Programming Assignment #2

- DUE DATE: Monday, October 1 2018

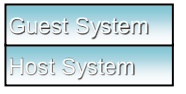
MINI-HOMEWORK DEADLINE: Tuesday Sep 25

- Download assignment files
- Build the Linux kernel
- Boot the QEMU image with the Linux kernel
- Report output of the procedure
  - uname -a

1

About QEMU

- A processor emulator
- Emulates code for target CPU on host CPU



2

Why QEMU?

- Allow you to change the kernel
  - OS requires privilege to overwrite kernel file
- Isolate kernel changes from the real machine
- Make programming and debugging easier

3

Download the Code

- Get the files you need:
  - Use git clone
  - Approximate space requirements
  - Kernel source and object code: 4 GB
  - Everything: About 4 GB of disk space

4

Get It Started!

- Start QEMU and default Debian installation
  - cd install
  - sh runqemu.sh
- No password
- Use poweroff command to shutdown
- Can kill qemu from different window if kernel panics

5

Start Up a New Kernel

- Specify path to your kernel bzImage to runqemu.sh
  - sh runqemu.sh linux-3.18-77/arch/x86_64/boot/bzImage
- When QEMU boots, use uname -a to check if you booted the correct kernel image
- README.md file contains useful information

6

CPU Scheduling

CS 256/456
Department of Computer Science
University of Rochester

9/20/2018 CSC 2/456 7

CPU Scheduling

- Selects from among the processes/threads that are ready to execute, and allocates the CPU to it
- CPU scheduling may take place at:
  1. Hardware interrupt/software exception
  2. System calls
- *Nonpreemptive:*
  - Scheduling only when the current process terminates or not able to run further
- *Preemptive:*
  - Scheduling can occur at any opportunity possible

9/20/2018 CSC 2/456 8

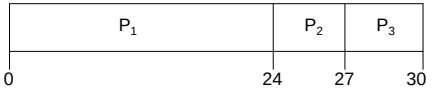
Scheduling Criteria

- Minimize turnaround time – amount of time to execute a particular process (includes I/O, CPU, memory time, waiting time in the ready queue)
- Maximize throughput – # of processes that complete their execution per time unit
- Maximize CPU utilization – the proportion of the CPU that is not idle
- Minimize response time – amount of time it takes from when a request was submitted until the first response is produced (interactivity)
- Waiting time: time spent in the ready queue
- Fairness: avoid starvation

First-Come, First-Served (FCFS) Scheduling

Process	CPU Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The schedule is:



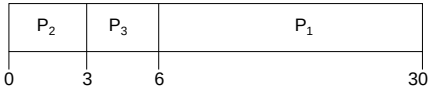
- Turnaround time for P_1 = 24; P_2 = 27; P_3 = 30
- Average turnaround time: (24 + 27 + 30)/3 = 27

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1 .

- The schedule is:



- Turnaround time for P_1 = 30; P_2 = 3, P_3 = 6
- Average turnaround time: (30 + 3 + 6)/3 = 13
- Much better than previous case.
- Short process delayed by long process: *Convoy effect*

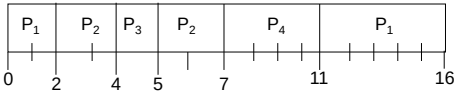
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its CPU time. Use these lengths to schedule the process with the shortest CPU time
- Two variations:
  - Non-preemptive – once CPU given to the process it cannot be taken away until it completes
  - preemptive – if a new process arrives with CPU time less than remaining time of current executing process, preempt
- Preemptive SJF is optimal – gives minimum average turnaround time for a given set of processes
- Problem:
  - don't know the process CPU time ahead of time

Example of Preemptive SJF

Process	Arrival Time	CPU Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)

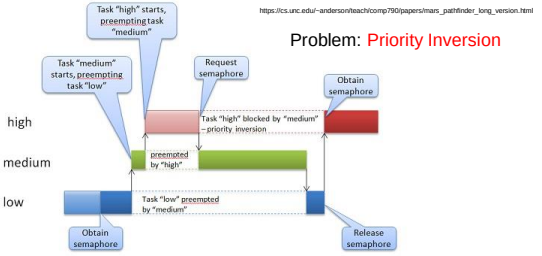


- Average turnaround time = (16 + 5 + 1 +6)/4 = 7

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted CPU time
- Problem: Starvation – low priority processes may never execute
- Solution: Aging – as time progresses, increase the priority of the process

What Happened on the Mars Pathfinder (1997)?



https://cs.unc.edu/~anderson/teach/comp790/papers/mars_pathfinder_long_version.html

Problem: Priority Inversion

https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

Solution: Priority Inheritance [*L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.]*

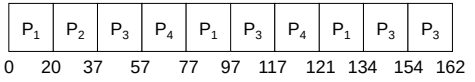
Round Robin (RR)

