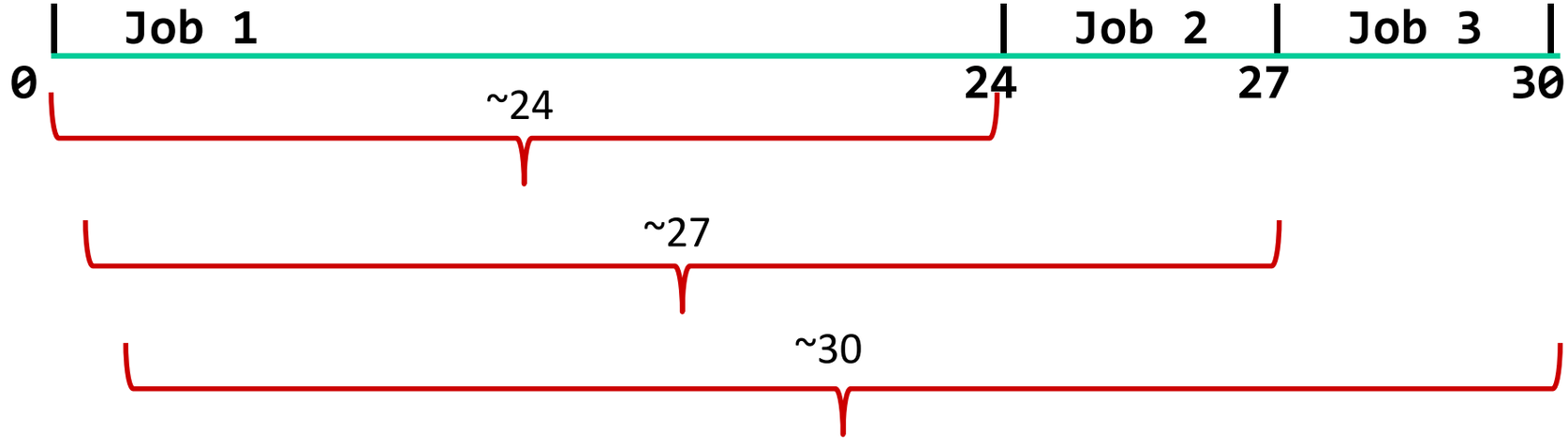
Example of FCFS

- Turnaround Time: From Arrival to Finished

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

	Time to Finish
Job 1	**24**
Job 2	**3**
Job 3	**3**

- FCFS schedule:



Total waiting time: 24 + 27 + 30 = 81

Average waiting time: 81/3 = 27

Poll Everywhere

pollev.com/cis5480

- What are the advantages/disadvantages/concerns with **First Come First Serve**

- Things a scheduler should prioritize:
  - Minimizing Waitime
  - Minimizing Latency
  - Maximizing fairness
  - Maximizing throughput
  - Task priority
  - Cost to schedule things
  - Cost to context Switch

- Imagine we have 1 core, and tasks of various (finite) lengths…

Shortest Job First (SJF)

- Idea: variation on FCFS, but have the tasks with the smallest CPU-time requirement run first
  - Arriving jobs are instead put into the queue depending on their run time, shorter jobs being towards the front
  - Scheduler selects the shortest job (1st in queue) and runs till completion

Example of SJF

- Example workload with three “jobs”:

1 CPU
Job 1, 2, & 3 arrive at the same time.

	Time to Finish
Job 1	**24**
Job 2	**3**
Job 3	**3**

- SJF schedule:

- Total waiting time: 0 + 3 + 6= 9
- Average waiting time: 3
- Total turnaround time: 3 + 6 + 30 = 39
- Average turnaround time: 39/3 = 13

Poll Everywhere

pollev.com/cis5480

- What are the advantages/disadvantages/concerns with **Shortest Job First**

- Things a scheduler should prioritize:
  - Minimizing wait time
  - Minimizing Latency
  - Maximizing fairness
  - Maximizing throughput
  - Task priority
  - Cost to schedule things
  - Cost to context Switch

- Imagine we have 1 core, and tasks of various (finite) lengths …

Lecture Outline

- High Level: Scheduling
- Non-Preemptive Algorithms
  - First Come First Serve (FCFS)
  - Shortest Job First
- Preemptive Algorithms
  - Round Robin
  - PennOS Round Robin
  - Round Robin Variants
  - Completely Fair Scheduling (CFS)
- "Nice" Threads
- Extra: Earliest Eligible Virtual Deadline First, Interactive Tasks, Etc.

Types of Scheduling Algorithms

- **Preemptive:** the thread may be ***interrupted*** after a given time and/or if another thread becomes ready
  - Round Robin
  - Priority Round Robin
  - …

Round Robin

- Sort of a preemptive version of FCFS
  - Whenever a thread is ready, add it to the end of the queue.
  - Run whatever job is at the front of the queue

- BUT only led it run for a fixed amount of time (quantum).
  - If it finishes before the time is up, schedule another thread to run
  - If time is up, then send the running thread back to the end of the queue.

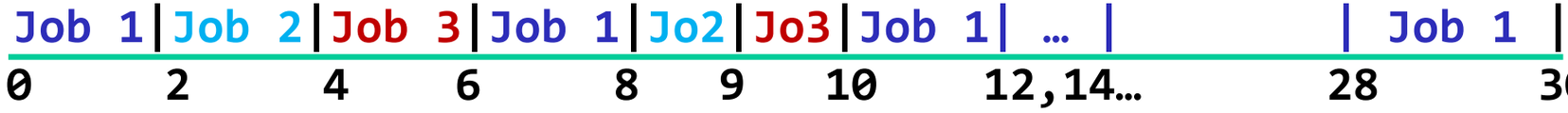
Example of Round Robin

- Same Example workload with three “jobs”:

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

	Time to Finish
Job 1	**24**
Job 2	**3**
Job 3	**3**

- FCFS schedule:



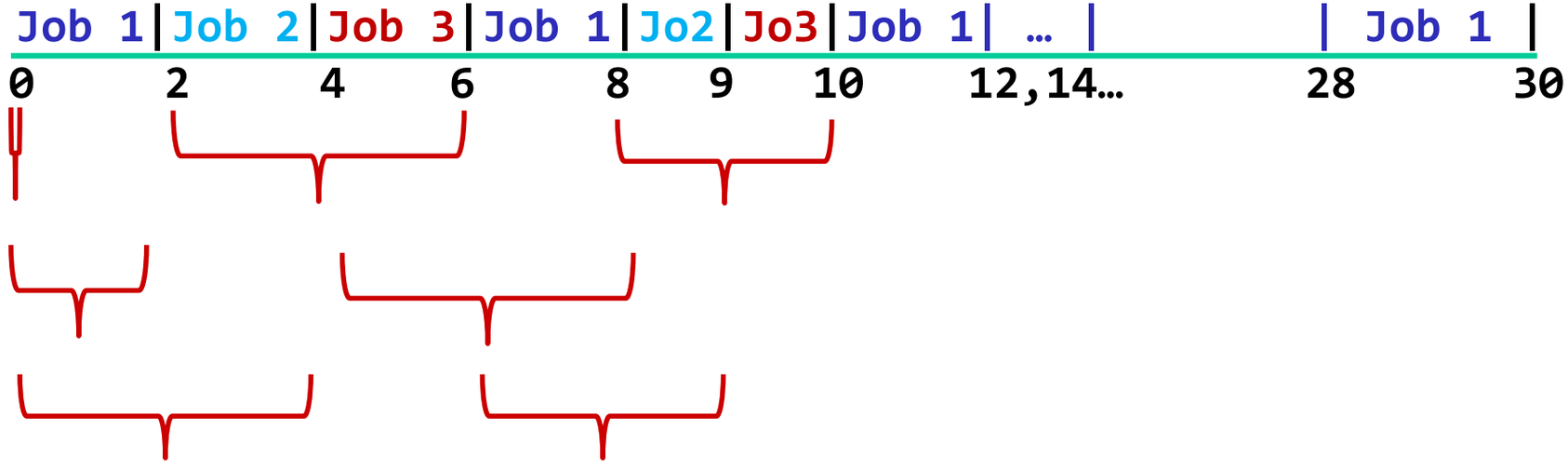
Example of Round Robin

- Waiting Time: From Ready To Running

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

	Time to Finish
Job 1	**24**
Job 2	**3**
Job 3	**3**

- FCFS schedule:



Total waiting time: : (0 + 4 + 2) + (2 + 4) + (4 + 3) = 19
Average waiting time: 19/3 (~6.33)

