Mission

Background: Priority Queue

To build a scheduler, a fundamental data structure is a priority queue. You do not need to implement one, but should read and understand `libpriqueue`, our priority queue library. You will be using this library in your scheduler.

Scheduler

You will need to implement scheduling callbacks for a userspace threading library.

The scheduling algorithms you are going to implement are:

- First Come First Served (FCFS)
- Preemptive Priority (PPRI)
- Priority (PRI)
- Preemptive Shortest Remaining Time First (PSRTF)
- Round Robin (RR)
- Shortest Job First (SJF)

You can read up on scheduling in the section at the end "High Level Scheduler Overview".

You should use the priority queue that we provided to help you complete this part of the lab.

To complete this task, you must implement the six comparator functions and eight scheduler functions (and one optional one) defined in `libscheduler.c`. These functions are self-descriptive, but a full function outline for each function is provided for you in the file. These functions will be utilized by the green threading library (Check out the section on Gthreads below).

You might want to understand how scheduler works. So we put a detailed explanation in the bottom of this webpage.

Directions

To help you finish this work efficiently, we recommend you to follow these steps:

1. Understand when your function will be called.
2. Try to write pseudocode for each comparator first and see what kind of information you will need. For example, you probably need the arrival time of each job so you can implement FCFS by setting priority according to time.
3. Create data members in `job_info` you need for step 2.
4. Go back and complete your comparator functions.

The second part of the task is to set up scheduler itself and manage incoming jobs and completed jobs. Now you should implement those functions related to the CPU
(like `scheduler_new_job()`, `scheduler_job_finished()`, `scheduler_quantum_expired()`).

1. Take a look at all these functions, write some pseudocode to realize your thoughts.
2. You might need to implement some helper functions to help you write these functions.
3. Finish these functions.

The last part of your job is computing stats and clean-up, which is fairly trivial. You may need some extra variables to help you keep track of these stats.

Job Struct

We have provided a job struct defined in libscheduler.h. You do not need to modify the **state**, **ctx**, or **stack_start** fields. The only field you will be using or modifying is **metadata**, where you must insert your **job_info** struct that you will define in **libscheduler.c**.

Functions you need to implement

The only graded file is **libscheduler.c**. You will need to augment the `job_info` struct and implement the following functions:

void scheduler_start_up(scheme_t s)

This function is implemented for you, but you can add some code to it if you need to initialize any global variables.

int comparer_fcfs(const void *a, const void *b)

Comparer function for fcfs scheme

int comparer_ppri(const void *a, const void *b)

Comparer function for ppri scheme

int comparer_pri(const void *a, const void *b)

Comparer function for pri sscheme

int comparer_psrtf(const void *a, const void *b)

Comparer function for psrtf scheme

int comparer_rr(const void *a, const void *b)

Comparer function for rr scheme

int comparer_sjf(const void *a, const void *b)

Comparer function for sjf scheme

void scheduler_new_job(job *newjob, int job_number, double time, scheduler_info *sched_data)

This function is responsible for setting up a new job. You must populate `newjob->metadata` with the information found in `scheduler_info`. The contents of `newjob->metadata` cannot be a pointer to `sched_data` since this variable may be on the stack of the calling function. Once you've set up `newjob` offer it to the queue.

job *scheduler_quantum_expired(job *job_evicted, double time)

This function is called at the end of every time quantum. If there is no job currently running, `job_evicted` will be NULL. If the current scheme is not preemptive and `job_evicted` is not NULL, return `job_evicted`. In all other cases, if `job_evicted` is not NULL, place it back on the queue and return a pointer to the next job that should run. (Note, it is possible for the next job to be the same as `job_evicted`)

void scheduler_job_finished(job *job_done, double time)

This function will be called when a job is finished. You should update any statistics you are collecting and free any buffers you may have allocated for this job's metadata.

double scheduler_average_waiting_time()

This function returns the average waiting time across all jobs so far.

double scheduler_average_turnaround_time()

This function returns the average turnaround time across all jobs so far.

double scheduler_average_response_time()

This function returns the average response time across all jobs so far.

void scheduler_show_queue()

This is an optional function that you can implement for debugging purposes.

