CS416 – CPU Scheduling

CS 416: Operating Systems Design, Spring 2011
Department of Computer Science
Rutgers University
Rutgers Sakai: 01:198:416 Sp11
(https://sakai.rutgers.edu)
Assumptions

- Pool of jobs contending for the CPU
  - CPU is a scarce resource

- Scheduler mediates between jobs to optimize some performance criteria
Process States

- New
- Admitted
- Interrupt
- Exit
- Terminated

- Ready
- Running
- Waiting

- I/O or event completion
- Scheduler dispatch
- I/O or event wait
Scheduling

- We have already Discussed Context Switching
  - Context Switching – Mechanism
  - Scheduling – Policy

- Which Thread to run next?

- How to ensure every thread gets a chance to run (Fairness)?

- How to prevent Starvation?

- Process Scheduling Vs Thread Scheduling: If the OS supports kernel level threads, threads are scheduled. If not, processes are scheduled.
  - We will use these terms interchangeably.
Scheduler

- Scheduler is the OS component that decides which thread to run next on the CPU

- The Scheduler operates on the ready queue
  - Why does it not deal with the I/O queues?

- When does the scheduler run?
  - When a thread exits
  - When a thread moves from ready queue to waiting queue (I/O, wait())
  - When a thread moves from waiting state to ready state (Completion of I/O)
  - When a thread moves from running state to ready state (Interrupt)

- Scheduling can be preemptive (forced context-switch) or non-preemptive

- Batch vs Interactive Scheduling
  - Batch: Non-Preemptive and No other jobs run if they block
  - Interactive: Preemptive and other jobs do run if they block
Job Behavior

1. load store
2. add store
3. read from file

- wait for I/O

store increment
index
write to file

- wait for I/O

load store
add store
read from file

- wait for I/O

- CPU burst
- I/O burst
- CPU burst
- I/O burst
- I/O burst
Multiprogramming Example

Process A

start | idle; input | idle; input | stop

Process B

start | idle; input | idle; input | stop

Time = 10 seconds
Multiprogramming Example (cont)

Process A

start idle; input idle; input stop A

Process B

start B idle; input idle; input stop B

Total Time = 20 seconds

Throughput = 2 jobs in 20 seconds = 0.1 jobs/second

Ave. Waiting Time = (0+10)/2 = 5 seconds
Multiprogramming Example (cont)

Process A

start     idle; input    idle; input    stop A
context switch to B

Process B

idle; input    idle; input    stop B

Throughput = 2 jobs in 11 seconds = 0.18 jobs/second

Ave. Waiting Time = (0+1)/2 = 0.5 seconds
Scheduling Goals

- Goal of a scheduling policy is to achieve some “optimal” allocation of CPU time in the system.

- Possible Goals
  - Maximize CPU Utilization (% of time the CPU is busy)
  - Maximize CPU Throughput (No. of jobs completed per second)
  - Minimize Turnaround time ($T_{job\_end} - T_{job\_start}$)
  - Minimize Waiting time (Total time spent Waiting on Queues)
    - *Which Queue?*
  - Minimize job Response time ($T_{first\_response} - T_{job\_start}$)

- These goals often conflict
  - Batch Systems: Maximize the Job throughput and minimize turnaround time
  - Interactive Systems: Minimize response times of interactive jobs (eg. Editors)
Starvation

Schedulers often try to eliminate Starvation

- e.g., If a high priority thread always gets to run before a low-priority thread
- We say the low priority thread is *starved*

Not all schedulers have this goal!

- Sometimes starvation is permitted to achieve other goals

Example: Real Time Systems

- Some threads run under a specific deadline
- In this case it is OK to starve other threads.
(Short-Term) CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
  - Long term scheduler: decide which processes should be swapped-in/out

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running.
Job Behavior

Two broad classes of processes: CPU bound and I/O bound

- CPU Bound:

  ![CPU Bound Diagram]

- I/O Bound

  ![I/O Bound Diagram]

- Examples of each Kind:
  - CPU Bound: Compiler, Number Crunching, games, MP3 encoder, etc
  - I/O Bound: Web browser, database engine, word processor, etc
First-Come-First-Served

- Jobs are scheduled in the order that they arrive
  - Also called FIFO

- Used only for batch scheduling
  - Jobs run to completion – Never blocks or gets context switched

- Jobs treated equally
  - NO Starvation!

- What's wrong with FCFS?
  - Short jobs get stuck behind long ones – Increases the waiting time, response time
**Example:**

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose that the processes arrive in the order: $P_1$, $P_2$, $P_3$

The Gantt Chart for the schedule is:

```
          P_1 | P_2 | P_3
0------|-----|-----|-----|
  24   |     |     |     |
  27   |     |     |     |
  30   |     |     |     |
```

Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

Average waiting time: $(0 + 24 + 27)/3 = 17$
Suppose that the processes arrive in the order $P_2, P_3, P_1$.

The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$

Average waiting time: $(6 + 0 + 3)/3 = 3$

Much better than previous case.