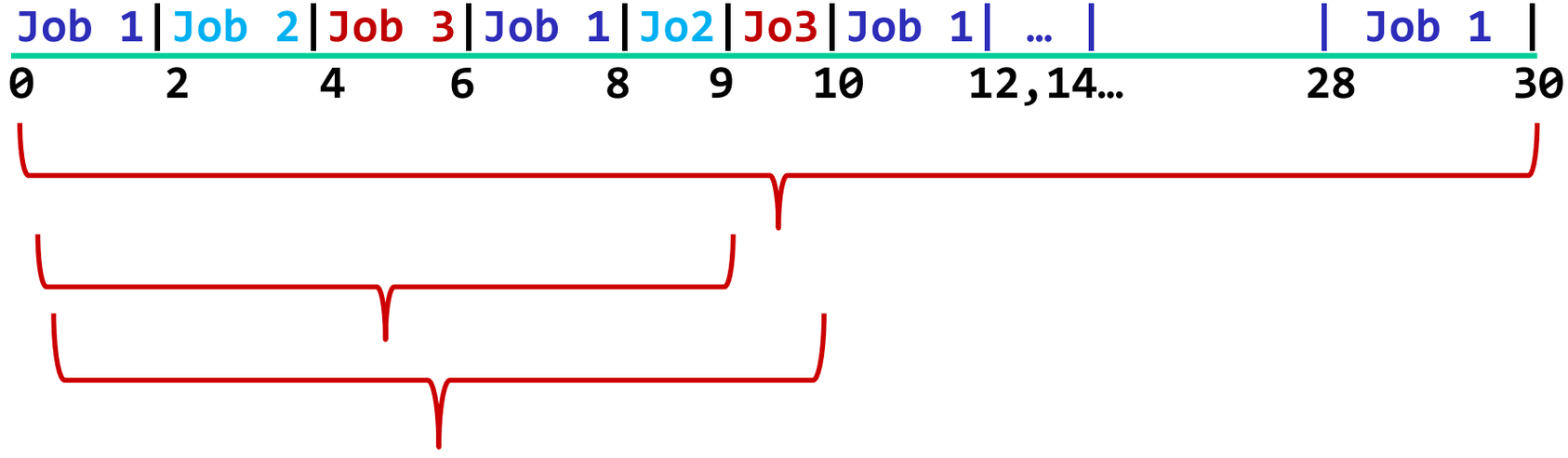
Example of Round Robin

- Turnaround Time: From Arrival to Finished

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

	Time to Finish
Job 1	**24**
Job 2	**3**
Job 3	**3**

- FCFS schedule:



Total turnaround time: 30 + 9 + 10 = 49
Average turnaround time: 49/3 (~16.33)

Poll Everywhere

pollev.com/cis5480

- What are the advantages/disadvantages/concerns with **Round Robin**

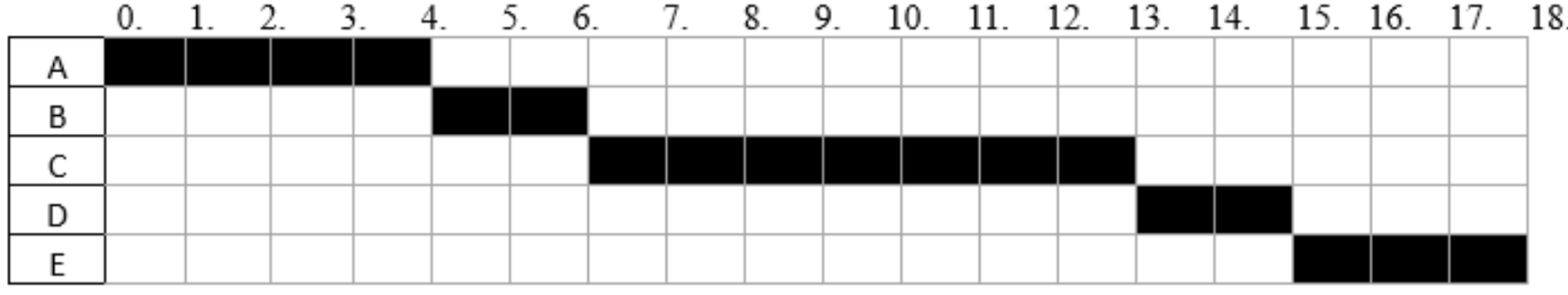
- Things a scheduler should prioritize:
  - Minimizing wait time
  - Minimizing Latency
  - Maximizing fairness
  - Maximizing throughput
  - Task priority
  - Cost to schedule things
  - Cost to context Switch

- Imagine we have 1 core, and tasks of various lengths…

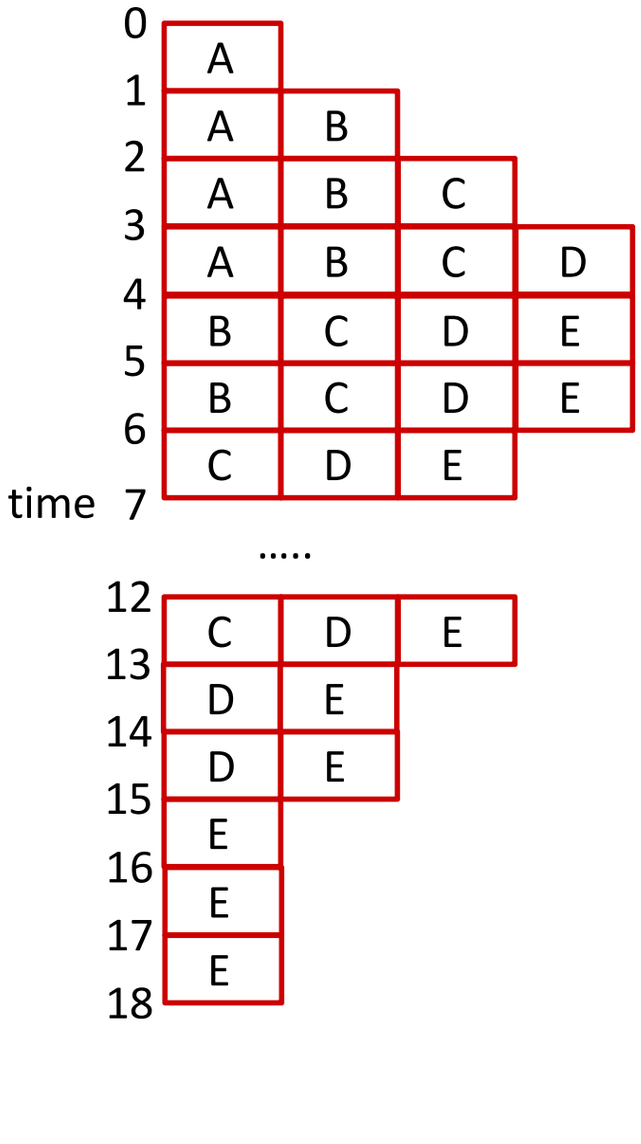
FCFS Example!

- Assume that we had the following set of processes that ran on a single CPU following the First Come First Serve (FCFS) scheduling policy.
- If we expressed this with a drawing, where a black square represents that the process is executing, we would get:

Process	Arrival Time	Finishing Time
A	0	4
B	1	6
C	2	13
D	3	15
E	4	18



FCFS Example: Different Visualization



Process	Arrival Time	Finishing Time
A	0	4
B	1	6
C	2	13
D	3	15
E	4	18

= 1 unit of time

Round Robin Practice!

- Instead, lets switch our algorithm to round-robin with a time quantum of 3 time units.

Process	Arrival Time	Finishing Time
A	0	4
B	1	6
C	2	13
D	3	15
E	4	18

- You can assume:
  - Context switching and running the Scheduler are instantaneous.
  - If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.	17.	18.
A																			
B																			
C																			
D																			
E																			

Round Robin Practice!

- Instead, lets switch our algorithm to round-robin with a time quantum of 3 time units.

Process	Arrival Time	Finishing Time
A	0	4
B	1	6
C	2	13
D	3	15
E	4	18

Process	Arrival Time	Length
A	0	4
B	1	2
C	2	7
D	3	2
E	4	3

Useful to convert to this style of table.

- You can assume:
  - Context switching and running the Scheduler are instantaneous.
  - If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.	17.	18.
A																			
B																			
C																			
D																			
E																			

RR Variant: PennOS Scheduler

- In PennOS you will have to implement a priority scheduler based mostly off of round robin.