Testing

We have provided three interfaces to testing this lab. The first testing interface is **scheduler_test.c**. This file allows you to directly test the functions you've implemented in libscheduler.c. We recommend using this one, and have already populated it with a few test cases of our own! This is

how the blind test will be testing your code. There are 10 tests that we have provided you. You can run each test with the following:

```
./scheduler_test [test_no]
```

On success the test will return zero (you can check the return code with `echo $?`). On failure one of the asserts will cause an error.

The second testing interface is write your own tests similar to the ones in **main.c** in **gtest.c**. You can then run `./gtest` and inspect gthreads.log or any printouts to check if your implementation works.

The final testing interface should be thought of as more of a way to gain intuition behind the concepts rather than a way to test your code for correctness. This method is to run your scheduler with the green threading library. You can see example code in **main.c** which you will not be able to edit. The expected outputs for the various scheduling algorithms are stored in **examples/** and you will be able to diff your output to see if your scheduler produces the correct output. There is an example below.

```
./main fcfs > test.txt
diff test.txt examples/expected_fcfs.txt
```

For your convenience, we've wrapped this with the bash script **testall.sh**. Running `./testall.sh` will run all the schemes and diff them with the expected output to check if your implementation is correct. If you'd like to test specific schemes, you can pass those in as arguments, for example `./testall.sh rr fcfs pri` will only test round robin, first-come-first-serve and priority.

However, since this method of testing relies on outputs generated every second, it may not accurately reflect the schedulers behavior, and may falsely report your solution as correctly working. To get around this, you can also take a look at the generated log in **gthread.log**. This contains information about each thread's context switches and you can manually inspect it to see if it does what you expect.

Using the log file, we have built a visualizer for this data that will display which thread ran in approximately 500ms intervals. Note that the visualizer doesn't perfectly display all the context switches, and performs especially badly with round robin output. However, it works well for schemes such as

sjf and will display the following if **gthread.log** contains the log of running `./main sjf`:

```
['d10', 'd40', 'd70', 'da0', 'dd0']
d10: ++
d40:   +++
d70:      +++++
da0:           +++++++
dd0:                  ++++++++
```

There are couple things to note about this output. The first is that each '+' represents something slightly more than half a second so we interpret every two '+'s as something close to 1s. The second thing is to note that the thread names are replaced by a uid in the log file. You can which ones corresponding to which by looking through the log for registration messages and the code to check which order the threads were created.

Spooky bugs

This lab has some strange gotchas when testing with the gthread library. For this reason, we recommend using (and augmenting) the tests in `scheduler_test.c`. If you notice any random segfaults or freezes in the program that occur non-deterministically (maybe only once or twice every 10 runs) please report this to us so we can get that patched! (This will not affect grading since the grader will directly test the functions in **libscheduler.c** as opposed to the actual context switches generated by the green threading library.

The threading model - gthreads

Gthreads is an implementation of green threads in c! It uses libscheduler to schedule the threads.

NOTE: The green thread scheduler uses alarms and a SIGALRM handler, it is an error to use some other handler for SIGALRM with gthreads.

Constants

There are two constants defined as enums in gthread.h, MaxGThreads (not used, remove) and StackSize. Stack size determines the size of the stack for each green thread spawned.

Gthread Functions

gtinit

This function should be called before anything else

This function sets up the scheduler and a signal handler for SIGALRM. It is undefined behavior to call any other function in gthread before this one.

Takes in a scheme_t detailing what scheduling algorithm to be used.

gtgo

This is like pthread_create. It spawns a new green thread (won't start until it's actually scheduled).

It takes in the function to execute and a scheduler_info* to get it's scheduler attributes.

gtstart

This function starts off the scheduling process with a call to ualarm.

gtret

This function should be called from a thread to clean up and exit. If called from main it will wait for all other threads to finish before exiting with the status as the argument to gtret. If any other thread calls this function, it is equivalent to calling return.

gtsleep

This function will sleep for at least the number of seconds specified by the argument. Unlike sleep(2), this function will also call yield and allow another thread to run (if there is one on the queue).

gtdoyield

This function is a wrapper around the internal function gtyield and might perform a context switch to another thread. The return value will be true if a

context switch occurred and false otherwise. The argument is not important, so long as it is not -1 or SIGALRM.

gtcurrjob

Returns a job* indicating the current running job

Okay, so how does this work?

