Scheduling

Questions Answered in this Lecture:

• What are some different scheduling policies?
• When do they work well?
Announcements

• Project 1a in; Project 1b out

• Project 1a: If I can’t associate your code with you, your project will not be graded (i.e, zero). Read instructions carefully!

• Reading: go read OSTEP Chapters 7 & 8, plus other readings I’ve linked

• Read the excerpt on process scheduling code for Linux

• Note on plagiarism
CPU Virtualization: Two Components

• Dispatcher -> mechanism (last week)
  • How do we switch from one process to another (ctx switch)
  • How do we save state of one process?
  • How do we interrupt the running process?
  • How do we pick the next one to run?

• Scheduler -> policy (today)
Scheduling

• This is an *old* problem! Not just applicable to OS (or computing systems for that matter)

• First well studied in the operations research (OR) community
  • “How do I best schedule my workers on the factory floor?”
  • “In what order do I send items down my assembly line?”

• You’ll never be able to forget this stuff at the grocery store

• Or the DMV

• Or the gate at O’Hare

• WHY CAN’T THE WORLD BE AS EFFICIENT AS MY OS?!
Abstracting Away

• The problem put generally:
  • n resources
  • k users (k is almost always >> n)
  • Come up with a mapping in the time domain from users to resources

• Someone’s got to wait

• We need queues…..

• Queueing Theory
The Parlance

• **Workload**: Intuitively, the set of things that’ll use our scheduler
  • Accurately, the set of *job* descriptions (arrival time, runtime)
  • As process moves between CPU (doing work) and I/O (waiting for something else to do the work), process goes from ready queue to blocked queue

• **Scheduler**: Code (logic) that decides *which* job to run

• **Metric**: a measurement of quality
Metrics we care about

- **Turnaround time**: time it takes for the job to complete once they’re submitted (completion_time – arrival_time)
- **Response time**: time it takes for interactive jobs to become active (initial_schedule_time – arrival_time)
- **Waiting time**: Job should not be queued (in the ready q) for long
- **Throughput**: completed jobs per unit time
- **Utilization**: expensive devices (CPUs, GPUs, etc.) should remain busy
- **Overhead**: number of context switches
- **Fairness**: jobs get same amount of CPU time over some interval
Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known
Scheduling Basics

**Workloads:**
- arrival_time
- run_time

**Scheduling Policies:**
- FIFO
- SJF (SJN, SPN)
- STCF
- RR

**Metrics:**
- turnaround_time
- response_time
Example: Workload, scheduler, metric

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival_time (s)</th>
<th>Run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~0</td>
<td>10</td>
</tr>
</tbody>
</table>

**FIFO**: First In, First Out
- also called FCFS (first come first served)
- run jobs in *arrival_time* order

**What is our turnaround?**: \( \text{completion_time} - \text{arrival_time} \)
FIFO: Event Trace

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~0</td>
<td>10</td>
</tr>
</tbody>
</table>

Time

0 A arrives
0 B arrives
0 C arrives
0 run A
10 complete A
10 run B
20 complete B
20 run C
30 complete C
FIFO: (Identical Jobs)

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~0</td>
<td>10</td>
</tr>
</tbody>
</table>

Gantt chart:
Illustrates how jobs are scheduled over time on a CPU
FIFO: (Identical Jobs)

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~0</td>
<td>10</td>
</tr>
</tbody>
</table>

What is the average turnaround time?

Def: turnoverad_time = completion_time - arrival_time
FIFO: (Identical Jobs)

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~0</td>
<td>10</td>
</tr>
</tbody>
</table>

What is the average turnaround time?

\[(10+20+30)/3 = 20s\]
Scheduling Basics

Workloads:
- arrival_time
- run_time

Scheduling Policies:
- FIFO
- SJF (SJN, SPN)
- STCF
- RR

Metrics:
- turnaround_time
- response_time
Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known
Any Problematic Workloads for FIFO?

**Workload:** ?

**Scheduler:** FIFO

**Metric:** turnaround is high
Example: Big First Job

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>60</td>
</tr>
<tr>
<td>B</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~0</td>
<td>10</td>
</tr>
</tbody>
</table>

Draw Gantt chart for this workload and policy...
What is the average turnaround time?
Example: Big First Job

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>60</td>
</tr>
<tr>
<td>B</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~0</td>
<td>10</td>
</tr>
</tbody>
</table>

Average turnaround time: 70s
Convoy Effect
Passing the Tractor

**Problem with Previous Scheduler:**

FIFO: Turnaround time can suffer when short jobs must wait for long jobs

**New scheduler:**

SJF (Shortest Job First)

Also (Shortest job next SJN, shortest process next (SPN))

Choose job with smallest *run_time*
Shortest Job First

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>60</td>
</tr>
<tr>
<td>B</td>
<td>~0</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~0</td>
<td>10</td>
</tr>
</tbody>
</table>

What is the average turnaround time with SJF?
SJF Turnaround Time

What is the average turnaround time with SJF?

\[(80 + 10 + 20) / 3 = \sim 36.7\text{s}\]

For minimizing average turnaround time (with no preemption):
SJF is provably optimal

Moving shorter job before longer job improves turnaround time of short job more than it harms turnaround time of long job
Scheduling Basics

**Workloads:**
- arrival_time
- run_time

**Scheduling Policies:**
- FIFO
- SJF (SJN, SPN)
- STCF
- RR

**Metrics:**
- turnaround_time
- response_time
Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known
Shortest Job First (Arrival Time)

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>60</td>
</tr>
<tr>
<td>B</td>
<td>~10</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~10</td>
<td>10</td>
</tr>
</tbody>
</table>

What is the average turnaround time with SJF?
Stuck Behind a Tractor Again

What is the average turnaround time?