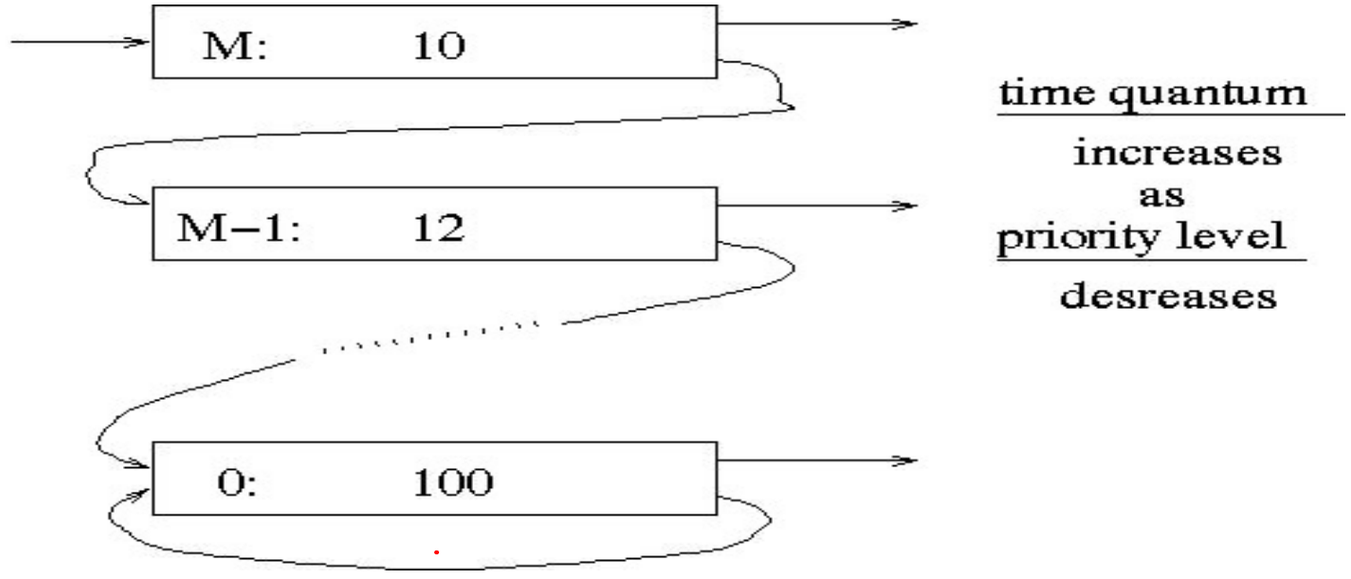
- You will have 3 queues, each with a different priority (0, 1, 2)
  - Each queue acts like normal round robin within the queue

- You spend time quantum processing each queue proportional to the priority
  - Priority 0 is scheduled 1.5 times more often than priority 1
  - Priority 1 is scheduled 1.5 times more often than priority 2

RR Variant: Priority Round Robin

- Same idea as round robin, but with multiple queues for different priority levels.
- Scheduler chooses the first item in the highest priority queue to run
- Scheduler only schedules items in lower priorities if all queues with higher priority are empty.

RR Variant: Multi Level Feedback



- Each priority level has a ready queue, and a time quantum
- Thread enters highest priority queue initially
  - Move to lower queue with each timer interrupt (e.g. it was pre-empted by the scheduler)
- If a thread 'voluntarily' stops using CPU before time is up, it is moved to the end of the current queue
- Bottom queue is standard Round Robin
- Thread in a given queue not scheduled until all higher queues are empty

Multi Level Feedback Analysis

- Threads with high I/O bursts are preferred
  - Makes higher utilization of the I/O devices
  - Good for interactive programs (keyboard, terminal, mouse is I/O)
- Threads that need the CPU a lot will sink to lower priority, giving shorter threads a chance to run
- Still have to be careful in choosing time quantum
- Also have to be careful in choosing how many layers

Multi Level Feedback Variants: Priority

- Can assign tasks different priority levels upon initiation that decide which queue it starts in
  - E.g. the bluetooth daemon should have higher priority than HelloWorld.java
- Update the priority based on recent CPU usage rather than overall cpu usage of a task
  - Makes sure that priority is consistent with recent behavior
- Many others that vary from system to system

Multiple Cores

- On a modern machine, we have multiple CPU Cores, each can run tasks
  - ***Generally*** each core has its own run-queue
  - It helps to keep threads in the same process on the same processor
  - Threads in the same process use the same memory: lower overhead
    - If we want to there are ways to make sure a thread/process is "pinned" to a CPU
      - See: Thread Affinity / Processor Affinity / CPU Pinning

- There is other stuff to balance tasks across cores, but I am leaving that out for time ☺

"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

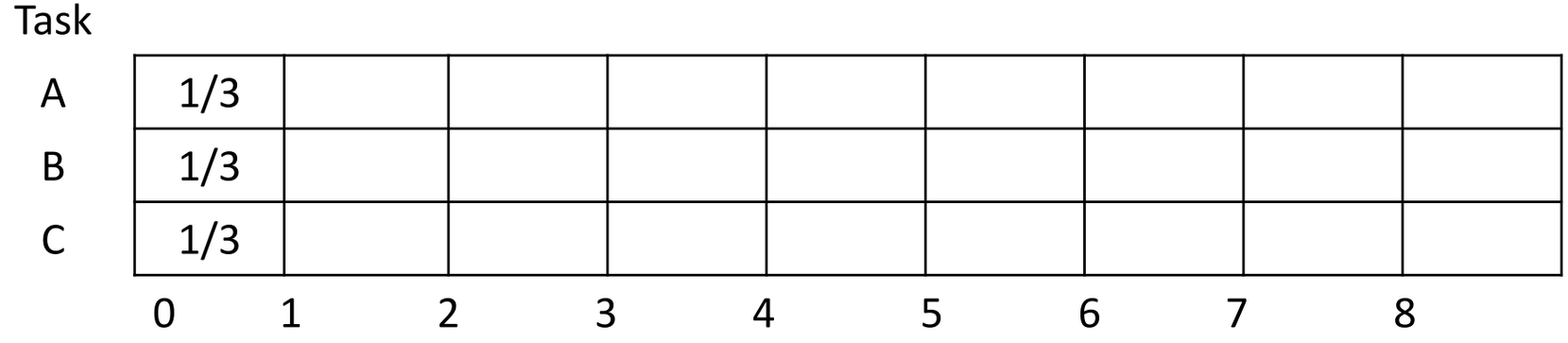
"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2



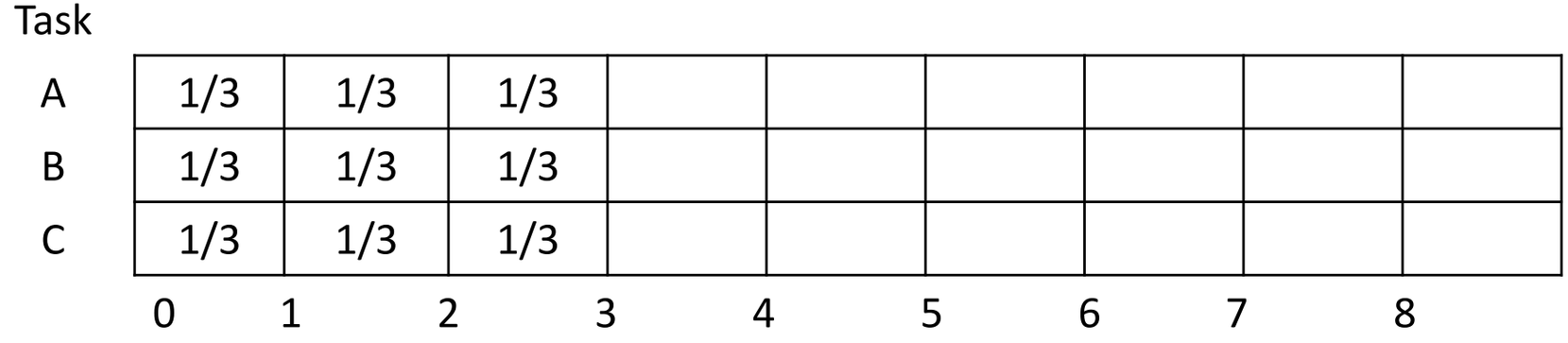
"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2



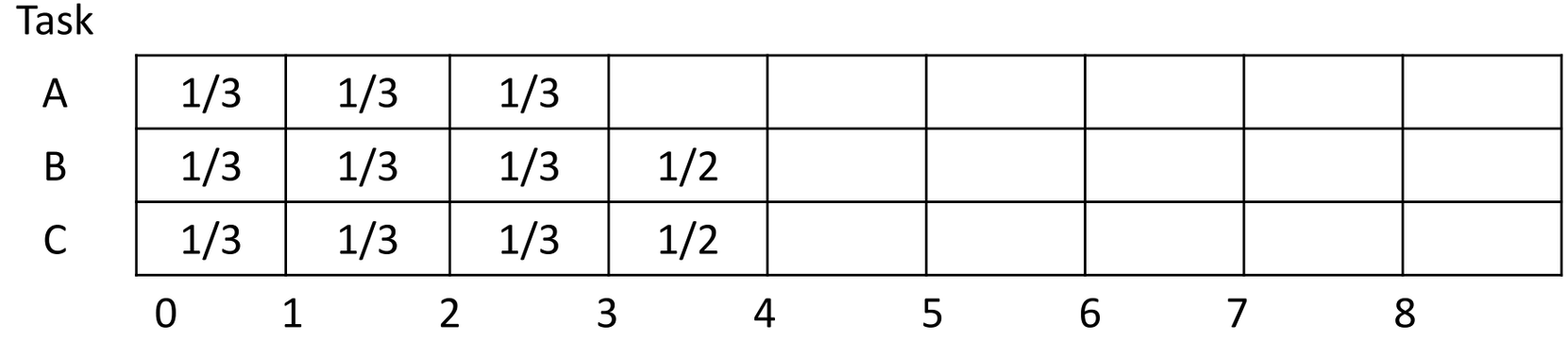
"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2