The idea of green threads is essentially to have a user-space thread that is lighter than a pthread, but at the cost of not executing in parallel. Instead a green thread will switch between many “threads of execution” giving the illusion of parallel processing. As you might have guessed, this switch involves what we call a “context switch” that allows us to save the current state of the thread before moving to a different one.

To learn more about the concept, read this green threading intro article that we used as a starting point.

Now, let’s talk about what we’ve added on top of that very simple implmentation. We need a way to preempt threads. For our purposes, a signal is an acceptable solution to this problem. We have used the function ualarm(3) to schedule alarms on regular intervals. The handler then calls gtyield which will call scheduler_quantum_expired to select the next job to run.

Note that almost all the scheduling magic is implemented in libscheduler! The only exception is that the main thread will never be sent to any of the functions in libscheduler. Instead, every other quanta, gtyield will store the current job do a context switch to main, and the next time gtyield is called from main, the process will switch back to the stored job.

High Level Scheduler Overview

Schedulers are pieces of software programs. In fact, you can implement schedulers yourself! If you are given a list of commands to exec, a program can schedule them

them with SIGSTOP and SIGCONT. These are called user space schedulers. Hadoop and python's celery may do some sort of user space scheduling or deal with the operating system.

At the operating system level, you generally have this type of flowchart, described in words first below. Note, please don't memorize all the states.

1. New is the initial state. A process has been requested to schedule. All process requests come from fork or clone. At this point the operating system knows it needs to create a new process.
2. A process moves from the new state to the ready. This means any structs in the kernel are allocated. From there, it can go into ready suspended or running.
3. Running is the state that we hope most of our processes are in, meaning they are doing useful work. A process could either get preempted, blocked, or terminate. Preemption brings the process back to the ready state. If a process is blocked, that means it could be waiting on a mutex lock, or it could've called sleep – either way, it willingly gave up control.
4. On the blocked state the operating system can either turn the process ready or it can go into a deeper state called blocked suspended.
5. There are so-called deep slumber states called blocked suspended and blocked ready. You don't need to worry about these.

We will try to pick a scheme that decides when a process should move to the running state, and when it should be moved back to the ready state. We won't make much mention of how to factor in voluntarily blocked states and when to switch to deep slumber states.

Measurements

Scheduling affects the performance of the system, specifically the *latency* and *throughput* of the system. The throughput might be measured by a system value, for example, the I/O throughput - the number of bits written per second, or the number of small processes that can complete per unit time. The latency might be measured by the response time – elapse time before a process can start to send a response – or wait time or turnaround time –the elapsed time to complete a task. Different schedulers offer different optimization trade-offs that may be appropriate for desired use. There is no optimal scheduler for all possible environments and goals. For example, Shortest Job First will minimize total wait time across all jobs but in interactive (UI) environments it would be preferable to minimize response time at the expense of some throughput, while FCFS seems intuitively fair and easy to implement but suffers

from the Convoy Effect. Arrival time is the time at which a process first arrives at the ready queue, and is ready to start executing. If a CPU is idle, the arrival time would also be the starting time of execution.

What is preemption?

Without preemption, processes will run until they are unable to utilize the CPU any further. For example the following conditions would remove a process from the CPU and the CPU would be available to be scheduled for other processes. The process terminates due to a signal, is blocked waiting for concurrency primitive, or exits normally. Thus once a process is scheduled it will continue even if another process with a high priority appears on the ready queue.

With preemption, the existing processes may be removed immediately if a more preferred process is added to the ready queue. For example, suppose at t=0 with a Shortest Job First scheduler there are two processes (P1 P2) with 10 and 20 ms execution times. P1 is scheduled. P1 immediately creates a new process P3, with execution time of 5 ms, which is added to the ready queue. Without preemption, P3 will run 10ms later (after P1 has completed). With preemption, P1 will be immediately evicted from the CPU and instead placed back in the ready queue, and P3 will be executed instead by the CPU.

