Cpu Scheduling - Simulation

Due: 12-10-2024 (Final Exam Day) Cpu scheduling is a classic and contemporary computer science problem that started when early computers had processors that remained idle much of the time. The goal initially was to get multiple programs loaded into memory, so they could run back to back. This was still inefficient and needed improvement. That soon evolved into multiple programs loaded into memory and when one process blocked itself, the cpu would work on another available process (multi-programming). This, of course, kept evolving into our multi-threaded / multi-processor world. Yet, we still need to be cognizant of keeping the cpu(s) busy! So what does a scheduler do?

Scheduler

A scheduler makes choices in order to minimize or maximize a set of criteria, where the criteria really can be simplified into this one concept: "minimize the time any process is doing nothing". However we measure that, or label it, its the crux of the scheduling problem.

Goals

1. *Maximizing Throughput* (the total amount of work completed per time unit)
2. *Minimizing Wait Time* (time from work becoming ready until the first point it begins execution) (We define it slightly differently)
3. *Minimizing Latency* or response time (time from work becoming ready until it is finished)
4. *Maximizing Fairness* (equal CPU time to each process, or more generally appropriate times according to the priority and workload of each process).

- In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise.
- Preference is measured by any one of the concerns mentioned above, depending upon the user's needs and objectives.

The scheduler will attempt to accomplish the above goals by moving a process around through a series of states. Of course each process has its own unique set of needs (IO intensive, CPU intensive for example), and the scheduler cannot change what the process needs or when it needs it. Its power is in determining when a process gets access to a resource, the most important of which is the CPU. If it's smart about the order in which it allows access, then the above goals can be met.

As I mentioned, the scheduler moves processes around through a series of states, based on the processes needs. Our simulation will be implementing the most basic set of states, in which there are five (see below).

Job State

1. **NEW** - The process is being created, and has not yet begun executing.
2. **READY** - The process is ready to execute, and is waiting to be scheduled on a CPU.
3. **RUNNING** - The process is currently executing on a CPU.
4. **WAITING** (BLOCKED) - The process has temporarily stopped executing, and is waiting on an I/O request (peripheral device to come open).
5. **IO** - Process has gained access to one of the peripheral devices.
6. **TERMINATED** - The process has completed.

Each state is represented as a queue that holds each process that is currently in that state. The success or failure of scheduling boils down to moving each process from queue to queue in an efficient and consistent manner. Efficiency depends on multiple factors, the main one being the actual scheduling algorithm choosing which process gets cpu time or resource time. The next section discusses the algorithms your simulation needs to implement.

Scheduling Algorithms

First-Come-First-Serve, FCFS

- FCFS is very simple - FIFO simply queues processes in the order that they arrive in the ready queue.
- This is the simplest scheduling algorithm.
- This is a **non-preemptive** scheduling algorithm.
- Context switches only occur upon cpu burst termination.
- **Example**:
  - *Pn* is first to arrive and has a 35 time unit cpu burst.
  - Once it is in the `Running` state it will stay there until its burst is complete and it moves to the `Wait` queue.
  - After it finishes its `IO` burst, it comes back to the `Ready` queue at the end of the line and waits for the next open cpu.

Round-Robin, RR

- The scheduler assigns a fixed time unit per process known as a time-slice or time-quantum, and cycles through each process equally.
- If the process completes within that time-slice it gets terminated otherwise it is rescheduled after giving a chance to all other processes.
- This is a **preemptive scheduling** algorithm.
- **Example**:
  - *Pn* is first to arrive and has a 35 time unit cpu burst.
  - Once it is in the `Running` state it will stay there until its `time-slice` is up, in which it will be interrupted and sent to the end of the `Ready` queue.
  - It will continue this circular cycle until it finishes its cpu burst. So if the `time-slice` is 5, then it will have five rounds of `Running->Ready`

before it can finally go to the `Wait` queue and start its `IO` burst.
  - The `Wait` queue has no `time-slice` so once it gains an `IO` device it keeps it until the burst is over and then back to the `Ready` queue"

Priority-Based, PB

- The operating system assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority.
- Lower-priority processes get interrupted by incoming higher-priority processes.
- This is a **preemptive scheduling** algorithm.
- Runs into possibility of starving processes with low priorities.
- I think we can postulate what would happen if we do not implement some kind of `promotion` for low priority processes.
- I will leave it up to you how you handle it. It does not have to be something complicated, it could be based on total time in system (which is tracked per process anyway) or even some random promotion event. Use your imagination.
- **Example1**:
  - *Pn* is first to arrive and has a 35 time unit cpu burst and the highest priority.
  - Next however many processes to arrive get a big so what and have to wait.
- **Example2**:
  - *Pn* is first to arrive and has a 35 time unit cpu burst and the lowest priority.
  - *Pn+1* is next to arrive and has a 10 time unit cpu burst and a medium priority, it interrupts *Pn* sending to the place in the ready queue equivalent to its priority and takes over the cpu.
  - *Pn+2* is next to arrive and has a 100 time unit cpu burst and the highest priority, it interrupts *Pn+1* sending it to the place in the ready queue equivalent to its priority and takes over the cpu. And stays since it cannot be preempted via priority.