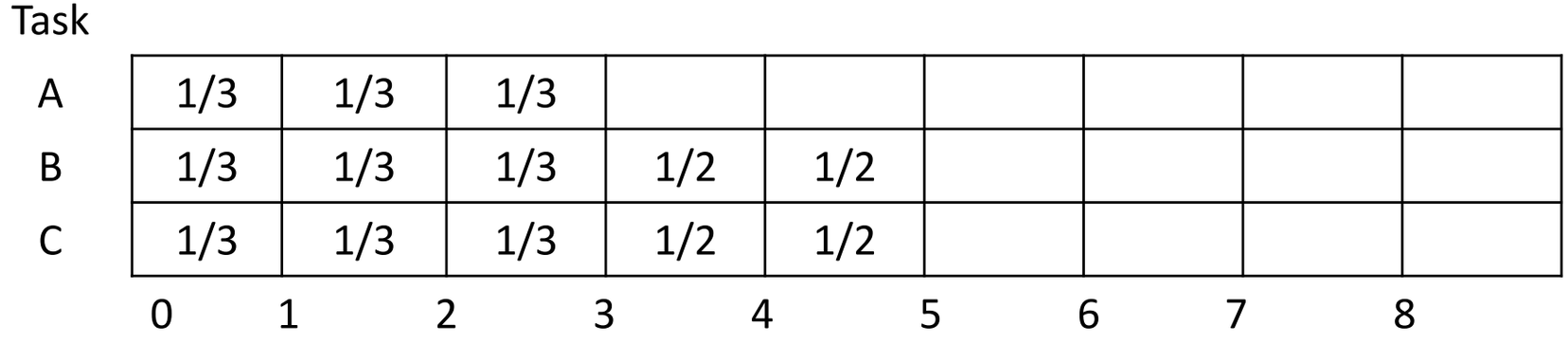
"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2



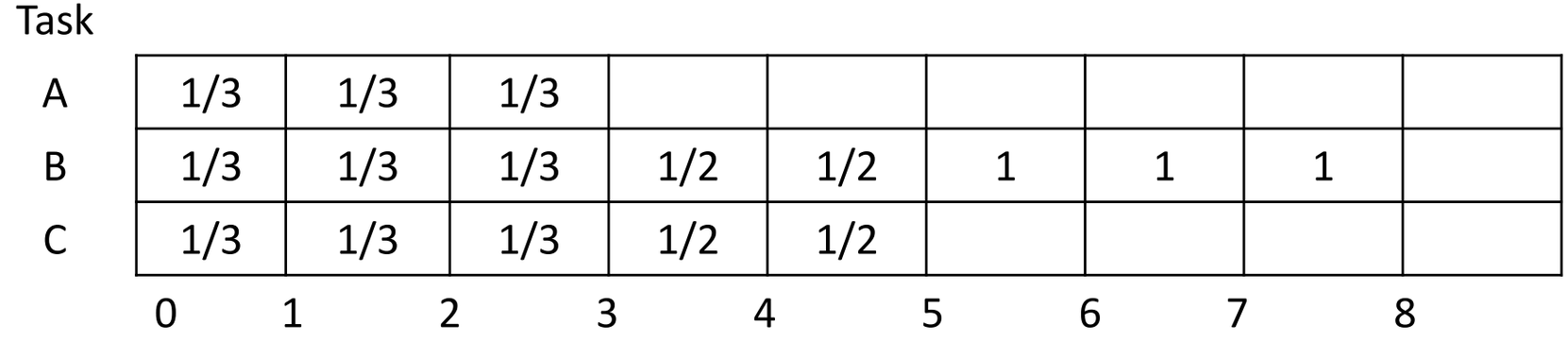
"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2



CFS – Reality

- In reality there are things that prevent us from having a “perfect multi-tasking processor”
  - Time to context switch
  - Time for the scheduler run
  - Time spent running other things in the kernel that don't really belong to a single task
  - **Task may not be pre-emptible sometimes and we need to wait for the task to become pre-emptible.**
  - Etc.

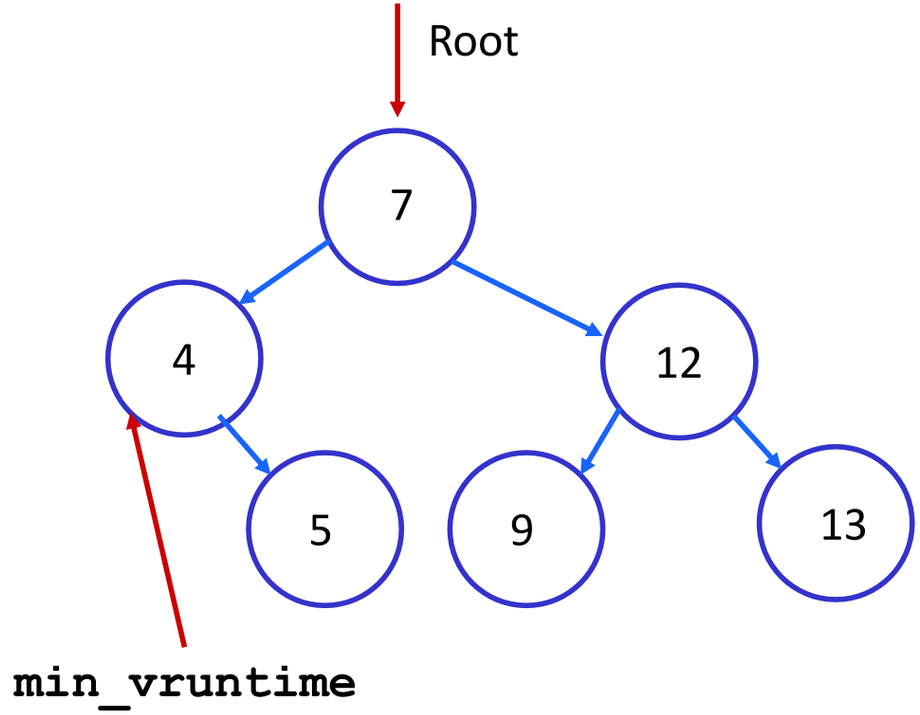
CFS – Implementation

- CFS maintains a current count for "how long has a task run" called `vruntime`.
- The runtimes of all tasks are stored by the scheduler
- Unlike round robin, a thread is not run for a fixed amount of time
  - Run a task till there is some thing with a lower `vruntime`
  - To avoid constantly switching back and forth between two tasks there is a minimum "granularity" (~2.25 milliseconds iirc)

CFS – Implementation Details

- CFS maintains a current count for "how long has a task run" called `vruntime`.

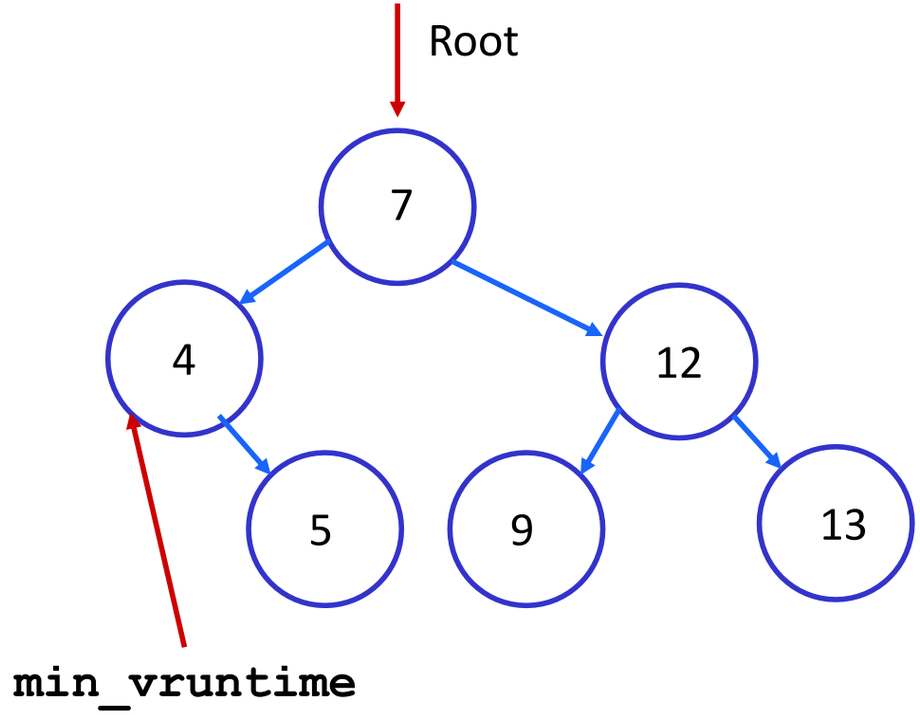
- The runtimes of all tasks are stored by the scheduler inside of a Red-Black Tree
  - Red-Black Tree is a Self balancing binary tree
  - Sorted on the vruntime for each task
  - Smallest vruntime task is the leftmost node



- Adding a node is O(log N) operation
- Pointer to leftmost node is maintained, so looking up is O(1)

CFS – Implementation Details

- CFS maintains a current count for "how long has a task run" called `vruntime`.
- On each scheduler "tick" the processor compares the current running task to the leftmost task
- If the min_vruntime is less than the current node (and granularity has passed) then start running the minimum task.