Any scheduler that doesn't use some form of preemption can result in starvation because earlier processes may never be scheduled to run (assigned a CPU). For example with SJF, longer jobs may never be scheduled if the system continues to have many short jobs to schedule. It all depends on the https://en.wikipedia.org/wiki/Scheduling_(computing)#Types_of_operating_system_schedulers.

Why might a process (or thread) be placed on the ready queue?

A process is placed on the ready queue when it can use a CPU. Some examples include:

- A process was blocked waiting for a `read` from storage or socket to complete and data is now available.
- A new process has been created and is ready to start.
- A process thread was blocked on a synchronization primitive (condition variable, semaphore, mutex lock) but is now able to continue.
- A process is blocked waiting for a system call to complete but a signal has been delivered and the signal handler needs to run.

Measures of Efficiency

First some definitions

1. `start_time` is the wall-clock start time of the process (CPU starts working on it)
2. `end_time` is the end wall-clock of the process (CPU finishes the process)
3. `run_time` is the total amount of CPU time required
4. `arrival_time` is the time the process enters the scheduler (CPU may start working on it)

Here are measures of efficiency and their mathematical equations

1. `Turnaround Time` is the total time from when the process arrives to when it ends. `end_time - arrival_time`
2. `Response Time` is the total latency (time) that it takes from when the process arrives to when the CPU actually starts working on it. `start_time - arrival_time`
3. `Wait Time` is the *total* wait time or the total time that a process is on the ready queue. A common mistake is to believe it is only the initial waiting time in the ready queue. If a CPU intensive process with no I/O takes 7 minutes of CPU time to complete but required 9 minutes of wall-clock time to complete we can conclude that it was placed on the ready-queue for 2 minutes. For those 2 minutes, the process was ready to run but had no CPU assigned. It does not matter when the job was waiting, the wait time is 2 minutes. `end_time - arrival_time - run_time`

Convoy Effect

The convoy effect is when a process takes up a lot of the CPU time, leaving all other processes with potentially smaller resource needs following like a Convoy Behind them.

Suppose the CPU is currently assigned to a CPU intensive task and there is a set of I/O intensive processes that are in the ready queue. These processes require a tiny amount of CPU time but they are unable to proceed because they are waiting for the CPU-intensive task to be removed from the processor. These processes are starved until the CPU bound process releases the CPU. But, the CPU will rarely be released. For example, in the case of an FCFS scheduler, we must wait until the process is blocked due to an I/O request. The I/O intensive process can now finally satisfy their CPU needs, which they can do quickly because their CPU needs are small and the CPU is assigned back to the CPU-intensive process again. Thus the I/O performance of the whole system suffers through an indirect effect of starvation of CPU needs of all processes.

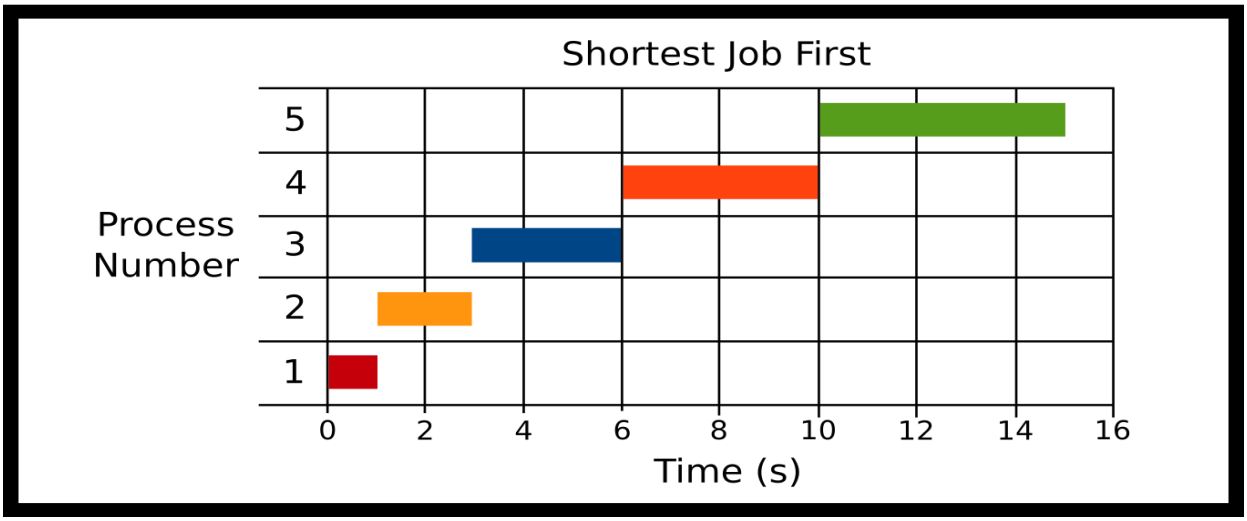
This effect is usually discussed in the context of FCFS scheduler; however, a Round Robin scheduler can also exhibit the Convoy Effect for long time-quanta.