Shortest-Job-First, SJF (Do Not Implement)

- Shortest job first (SJF) also known as Shortest job next (SJN) or shortest process next (SPN), is a scheduling policy that selects for execution the waiting process with the smallest execution time.
- SJF is a **non-preemptive** algorithm
- A disadvantage of using shortest job next is that the total execution time of a job must be known before execution. But we know, so we can implement it.
- Total Execution Time = Sum(cpuBurst**1**+cpuBurst**2**+cpuBurst**3**+...+cpuBurst**n**)

Shortest-Remaining-Time, SRT (Do Not Implement)

- Shortest remaining time, also known as shortest remaining time first (SRTF), is a scheduling method that is a **preemptive version** of shortest job next scheduling.
- In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute.
- Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, the process will either run until it completes or get preempted if a new process is added that requires a smaller amount of time.
- Execution Time Remaining = Sum of remaining cpuBursts.

MLFQ (Multi Level Feedback Queue)

A Multi-Level Feedback Queue (MLFQ) is a CPU scheduling algorithm designed to manage tasks of varying priorities dynamically. It improves efficiency by adjusting a process's priority level based on its observed behavior, adapting to both I/O-bound and CPU-bound processes.

Key Characteristics:

- Multiple Queues: The system has multiple queues, each with a different priority level. Processes in higher-priority queues are scheduled before those in lower-priority queues.
- Time Quantum Variation: Each queue can have a different time quantum (the amount of time a process is allowed to run). Higher-priority queues often have shorter time quanta, favoring interactive or short tasks, while lower-priority queues have longer time quanta.
- Dynamic Priority Adjustment: A process's priority changes based on its CPU usage pattern. For example:
- If a process consumes its entire time quantum, it is demoted to a lower-priority queue.
- If a process yields the CPU before its time quantum expires (indicating I/O-bound behavior), it may stay in the same queue or even be promoted to a higher-priority queue.

Algorithmic Steps:

1. Initialization: Define multiple queues, each with its priority level and time quantum.
2. Scheduling:

- Start with the highest-priority queue. Select the process at the head of the queue and allocate CPU time based on the queue's time quantum.
- If the process completes within its time quantum, remove it from the queue.

- If the process exhausts its time quantum without completing, demote it to a lower-priority queue.

3. Feedback Mechanism:

- Adjust the priority of processes based on behavior. For example, processes that yield the CPU frequently may be promoted back to higher-priority queues.

4. Aging (Optional): To prevent starvation of low-priority processes, periodically promote them to higher-priority queues, ensuring they receive CPU time.

MLFQ is adaptive and responsive to varying process behaviors, making it suitable for systems with a mix of interactive and batch processes. However, its complexity and tuning requirements (e.g., number of queues, time quanta, and feedback policies) are critical to achieving optimal performance.

CFS (Completely Fair Scheduler)

- NOT IMPLEMENTING

Resources

The goal of each process is to gain access to a resource at the time it needs it, so that it can complete its lifecycle. In class we discussed implementing queues for each state. All of them unbounded with the exception of a single cpu. Well, now I would like a little more control over the amount of available resources. In the table below, you can see all the queues are `unbounded` but now we are adding a bounded size on the `IO` queue.

To emphasize the resource issue again, I need to mention that the number of resources heavily influences throughput. Resources go up, throughput gets better. We are making each IO device generic for ease of implementation, but if many process were waiting for a specific device, no matter how many total IO devices there are, then through put could suffer. For ease of implementing this simulation, all resources will be categorized as one of two things: a `CPU`, or an `IO` device.

Processors

In fact, all CPU's (processors) are assumed to be the same. This means we do not need to worry about **heterogeneous** or **homogenous** processors.

- **heterogeneous**: different processor types. Usually meaning one would run kernel code and others everything else, or one cpu being the "cpu in charge" delegating to others.
- **homogenous**: all cpu's the same type. But could still mimic above strategy regardless.