CFS – New Tasks

- New tasks haven't run on the CPU, so their **vruntime** is 0 when they are created?
  - No, instead new tasks start with their vruntime equal to the min_vruntime.
  - This way fairness is maintained between newer and older tasks.

CFS – I/O Bound Tasks

- CFS will also maintain whether a job is sleeping or blocked. Won't schedule to run those tasks and store them in a separate structure.

- CFS handles I/O bound tasks pretty well :)

- Tasks with many I/O bursts will have small usage of CPU.
  So they also have a low vruntime and have higher priority.

nice

- nice

nice

- Linux has a way to set priority with a `nice` value.
  - Each process starts with a nice value of 0
  - Nice is clamped to [-20, 19]

- The higher your nice score, the “nicer” you are
  (the task runs less often thus letting other tasks run instead of it)

- Higher nice score -> lower priority
- Lower nice score -> higher priority

CFS – Vruntime

- CFS uses vruntime as the dominant metric
  - V stands for virtual (e.g. not real runtime)

- You may have thought:
  - `curr_task->runtime += time_running`
  - This is false

- vruntime takes other things (like nice scores) into consideration
  - `curr_task->vruntime += (time_running * weight_based_on_nice)`

- CFS takes other things into consideration that make it more complex :)

Earliest Eligible Virtual Deadline First (EEVDF)

- New Linux scheduler!
  - Replaced CFS less than a year ago (April 2024)
  - Still aims for fairness, just with some different metrics

- Utilizes a new concept called “lag” (in addition to vruntime)
  - A measurement for how much time a task is “owed” if it did not get its fair share of time
  - Tasks that took more CPU time than its fair share have negative “lag”
    - Will not be considered “Eligible”. will not be run until lag >= 0
    - Sleeping / blocked tasks will not get free lag increases

Earliest Eligible Virtual Deadline First (EEVDF)

- Some Lore:

Wrote the CFS Schedular

```
From: Peter Zijlstra <peterz@infradead.org>
To: mingo@kernel.org, vincent.guittot@linaro.org
Cc: linux-kernel@vger.kernel.org, peterz@infradead.org,
        juri.lelli@redhat.com, dietmar.eggemann@arm.com,
        rostedt@goodmis.org, bsegall@google.com, mgorman@suse.de,
        bristot@redhat.com, corbet@lwn.net, qyousef@layalina.io,
        chris.hyser@oracle.com, patrick.bellasi@matbug.net,
        pjt@google.com, pavel@ucw.cz, qperret@google.com,
        tim.c.chen@linux.intel.com, joshdon@google.com, timj@gnu.org,
        kprateek.nayak@amd.com, yu.c.chen@intel.com,
        youssefesmat@chromium.org, joel@joelfernandes.org, efault@gmx.de,
        tglx@linutronix.de
Subject: [PATCH 08/15] sched: Commit to EEVDF
Date: Wed, 31 May 2023 13:58:47 +0200   [thread overview]
Message-ID: <20230531124604.137187212@infradead.org> (raw)
In-Reply-To: 20230531115839.089944915@infradead.org

EEVDF is a better defined scheduling policy, as a result it has less
heuristics/tunables. There is no compelling reason to keep CFS around.

Signed-off-by: Peter Zijlstra (Intel) <peterz@infradead.org>
```

Earliest Eligible Virtual Deadline First (EEVDF)

- Some Lore:

```
From: Ingo Molnar <mingo@kernel.org>
To: Joel Fernandes <joel@joelfernandes.org>
Cc: Peter Zijlstra <peterz@infradead.org>,
        vincent.guittot@linaro.org, linux-kernel@vger.kernel.org,
        juri.lelli@redhat.com, dietmar.eggemann@arm.com,
        rostedt@goodmis.org, bsegall@google.com, mgorman@suse.de,
        bristot@redhat.com, corbet@lwn.net, qyousef@layalina.io,
        chris.hyser@oracle.com, patrick.bellasi@matbug.net,
        pjt@google.com, pavel@ucw.cz, qperret@google.com,
        tim.c.chen@linux.intel.com, joshdon@google.com, timj@gnu.org,
        kprateek.nayak@amd.com, yu.c.chen@intel.com,
        youssefesmat@chromium.org, efault@gmx.de, tglx@linutronix.de
Subject: Re: [PATCH 08/15] sched: Commit to EEVDF
Date: Thu, 22 Jun 2023 14:01:07 +0200   [thread overview]
Message-ID: <ZJQ4A2Jm4VoGMKbl@gmail.com> (raw)
In-Reply-To: <20230616212353.GA628850@google.com>


* Joel Fernandes <joel@joelfernandes.org> wrote:

> On Wed, May 31, 2023 at 01:58:47PM +0200, Peter Zijlstra wrote:
> > EEVDF is a better defined scheduling policy, as a result it has less
> > heuristics/tunables. There is no compelling reason to keep CFS around.
> >
> > Signed-off-by: Peter Zijlstra (Intel) <peterz@infradead.org>
> > ---
> >  kernel/sched/debug.c    |    6
> >  kernel/sched/fair.c     |  465 +++---------------------------------------------
>
> Whether EEVDF helps us improve our CFS latency issues or not, I do like the
> merits of this diffstat alone and the lesser complexity and getting rid of
> those horrible knobs is kinda nice.

To to be fair, the "removal" in this patch is in significant part an
artifact of the patch series itself, because first EEVDF bits get added by
three earlier patches, in parallel to CFS:

 kernel/sched/fair.c     |  137 ++++++++++++++++++++++++++++++++++++++++++++++++++++--
 kernel/sched/fair.c     |  162 +++++++++++++++++++++++++++++++++++++++-----------
 kernel/sched/fair.c     |  338 ++++++++++++++++++++++++++++++++++++++++++++-------

... and then we remove the old CFS policy code in this 'commit to EEVDF' patch:

 kernel/sched/fair.c     |  465 +++--------------------------------------------

The combined diffstat is close to 50% / 50% balanced:

 kernel/sched/fair.c             | 1105 ++++++++++++++++++--------------------------

But having said that, I do agree that EEVDF as submitted by Peter is better
defined, with fewer heuristics, which is an overall win - so no complaints
from me!

Thanks,

        Ingo
```

Wrote the CFS Schedular

Earliest Eligible Virtual Deadline First (EEVDF)

- Not going over it due to:
  - Time in lecture, looks like it may be more complex and take longer to explain
  - It is new! Not as much information out there on it
    - I could read the Linux kernel source code, but that takes time :)))))))
    - I invite you to read the 13718 lines of the schedular implementation. ☺

- Take a look at these articles from LWN.net if you want to learn more about EEVDF
  - https://lwn.net/Articles/925371/
  - https://lwn.net/Articles/969062/

Why did we talk about this?

- Scheduling is fundamental towards how computer can multi-task
- This is a great example of how “systems” intersects with algorithms :)
- It shows up ‘occasionally’ in the real world :)

What really happened on Mars Rover Pathfinder, Mike Jones.
http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

More

- For those curious, there was a LOT left out

- RTOS (Real Time Operating Systems)
  - For real time applications
  - CRITICAL that data and events meet defined time constraints
  - Different focus in scheduling. Throughput is de-prioritized

- Fair-share scheduling
  - Equal distribution across different users instead of by processes

- Etc.

FCFS Analysis

- Advantages:
  - Simple, low overhead
  - Hard to screw up the implementation lol
  - Each thread will DEFINITELY get to run eventually.

- Disadvantages
  - Doesn’t work well for interactive systems
  - Throughput can be low due to long threads
  - Large fluctuations in average turn around time
  - Priority not taken into considerations

SJF Analysis

- Advantages:
  - Still relatively simple, low overhead
  - perhaps minimal average turnaround time

- Disadvantages
  - Starvation possible
    - If quick jobs keep arriving, long jobs will keep being pushed back and won't execute
  - How do you know how long it takes for something to run?
    - You CAN'T. You can use a history of past behavior to make a guess.
  - Priority not taken into considerations

Round Robin Analysis

- Advantages:
  - Still relatively simple
  - Can work for interactive systems

- Disadvantages
  - If quantum is too small, can spend a lot of time context switching
  - If quantum is too large, approaches FCFS
  - Still assumes all processes have the same priority.

- Rule of thumb:
  - Choose a unit of time so that most jobs (80-90%) finish in one usage of CPU time

Round Robin Practice!