Scheduling Algorithms

Unless otherwise stated

1. Process 1: Runtime 1000ms
2. Process 2: Runtime 2000ms
3. Process 3: Runtime 3000ms
4. Process 4: Runtime 4000ms
5. Process 5: Runtime 5000ms

Shortest Job First (SJF)



- P1 Arrival: 0ms
- P2 Arrival: 0ms
- P3 Arrival: 0ms
- P4 Arrival: 0ms

- P5 Arrival: 0ms

The processes all arrive at the start and the scheduler schedules the job with the shortest total CPU time. The glaring problem is that this scheduler needs to know how long this program will run over time before it ran the program.

Technical Note: A realistic SJF implementation would not use the total execution time of the process but the burst time or the number of CPU cycles needed to finish a program. The expected burst time can be estimated by using an exponentially decaying weighted rolling average based on the previous burst time (Silberschatz, Galvin, and Gagne #ref-silberschatz2005operating Chapter 6). For this exposition, we will simplify this discussion to use the total running time of the process as a proxy for the burst time.

Advantages

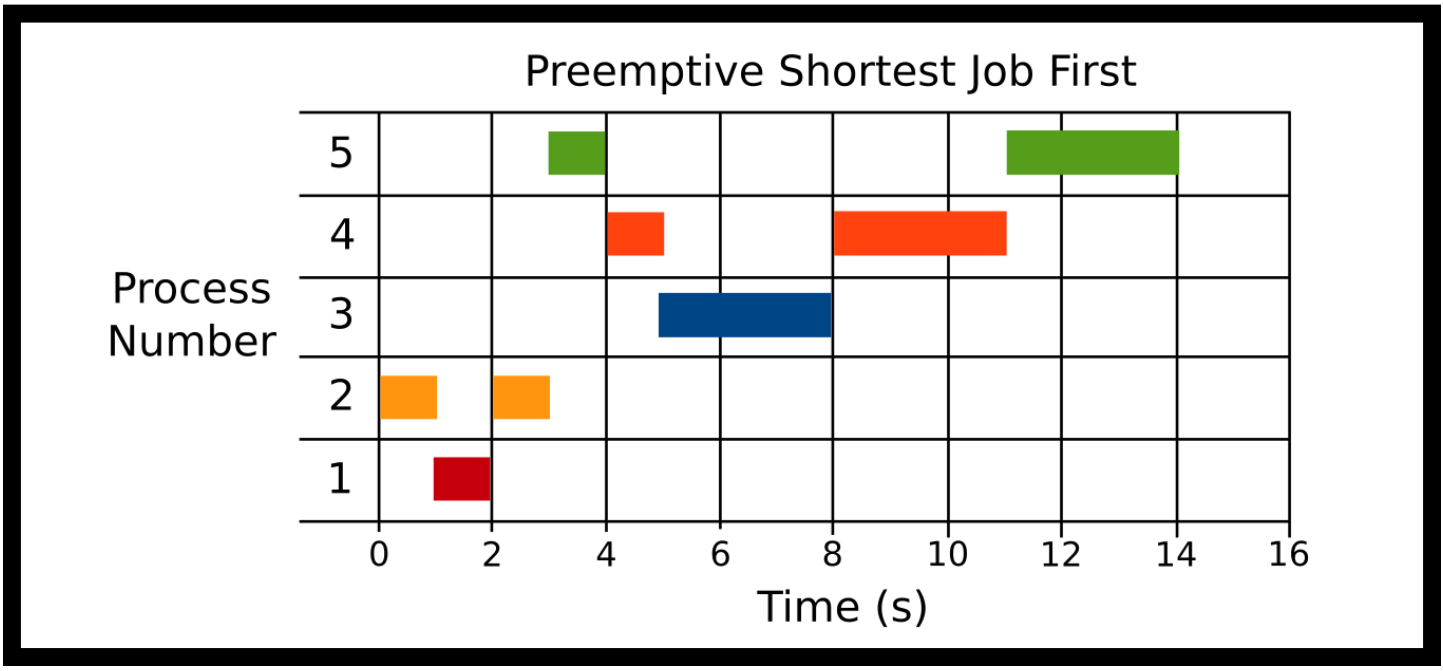
1. Shorter jobs tend to get run first
2. On average wait times and response times are down