In our simulation a cpu is available to any and all processes, and we will have the ability to change the number of cpu's based on the current simulation run. I will say that `1-4 cpu's` is what we will go with.

IO Devices

Likewise, we will not distinguish between different types of IO devices (like printers, network cards, disks, etc.). So, again, when you write code to implement an IO device, you can simply create duplicate instances to increase the number of devices for processes to use during their IO bursts. `3 - 6 devices` We should discuss more in class to determine a good number to run our sims.

Getting Jobs

Instructions to Fetch CPU Jobs from the API

This is a simple guide to interact with the CPU jobs API. The Python script provided HERE gives you a very succinct example showing you which parameters to use and the order in wich to call end points so your simulation can run.

Step 1: Initialize the Session

- Call the `/init` endpoint to start a session and have the jobs generated for your simulation.
- **Input**: A configuration dictionary (generated using getConfig).
- **Output**: session_id (unique to the session) and start_clock (initial clock time).

Code Example

```
config = getConfig(client_id="your_client_id")
response = init(config)
start_clock = response['start_clock']
session_id = response['session_id']
```

- The `start_clock` will be an integer value from 1-4 digits. The chances of your clock starting at zero are `1/1000`. Once you receive the start clock value, initialize your simulation clock to that value, then increment by 1 for every iteration of your simulation.
- The `session_id` identifys each `client_id`'s unique runs. It starts at zero and adds one every time you restart (by calling init).

Step 2: Fetch Jobs at a Given Clock Time

You have your system start clock time (and your session id), so you will loop and increment your system clock calling the `/job` endpoint every clock tick to check if any jobs have arrived.

- Use the `/job` endpoint to fetch jobs available at the current clock time. This means you will loop, and call `/job` every clock tick to see if new jobs arrive. Mostly it will be none, but depending on the configuration file, you could have many jobs show up at the same time.
- **Input**: `client_id`, `session_id`, and `clock_time`.
- **Output**: A list of jobs that arrived at the specified clock time.

Code Example

```
response = getJob(client_id="your_client_id", session_id=session_id, clock_time=start_clock)
if response and response['success']:
jobs = response['data']
print(f"Jobs at clock {start_clock}: {jobs}")
```

Step 3: Process Each Job's Bursts

- For each job fetched, call `/burst` to retrieve that jobs next burst details. This is the end point that will get called the most since everytime a job finishes a "burst", you need to call `/burst` again until you get a burst type of `Exit`.
- **Input**: `client_id`, `session_id`, and `job_id`.
- **Output**: Details of the current burst:
  - burst_id (int)
  - burst_type (str) [CPU, IO, EXIT]
  - duration (int)

Code Example

```
for job in jobs:
job_id = job['job_id']
burst = getBurst(client_id="your_client_id", session_id=session_id, job_id=job_id)
print(f"Job {job_id}, Burst Details: {burst}")
```

Simulation Loop

- Keep incrementing the clock and calling the `/job` endpoint while jobs are still in your system.
- For each job, continue calling `/burst`, after that jobs current burst hits zero. This will move it through all the process states until its last `/burst` call for a job returns `EXIT`.
- To terminate, you must process all jobs. This can be discovered in two ways:
  1. Call `jobs_left` and if it equals zero, you have no more jobs.
  2. When the `system_clock - latest_jobs_arrival_time` > `max_job_interval` then no more jobs are coming.

Complete Workflow Example

Here's a minimal example that ties everything together:

```
client_id = "your_client_id"
config = getConfig(client_id)
response = init(config)

if response:
start_clock = response['start_clock']
session_id = response['session_id']
clock = start_clock

    while True:
        # Check remaining jobs
        jobs_left = getJobsLeft(client_id, session_id)
        if not jobs_left:
            print("No more jobs left. Ending session.")
            break

        # Fetch jobs for the current clock time
        response = getJob(client_id, session_id, clock)
        if response and response['success']:
            for job in response['data']:
                job_id = job['job_id']
                print(f"Processing Job {job_id} at Clock {clock}")

                # Check bursts for the job
                bursts_left = getBurstsLeft(client_id, session_id, job_id)
                if bursts_left:
                    burst = getBurst(client_id, session_id, job_id)
                    print(f"Burst Details for Job {job_id}: {burst}")

        # Increment clock
        clock += 1
```

Basic High Level Algorithm This is similar to the simulation loop I describe above, with a little more detail injected.