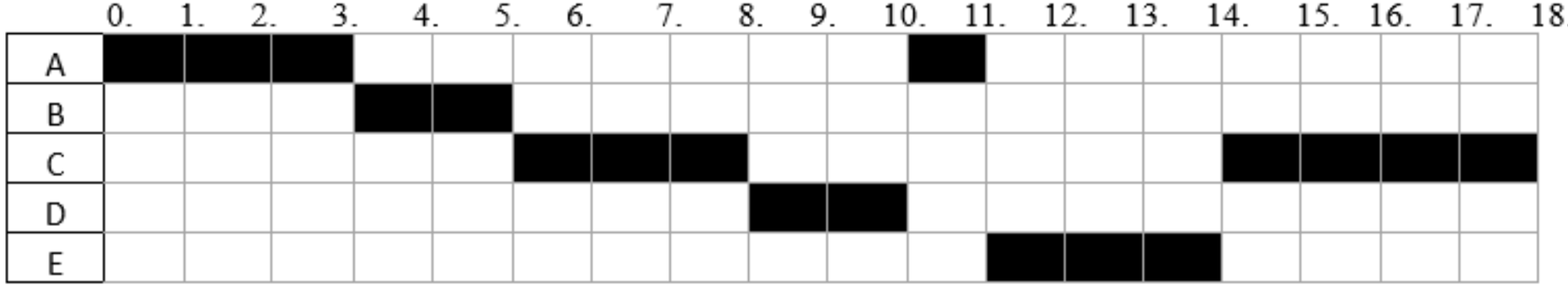
- Instead, lets switch our algorithm to round-robin with a time quantum of 3 time units.

Process	Arrival Time	Finishing Time
A	0	4
B	1	6
C	2	13
D	3	15
E	4	18

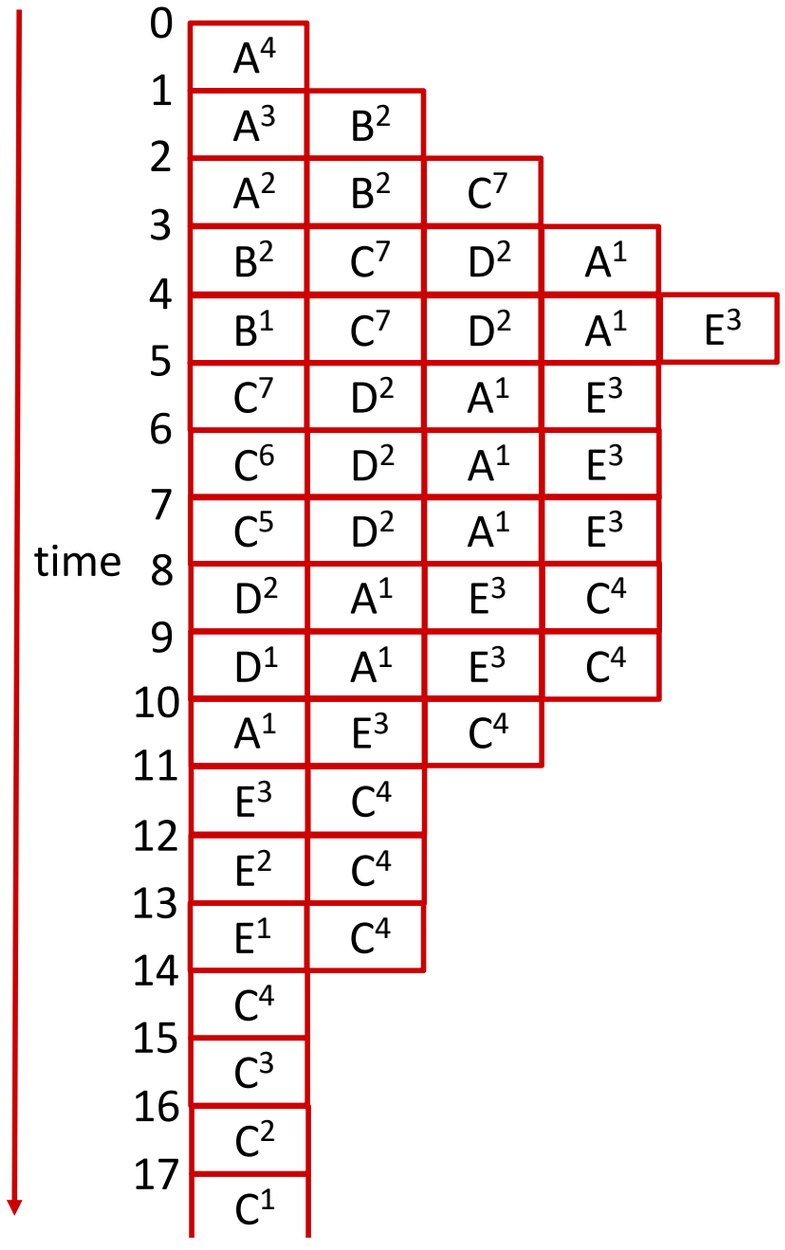
→

Process	Arrival Time	Length
A	0	4
B	1	2
C	2	7
D	3	2
E	4	3

- You can assume:
  - Context switching and running the Scheduler are instantaneous.
  - If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.