Disadvantages

1. Needs algorithm to be omniscient
2. Need to estimate the burstiness of a process which is harder than let's say a computer network

Preemptive Shortest Job First (PSJF)

Preemptive shortest job first is like shortest job first but if a new job comes in with a shorter runtime than the total runtime of the current job, it is run instead. If it is equal like our example our algorithm can choose. The scheduler uses the *total* runtime of the process. If the scheduler wants to compare the shortest *remaining* time left, that is a variant of PSJF called Shortest Remaining Time First (SRTF).



- P2 at 0ms
- P1 at 1000ms
- P5 at 3000ms
- P4 at 4000ms
- P3 at 5000ms

Here's what our algorithm does. It runs P2 because it is the only thing to run. Then P1 comes in at 1000ms, P2 runs for 2000ms, so our scheduler preemptively stops P2, and let's P1 run all the way through. This is completely up to the algorithm because the times are equal. Then, P5 Comes in – since no processes running, the scheduler will run

process 5. P4 comes in, and since the runtimes are equal P5, the scheduler stops P5 and runs P4. Finally, P3 comes in, preempts P4, and runs to completion. Then P4 runs, then P5 runs.

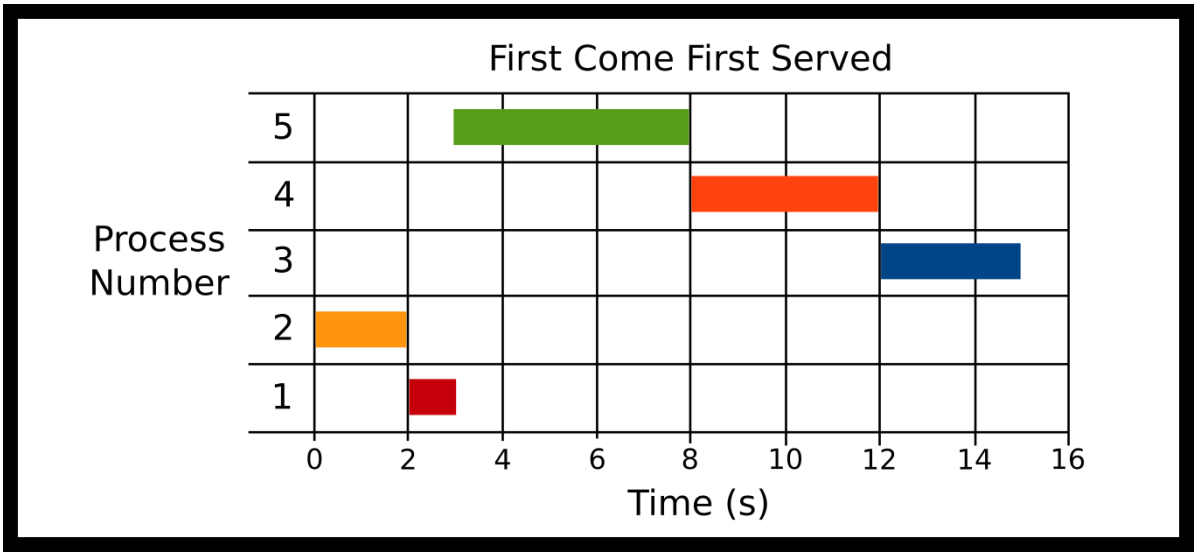
Advantages

1. Ensures shorter jobs get run first

Disadvantages

1. Need to know the runtime again
2. Context switching and jobs can get interrupted

First Come First Served (FCFS)



