HW: chapter 5 all fork() coding problems are good; chapter 7 – Q1, Q2
Read 11, 7, 8, 9, 10(skim), 11

Re-read the aside atop the page with Section 5.5 ... RTFM ;)"Spend some time reading man pages, a key step in the growth of a systems programmer."

We will separate policy, which process should run, from mechanism, how we stop A and start B. Later we will
1. discuss the swapper, which moves jobs to and from the disk.
2. mention real time: ‘maximum urgency first scheduling’.
3. consider the Mars probe priority inversion oops ☹ and its fix (see handout).

We need some performance metrics to compare scheduling policies:
1. T_turnaround = T_completion – T_arrival
2. T_responseTime = T_firstrun – T_arrival
3. Fairness
4. Efficiency (keep CPU (or other hardware) as busy as possible)

First policy: FIFO
Basic algorithm: jobs are processed in order of arrival

We have 3 jobs (A, B, and C) arriving at time 0 that each take 10 seconds

How is the CPU scheduled?

Assuming order A, B, C then A, B, C ☹

What is the average turnaround time?

\[ ((10-0) + (20-0) + (30-0)) / 3 \rightarrow 20 \]

When is this a bad policy?

If A takes 100 seconds ...? ((stuck behind a) convoy effect)

How could we fix this issue? ... Second policy:

SJF

Let’s compare the average turnaround time?

\[ ((100-0) + (110-0) + (120-0))/3 \quad \text{versus} \quad ((10-0) + (20-0) + (120-0))/3 \quad \text{110 versus 50} \]

What if B and C are late to the dance (i.e., arrive at time 10)?

Define preemptive versus non preemptive scheduling
Can we now do better? Third policy!

**Shortest Time to Completion First (STCF)**

Algorithm: When a new job enters the system, determine which job has the least time left and schedule it.

So what do things look like at time 10?

A has 90 left, B and C 10 each ... schedule B or C

This is all so 1960's. I *interact* with my processes. How can schedule interactive processes?

(when can I make my first move, not when is the game done)
T_responseTime = T_firstRun - T_arrival

(mention) T_responseTime = T_nextRun - T_last-became-ready
(recall running, ready, blocked state chart)

Our first policy that takes this new metric into account:

First up, turnaround time, **SJF**, and **STCF** are no longer viable.
(e.g. for STCF if all 3 enter system at same time, each runs completely before the next starts because the first to run is now the closest to being done....)

Enter – Round Robin
Run each job for a single time slice then switch

How does round robin perform on turnaround time compared to SJF? Consider two processes arriving at time 0, taking 5ms each, and using a 1ms time slice:

SJF: \((5 + 10) \div 2 \rightarrow 7.5\)

RR: ABABABABAB \(\rightarrow (9 + 10) \div 2 \rightarrow 9.5\)

In other words, there is a classic trade-off

turnaround time vs. response time

**Length of time slice is a critical choice**

Too short means lots of context switching, too long means it has little effect.

Might be nice to give different amount of time to difference kinds of processes (e.g. I/O bound versus CPU bound)
Review - we considered two scheduling metrics:
- Turnaround time - goal of FIFO, SJF, SRWF
- Response time - goal of RR

They were at odds with each other; algorithms generally prefer one over the other.

Context: we generally have 2 types of jobs:
1. Interactive Jobs (short CPU time because of I/O blocks)
2. CPU-bound Jobs (heavy processing, not interactive)
   - Response time is important for interactive jobs, but not CPU-bound jobs
   - Interactive jobs often will give up CPU for I/O blocks

Today's Goal: adjust priorities to give
long time slices to compute bound jobs (good for turnaround) and
frequent time slices to I/O bound jobs (good for response time)

Algorithm: Multi-Level Feedback Queues (MLFQ)

Each level (queue) has its own priority

How do we choose which job to run?

Rule 1: Priority(A) > Priority(B) \(\rightarrow\) run A (and not B)

Rule 2: Priority(A) = Priority(B) \(\rightarrow\) run A and B using RR

See a problem?

Low priority jobs never run

How do we fix?

Rule 5: After some time period S, move all jobs to the topmost queue

Ok that's up what about down

Rule 4: Once a job uses up an assigned time allotment at a given level its priority is reduced
(i.e., it moves down one queue (Regardless of how many times it has given up the CPU.))

What's Missing?

Rule 3: New jobs are placed at the highest priority (topmost) queue

Can you explain how these five rules work together to provide good response time?
Why does this algorithm work well?

- Prioritizes short jobs yielding fast response time for I/O bound jobs
- New jobs start at highest level – ensures fast initial response
- If a job uses little bursts of CPU it's interactive (and consumes its allotment slowly)
- If a job doesn't give up the CPU it's probably CPU bound – move down allowing interactive jobs first go at the CPU. Optionally: might give lower levels longer turns
- Rule 4’s use of an allotment prevents process from gaming the system by doing an I/O just before their time slice ends (in order to retain its current level)
- Rule 5 prevents too many interactive jobs from starving non-interactive jobs
- Rule 5 als accounts for Jobs vacillating between interactive or non-interactive.

Design Issues

- How many queues?
- Time slice per queue? Why might longer time slices for low priority queues make sense?
- Frequency of priority boost (i.e. Rule 5) ? (Alas, setting S can be hard. No answer that is always right.)
- Might reserve highest priority for OS.
- Allow user to give the OS advice for a job’s improntance [man nice]

Summary:

MLFQ allows us to have good response and turnaround time.

MLFQ has the following rules:

1. If Priority(A) > Priority(B), run A
2. If Priority(A) = Priority(B), run with RR
3. New jobs start at highest priority
4. Once a job uses up its time allotment at its priority level, it is move down a queue
5. After time S, all jobs move back to highest priority

Actual OSes that you have used, use a form of this scheduler!