RR Example: Different Visualization

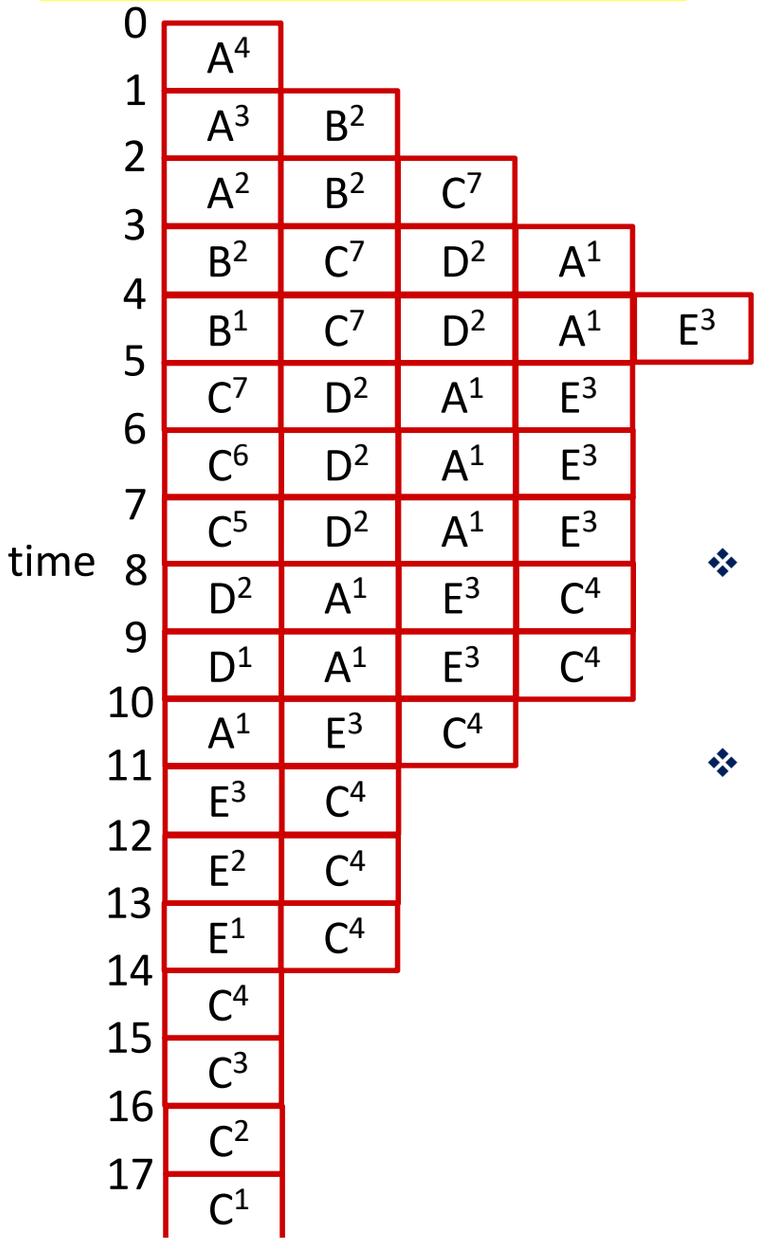


Process	Arrival Time	Length
A	0	4
B	1	2
C	2	7
D	3	2
E	4	3

= 1 unit of time

Poll Everywhere

pollev.com/cis5480



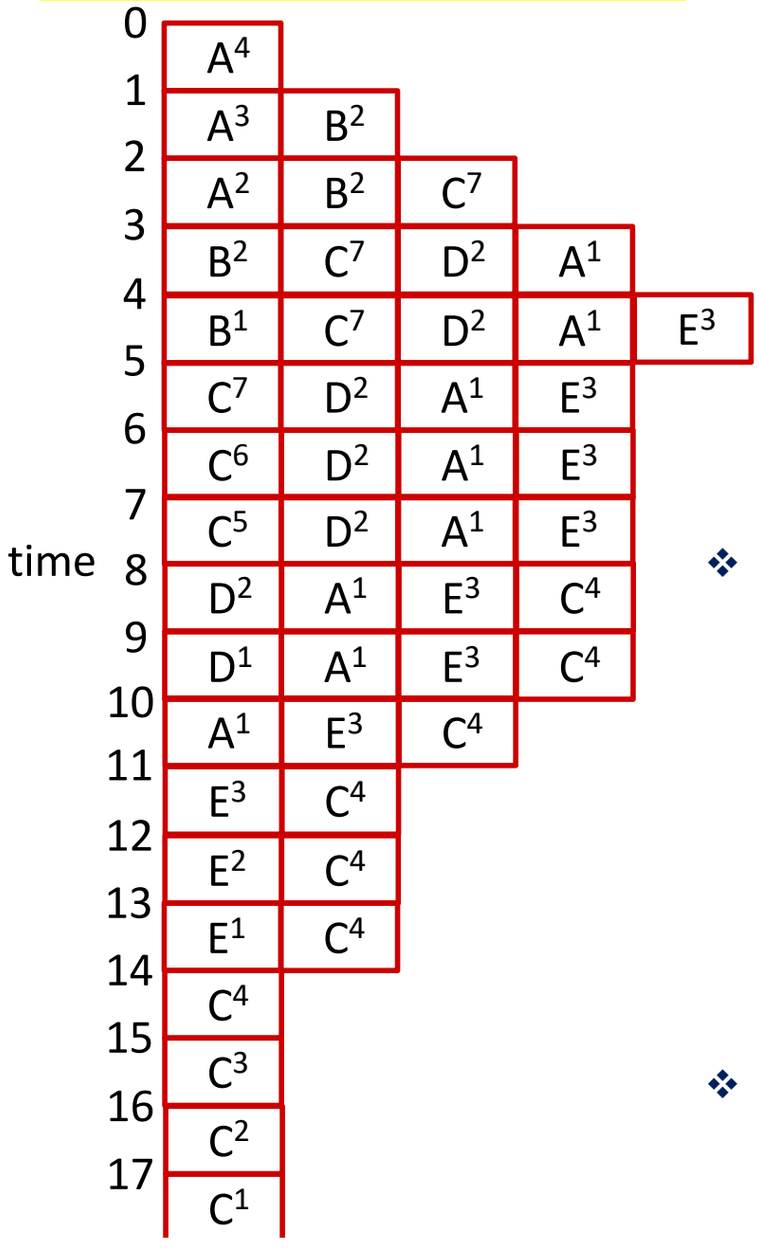
Process	Arrival Time	Length
A	0	4
B	1	2
C	2	7
D	3	2
E	4	3

- What is the average wait time?
- What is the average turnaround time?

= 1 unit of time

Poll Everywhere

pollev.com/cis5480



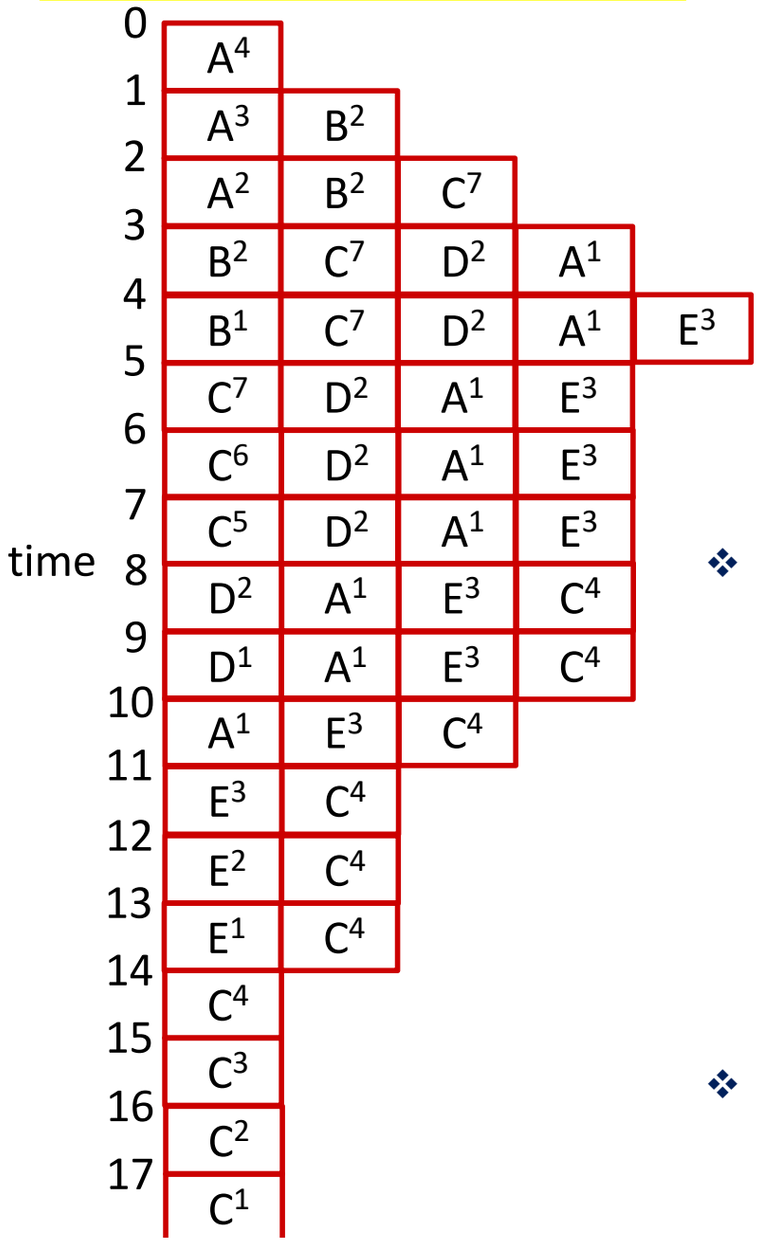
Process	Arrival Time	Length
A	0	4
B	1	2
C	2	7
D	3	2
E	4	3

- What is the average wait time?
  - A -> (0 + 7)
  - B -> (3)
  - C -> (3 + 6)
  - D -> (5)
  - E -> (7)
- (7 + 3 + 9 + 5 + 7)/5 = 6.2

= 1 unit of time

Poll Everywhere

pollev.com/cis5480



Process	Arrival Time	Length
A	0	4
B	1	2
C	2	7
D	3	2
E	4	3

- What is the average turnaround time?
  - A -> (11)
  - B -> (4)
  - C -> (16)
  - D -> (7)
  - E -> (10)
- (11 + 4 + 16 + 7 + 10)/5 =9.6

= 1 unit of time

Round Robin vs. FCFS

- What is the average wait time?
  - (7 + 3 + 9 + 5 + 7)/5 = 6.2
- What is the average turnaround time?
  - (11 + 4 + 16 + 7 + 10)/5 =9.6

- What is the average wait time?
  - (0 + 3 + 4 + 10 + 11)/5 = 5.6
- What is the average turnaround time?
  - (4 + 5 + 12 + 12 + 14)/5 =9.4

In this small example FCFS does better than Round Robin! But does it really?

I invite you to think about what we could change this to make it much worse for FCFS.

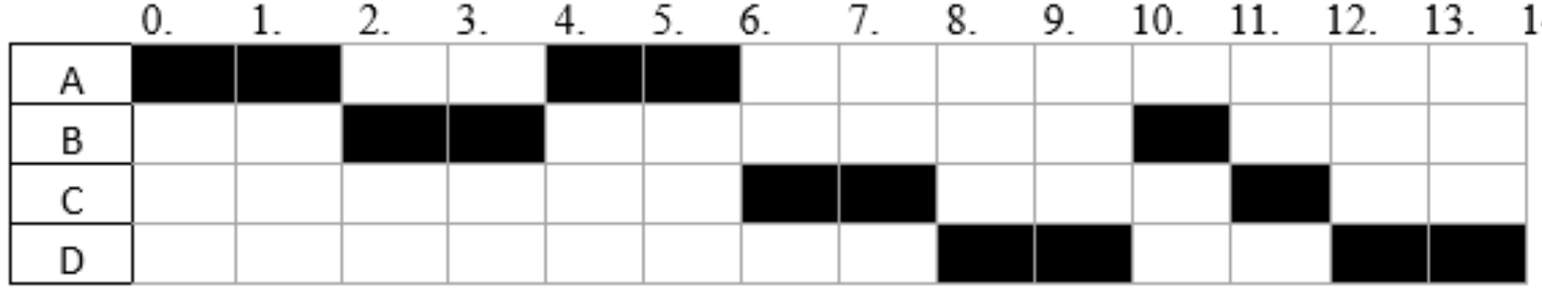
Consideration: Interactive Tasks

- There is still ongoing work to make schedulers that are better
  - "Better" either in general or to specific situations