\[
\frac{(60 + (70 - 10) + (80 - 10))}{3} = 63.3s
\]
Preemptive Scheduling

Prev schedulers:
- FIFO and SJF are non-preemptive
- Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)

New scheduler:
- Preemptive: Potentially schedule different job at any point by taking CPU away from running job
- STCF (Shortest Time-to-Completion First)
- Always run job that will complete the quickest
NON-PREEMPTIVE: SJF

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>60</td>
</tr>
<tr>
<td>B</td>
<td>~10</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~10</td>
<td>10</td>
</tr>
</tbody>
</table>

Average turnaround time:

\[
(60 + (70 – 10) + (80 – 10)) / 3 = 63.3s
\]
Preemptive: STCF

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival_time (s)</th>
<th>run_time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>~0</td>
<td>60</td>
</tr>
<tr>
<td>B</td>
<td>~10</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>~10</td>
<td>10</td>
</tr>
</tbody>
</table>

Average turnaround time with STCF? **36.6**

Average turnaround time with SJF: **63.3s**
Scheduling Basics

**Workloads:**
- arrival_time
- run_time

**Scheduling Policies:**
- FIFO
- SJF (SJN, SPN)
- STCF
- RR

**Metrics:**
- turnaround_time
- response_time
Response Time

• Sometimes we care about when a job starts instead of when it finishes
• New metric:
  • response_time = first_run_time – arrival_time
Response vs. Turnaround

B’s turnaround: 20s
B’s response: 10s

[B arrives]
Round-Robin

**Prev schedulers:**
FIFO, SJF, and STCF can have poor response time

**New scheduler:** RR (Round Robin)
Alternate ready processes every fixed-length time-slice
FIFO vs RR

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Avg Response Time?

\[
(0+5+10)/3 = 5
\]

```
<table>
<thead>
<tr>
<th></th>
<th>ABC ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
```

Avg Response Time?

\[
(0+1+2)/3 = 1
\]

In what way is RR worse?

Ave. turn-around time with equal job lengths is horrible

Other reasons why RR could be better?

If don’t know run-time of each job, gives short jobs a chance to run and finish fast
Scheduling Basics

**Workloads:**
- arrival_time
- run_time

**Scheduling Policies:**
- FIFO
- SJF (SJN, SPN)
- STCF
- RR

**Metrics:**
- turnaround_time
- response_time
Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known
Don’t let Job A hold on to CPU while blocked waiting for disk
I/O Aware (Overlap)

Treat Job A as 3 separate CPU bursts
When Job A completes I/O, another Job A is ready

Each CPU burst is shorter than Job B, so with SCTF, Job A preempts Job B
Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

(Need smarter, fancier scheduler)
MLFQ  
(Multi-Level Feedback Queue)

**Goal:** general-purpose scheduling  
Must support two job types with distinct goals  
- “interactive” programs care about response time  
- “batch” programs care about turnaround time  

**Approach:** multiple levels of round-robin;  
each level has higher priority than lower levels and preempts them
Priorities

Rule 1: If priority(A) > Priority(B), A runs
Rule 2: If priority(A) == Priority(B), A & B run in RR

Q3 → A

Q2 → B

How to know how to set priority?

Q1

Approach 1: nice

Q0 → C → D

Approach 2: history “feedback”

“Multi-level”
History

• Use past behavior of process to predict future behavior
  • Common technique in systems
• Processes alternate between I/O and CPU work
• Guess how CPU burst (job) will behave based on past CPU bursts (jobs) of this process
More MLFQ Rules

Rule 1: If priority(A) > Priority(B), A runs
Rule 2: If priority(A) == Priority(B), A & B run in RR

More rules:
  Rule 3: Processes start at top priority
  Rule 4: If job uses whole slice, demote process
        (longer time slices at lower priorities)
One Long Job (Example)
An Interactive Process Joins

Interactive process never uses entire time slice, so never demoted
Problems with MLFQ?

- unforgiving + starvation
- gaming the system
Prevent Starvation

Problem: Low priority job may never get scheduled

Periodically boost priority of all jobs (or all jobs that haven’t been scheduled)
Prevent Gaming

**Problem:** High priority job could trick scheduler and get more CPU by performing I/O right before time-slice ends

**Fix:** Account for job’s total run time at priority level (instead of just this time slice); downgrade when exceed threshold
Programming Patterns: *The Bridge Pattern*

- Used to separate policy from mechanism
- More generally, *separate an implementation from its abstraction*
Gang of Four (GOF) Book
Proc * candidate = curr;
Schedule () {
    for (I = 0; I < NUM_PROCS; i++) {
        if (procs[i].priority > candidate) {
            candidate = procs[i];
        }
    }
}
switch_to(candidate);
The Bridge

Proc * next;
Schedule () {
    next = scheduler->policy->choose_next(sched_state);
    switch_to(next);
}
The Bridge

Proc * next;
Schedule () {
    next = scheduler->policy->choose_next(sched_state);
    switch_to(next);
}

/* this is the scheduler proper: */

while (1) {
    c = -1;
    next = 0;
    i = NR_TASKS;
    p = &task[NR_TASKS];
    while (--i) {
        if (!*(--p))
            continue;
        if (((*p)->state == TASK_RUNNING && (*p)->counter > c))
            c = (*p)->counter, next = i;
    }
    if (c) break;
    for (p = &LAST_TASK; p > &FIRST_TASK; --p)
        if (*p)
            (*p)->counter = (((*p)->counter >> 1) + (*p)->priority);
    switch_to(next);
TODO

• Work on project 1b! Due next Monday
• Do your reading, check out optional reading
  • Multiprocessor scheduling
  • Lottery Scheduling
  • Linux processes and scheduler