1. Jobs arrive at time `N`, and enter the `New` queue.
2. Any jobs already in the `New` queue go to the `Ready` queue.
3. Decrement burst times on Cpu(s), if any are zero, move to `Wait` queue.
4. Decrement burst times on Peripheral(s), if any are zero, move to `Ready` queue.
5. If any Cpu(s) are free, take next from `Ready` queue.
6. If any Periphal(s) are free, take next from `Wait` queue.
7. Do accounting (e.g. increment ready queue wait time) for each process in

the appropriate queues when appropriate (basically if it hasn't just been moved).

API Reference

Endpoint	Purpose	Example Input	Example Output
`/init`	Start a new session Config dictionary		{ "session_id": 13, "start_clock": 0 }
`/job`	Get jobs available at the current clock time	client_id, session_id, clock_time	List of jobs arriving at the clock time
`/burst`	Get details of a burst for a specific job	client_id, session_id, job_id	{ "burst_id": 1, "type": "CPU", "duration": 10 }
`/burstsLeft`	Get remaining bursts for a job	client_id, session_id, job_id	Integer count of bursts left
`/jobsLeft`	Get number of jobs left in the session	client_id, session_id	Integer count of jobs left

---

No Pressure

- Remember this is a simulation, where the system clock (aka timer) is simply an integer variable incremented in some loop construct.
- We will read the the processes that will run in our simulation from multiple files, where each file will emphasize a different type of load (see previous).
- This program is not a true system program, it is just a typical user application that requires no spawning of processes, no timer interrupt handling, no I/O interrupt handling, etc. It is a **simulation**

Requirements

Visualization

- Your program must use some form of "visual presentation" to show:
  - Each of the 6 queues [New, Ready, Running, Waiting, IO, Exit] and which processes are in them
- Your choice of "visual presentation" can be NCurses or a GUI program with something like DearPyGui or my Rich Example
- Specific messages at some time throughout the simulation. Obviously for presentation purposes we will do short runs with small files. If you were to use my Python Rich table example, you could print messages below the table in a panel or similar.

- It won't matter how correct your code is if we cannot visually follow along.

  [!IMPORTANT]

- **IMPLEMENT A METHOD TO PAUSE YOUR SIM.** Below is an example to pause a visualization:

```
## Imports for rich or whatever
import time
import keyboard  # Install via `pip install keyboard`

# code
# code
#  .
#  .
#  .

paused = False

def toggle_pause():
    global paused
    paused = not paused
    console.print("Simulation Paused!" if paused else "Resuming Simulation...")

keyboard.add_hotkey("space", toggle_pause)

while(looping):
  while paused:  # Pause the simulation loop
      time.sleep(0.1)
  # do stuff show stuff
  # pretty colors
  # I think someone took acid

# code
# code
#  .
#  .
#  .
```

Presentation

- You will show 6 runs, 2 of each scheduling type.
- I will provide an `srand` seed for each of the 6 jobs ensureing everyone gets the same values generated for their presentations.
- You should be able to enter this seed and the scheduling algorithm from the command line (`sys.argv`)

  - Example:
    * `python sim.py sched=RR seed=32134 cpus=4 ios=3`
    * `python sim.py sched=FCFS seed=666 cpus=2 ios=2`
    * `python sim.py sched=MLFQ seed=32000 cpus=4 ios=7`

- The types of messages that should print as your presentation runs are listed below.
- Coloring times, process's, cpu's, and device's would be preferred.
- Messages:
  - At tn job *pn* entered new queue.
    * Ex: At t:31 job p12 entered new queue
  - At tn job *pn* obtained *cpun*
    * Ex: At t:35 job p12 obtained cpu:0
  - At tn job *pn* obtained *devicen*
    * Ex: At t:44 job p15 obtained device:2
- When a process terminates, the output should give that jobs stats. The stat acronyms are as follows:
  - ST = Time entered system
  - TAT = Turn Around Time (time exited system - time entered)
  - RWT = Time spent in ready queue
  - IWT = Time spent in wait queue
- Example:
  - Job *pn* TAT = *tn*, RWT = *tn*, IWT = *tn*
  - Job p:23 ST = 101 TAT = 545, RWT = 121, IWT = 211
- At the end of simulation, the simulator should display the percentage of CPU utilization, average TAT, average ready wait time, and average I/O wait time.

Complete Runs And Output

- You need aggregate data for jobs consisting of:
  - Scheduling Algorithms: [FCFS, RR, PB]
  - Cpus: [1,2,3,4]
  - Devices: [2,4,6] > Note: For Round Robin you need to change the time quantum: [3,5,7,9]
- We can discuss how to output this in a csv in class.

Deliverables

- Place code on github with a write up describing the process of writing your project and members of your group.

- Look HERE for how to write up a readme.
- Present your results in class when specified
- Ensure your presentation follows guidelines above