Convoy effect: short process behind long process

- Low CPU and I/O utilization
Round Robin (RR)

- Essentially FCFS with preemption
- A thread runs until it blocks or its **CPU quantum** expires.
- How to determine the ideal CPU quantum?
  - Quantum needs to be large compared to the context switch overhead
  - In modern systems, Quanta range from 10 to 100msec and CS time is < 10 μs

Waiting time for Job A: 8, Job B: 7, Job C: 8
Average Waiting Time = \((8+7+8)/3 = 7.66\) (Higher than SJF, Lower than FCFS)
Response Time is however the lowest!
Shortest Job First (SJF)

- Schedule Job with shortest expected CPU Burst
  - This is non-preemptive and will run until it blocks for I/O

- Idea:
  - Running short-CPU-burst jobs first gets them done, and out of the way.
  - Allows their I/O to overlap with each other: more efficient use of the CPU
  - Interactive programs often have a short CPU burst: Good to run them first

- How to predict a process’s CPU Burst?
  - Get a pretty good guess by looking at the history

1. $t_n = \text{actual length of } n^{th} \text{ CPU burst}$
2. $\tau_{n+1} = \text{predicted value for the next CPU burst}$
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define:

- We use exponential averaging $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$. 
Examples of Exponential Averaging

\( \alpha = 0 \)

\[ \tau_{n+1} = \tau_n \]

Recent history does not count.

\( \alpha = 1 \)

\[ \tau_{n+1} = t_n \]

Only the actual last CPU burst counts.

If we expand the formula, we get:

\[ \tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n-1} t_0 \]

Since both \( \alpha \) and \((1 - \alpha)\) are less than or equal to 1, each successive term has less weight than its predecessor.
SJF Example

Resulting schedule:

- **Job B**: i/o
- **Job A**: i/o
- **Job B**: i/o
- **Job A**: i/o
- **Job C**: i/o

*B is not on the ready queue!
Shortest Remaining Time First (SRTF)

- SJF is non-preemptive policy
- Preemptive variant: **Shortest Remaining Time First (SRTF)**
  - If a job becomes runnable with a shorter expected CPU burst, preempt current job and run the new job
SRTF vs RR

Say we have three jobs:
- Job A and B: both CPU-bound, will run for hours on the CPU with no I/O
- Job C: Requires a 1ms burst of CPU followed by 10ms I/O operation

RR with 25 ms time slice:

RR with 1 ms time slice:
- Lots of pointless context switches between Jobs A and B!

SRTF:
- Job A runs to completion, then Job B starts
- C gets scheduled whenever it needs the CPU
Priority Scheduling

- A priority number (integer) is associated with each process
  - Can be set by User/OS or combination of two.

- The CPU is allocated to the process with the highest priority (smallest integer = highest priority).
  - Preemptive: Whenever higher priority process comes, lower priority process gets preempted.
  - Non-preemptive: Puts the higher priority process at the head of the queue

- SJF is a priority scheduling where priority is the predicted next CPU burst time.

- Problem: Starvation – low priority processes may never execute.

- Solution: Aging – as time progresses increase the priority of the process.
Multi-Level Queue

highest priority

- system processes

interactive processes

interactive editing processes

batch processes

student processes

lowest priority
Multi-Level Queue

- Ready queue is partitioned into separate queues:
  - Could be one queue for each priority level
- Each queue has its own scheduling algorithm,
- Scheduling must be done between the queues.
- Example: 2 Priority Levels (0 -> Foreground, 1-> Background)
  - Fixed priority scheduling: serve all foreground processes and then serve background processes. Possibility of starvation.
  - Time slice: Each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e.,
    - 80% to foreground in RR
    - 20% to background in FCFS
Multi-Level Feedback Queue (MLFQ)

quantum = 8

quantum = 16

FCFS
Multi-Level Feedback Queue (MLFQ)

- **Observation:** Want to give *higher priority to I/O-bound jobs*
  - They are likely to be interactive and need CPU rapidly after I/O completes
  - However, jobs are not always I/O bound or CPU-bound during execution!
    - *Web browser is mostly I/O bound and interactive but, becomes CPU bound when running a Java applet*

- **Basic idea:** Adjust priority of a thread in response to its CPU usage
  - Increase priority if job has a short CPU burst
  - Decrease priority if job has a long CPU burst (e.g., uses up CPU quantum)
  - Whenever processes with higher priority arrives, **Preempt the lower priority job**
  - Jobs with lower priorities get longer CPU quantum

- **What is the rationale for this??**
  - Don't want to give high priority to CPU-bound jobs...
    - Because lower-priority jobs can't preempt them if they get the CPU.
  - OK to give longer CPU quantum to low-priority jobs:
    - I/O bound jobs with higher priority can still preempt when they become runnable.
MLFQ Implementation
MLFQ Implementation

- High prio
  - PID 4391, T2
  - State: Ready
  - FC
  - Registers
  - Uses entire CPU burst (preempted)
  - Placed into lower priority queue

- Medium prio
  - PID 3202, T1
  - State: Ready
  - FC
  - Registers

- Low prio
  - PID 4277, T0
  - State: Ready
  - FC
  - Registers
MLFQ Implementation

High prio

Medium prio

Low prio
MLFQ Implementation

High prio

Medium prio

Low prio

Preempted
MLFQ Implementation

- **High prio**
  - PID 3202, T1
  - State: Ready
  - PC
  - Registers
  - Runs with short CPU burst (blocks on I/O)

- **Medium prio**
  - PID 4277, T0
  - State: Ready
  - PC
  - Registers

- **Low prio**
  - PID 4391, T2
  - State: Ready
  - PC
  - Registers