- Example: People are already working on EEVDF upgrades. VARD scheduler for SteamOS
  https://youtu.be/xJjZ5tzlHOY?si=lgGNWaQe03qSgCP2&t=1682

More Round Robin Practice

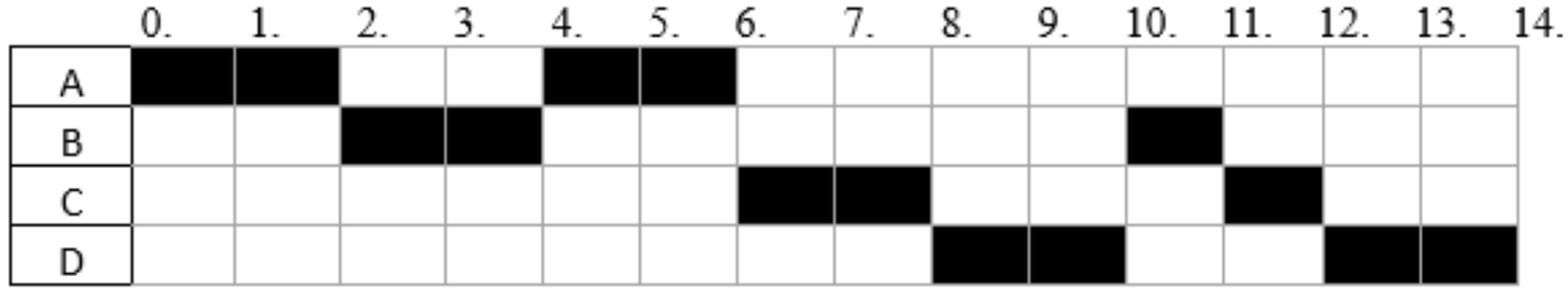
- Four processes are executing on one CPU following round robin scheduling:



- You can assume:
  - All processes do not block for I/O or any resource.
  - Context switching and running the Scheduler are instantaneous.
  - If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

Solutions at end of slide deck

More Round Robin Practice

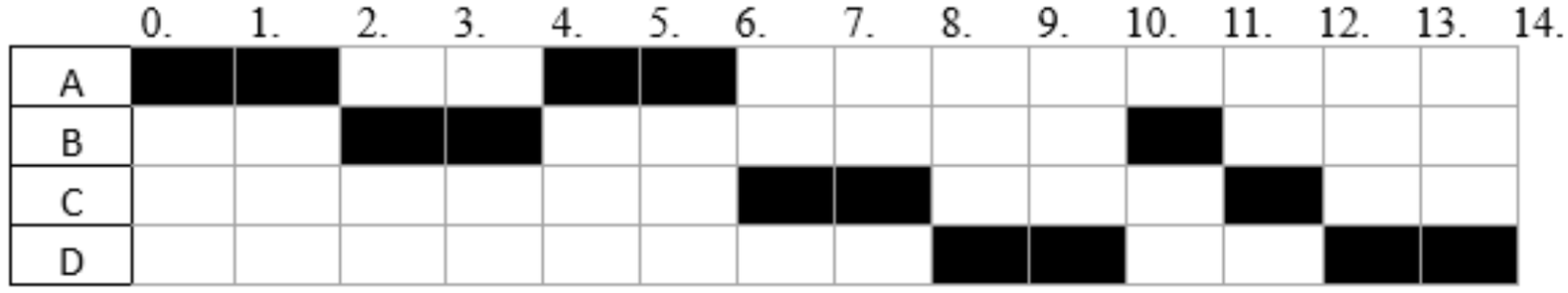


- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

- What is the earliest time that process C could have arrived?
- Which processes are in the ready queue at time 9?
- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?

Solutions at end of slide deck

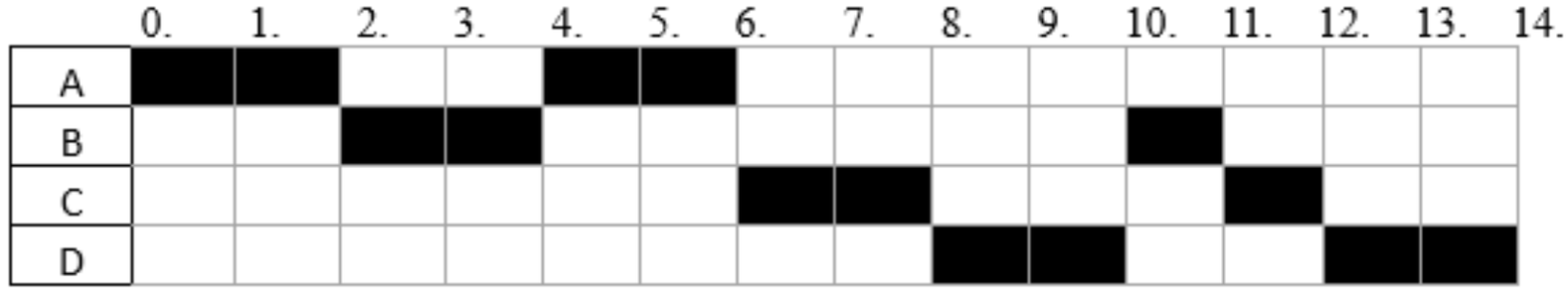
More Round Robin Practice



- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

- What is the earliest time that process C could have arrived?
  - If C arrived at time 0, 1, or 2, it would have run at time 4
  - C could have shown up at time 3 and come after A in the queue
  - C showed up at time 3 at earliest

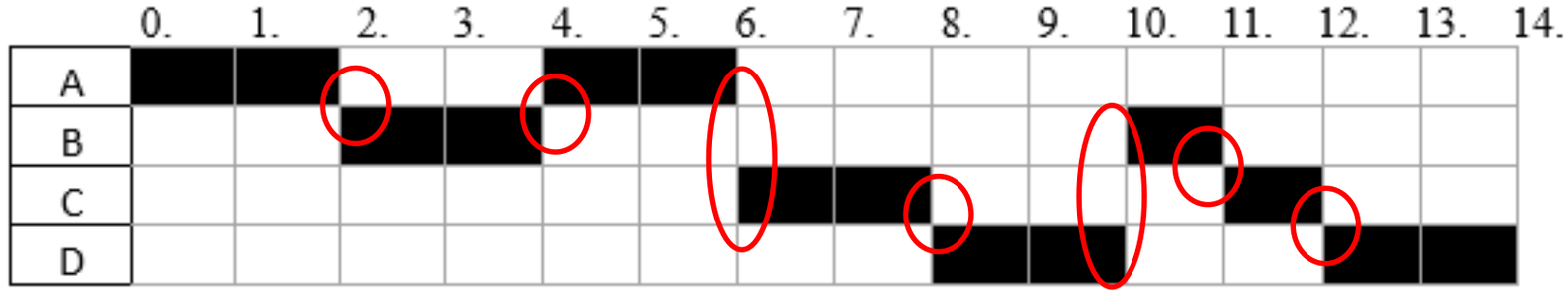
More Round Robin Practice



- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

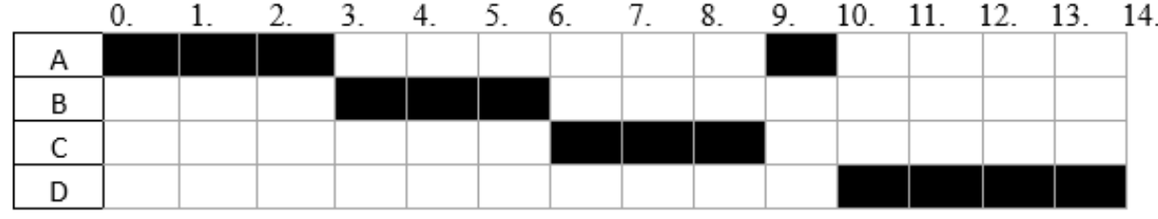
- Which processes are in the ready queue at time 9?
  - D is running, so it is not in the queue
  - A has finished
  - B and C still have to finish, so they are in the queue.

More Round Robin Practice

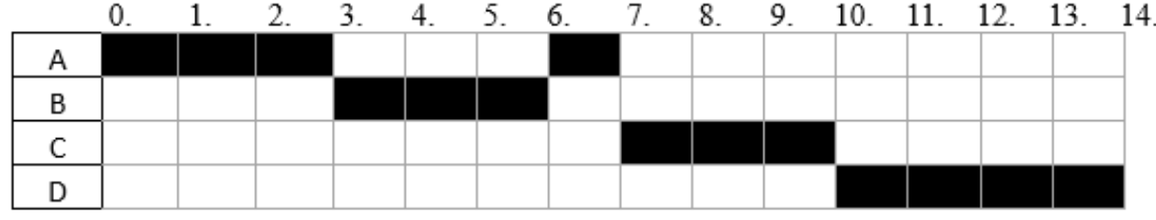


- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?
  - Currently there are 7 context switches
  - If quantum was 3:

Depends on if C shows up at time 3 or 4



  - Or:



Either way, only 4 context switches, so 3 less than quantum = 2