- P2 at 0ms
- P1 at 1000ms
- P5 at 3000ms
- P4 at 4000ms
- P3 at 5000ms

Processes are scheduled in the order of arrival. One advantage of FCFS is that scheduling algorithm is simple The ready queue is a FIFO (first in first out) queue. FCFS suffers from the Convoy effect. Here P2 Arrives, then P1 arrives, then P5, then P4, then P3. You can see the convoy effect for P5.

Advantages

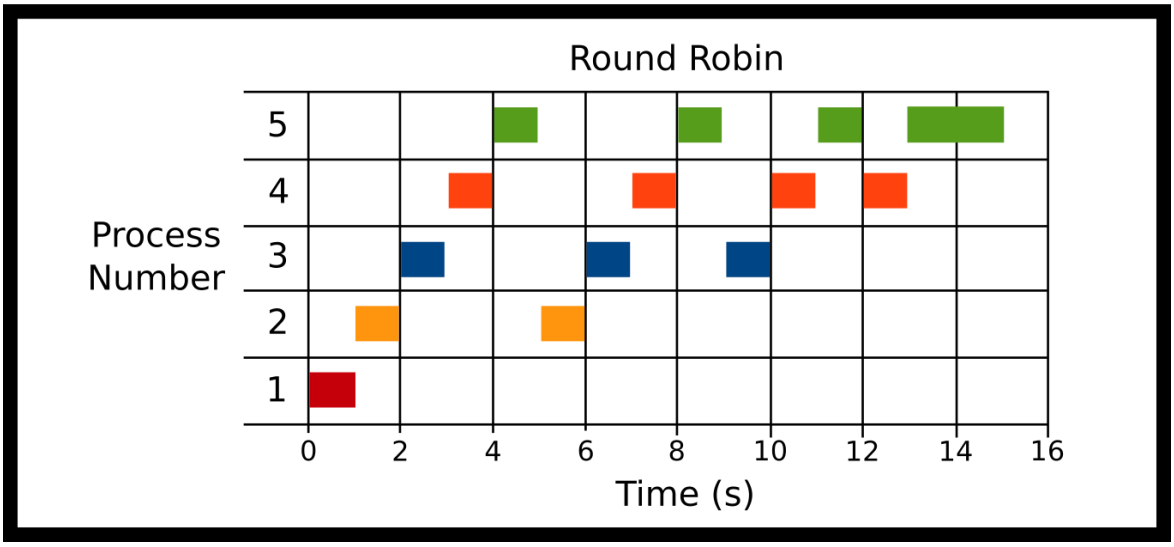
- Simple algorithm and implementation
- Context switches infrequent when there are long-running processes
- No starvation if all processes are guaranteed to terminate

Disadvantages

- Simple algorithm and implementation
- Context switches infrequent when there are long-running processes

Round Robin (RR)

Processes are scheduled in order of their arrival in the ready queue. After a small time step though, a running process will be forcibly removed from the running state and placed back on the ready queue. This ensures long-running processes refrain from starving all other processes from running. The maximum amount of time that a process can execute before being returned to the ready queue is called the time quanta. As the time quanta approaches to infinity, Round Robin will be equivalent to FCFS.



- P1 Arrival: 0ms
- P2 Arrival: 0ms
- P3 Arrival: 0ms

- P4 Arrival: 0ms
- P5 Arrival: 0ms

Quantum = 1000ms

Here all processes arrive at the same time. P1 is run for 1 quantum and is finished. P2 for one quantum; then, it is stopped for P3. After all other processes run for a quantum we cycle back to P2 until all the processes are finished.

Advantages

1. Ensures some notion of fairness

Disadvantages

1. Large number of processes = Lots of switching

Priority

Processes are scheduled in the order of priority value. For example, a navigation process might be more important to execute than a logging process.