- Each process gets a fixed unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units
- Performance
  - q small $\Rightarrow$ fair, starvation-free, better interactivity
  - q large $\Rightarrow$ FIFO
  - q must be large with respect to context switch cost, otherwise overhead is too high

Example of RR with Quantum = 20

Process	CPU Time
P_1	53
P_2	17
P_3	68
P_4	24

- The schedule is:



- Typically, higher average turnaround than SJF, but better *response*

Multilevel Scheduling

- Ready tasks are partitioned into separate classes:
  foreground (interactive)
  background (batch)
- Each class has its own scheduling algorithm,
  foreground – RR
  background – FCFS
- Scheduling must be done between the classes.
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation
  - Time slice – each class gets a certain amount of CPU time which it can schedule amongst its processes; e.g.,
    - 80% to foreground in RR
    - 20% to background in FCFS

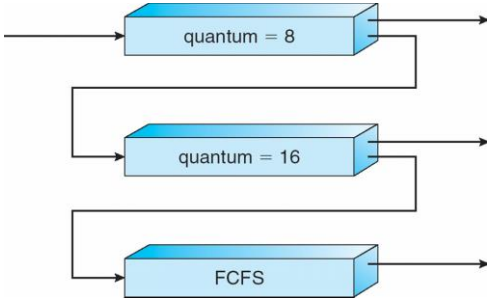
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Lottery Scheduling

- Give processes lottery tickets for various system resources
- Choose ticket at random and allow process holding the ticket to get the resource
- Hold a lottery at periodic intervals
- Properties
  - Chance of winning proportional to number of tickets held (highly responsive)
  - Cooperating processes may exchange tickets
  - Fair-share scheduling easily implemented by allocating tickets to users and dividing tickets among child processes

Real-Time Scheduling

- *Hard real-time systems* – required to complete a critical task within a guaranteed amount of time
- *Soft real-time computing* – requires that critical processes receive priority over less fortunate ones
- EDF – Earliest Deadline First Scheduling

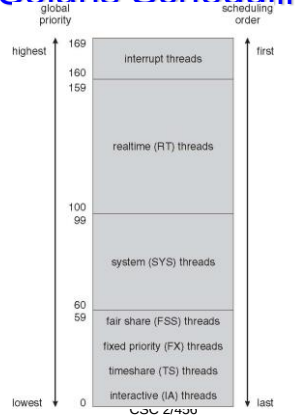
Cost of Context Switch

- Direct overhead of context switch
  - saving old contexts, restoring new contexts, … …
- Indirect overhead of context switch
  - caching, memory management overhead

Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

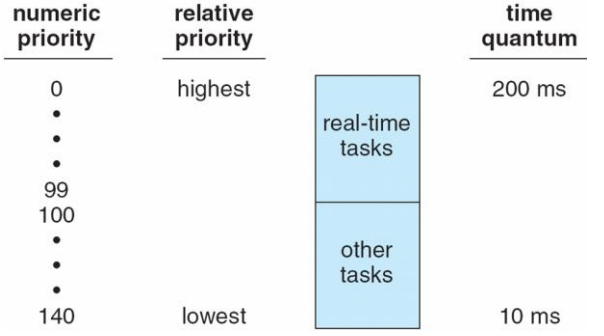
Solaris Scheduling



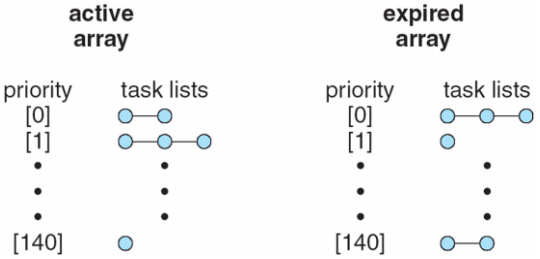
Linux Task Scheduling

- Linux 2.5 and up uses a preemptive, priority-based algorithm with two separate priority ranges:
  - A time-sharing class/range for fair preemptive scheduling (nice value ranging from 100-140)
  - A real-time class that conforms to POSIX real-time standard (0-99)
- Numerically lower values indicate higher priority
- Higher-priority tasks get longer time quanta (200-10 ms)
- One runqueue per processor (logical or physical); load balancing phase to equally distribute tasks among runqueues
- Runqueue indexed by priority and contains two priority arrays – **active** and **expired**
- Choose task with highest priority on active array; switch active and expired arrays when active is empty
- Time-sharing tasks are assigned the nice value +/- 5

Priorities and Time-slice length



List of Tasks Indexed According to Priorities



Multiprocessor Context Switch

- Disabling signals not sufficient
- Acquire scheduler lock when accessing any scheduler data structure, e.g.,

yield:
  disable_signals
  acquire(scheduler_lock) // spin lock
  enqueue(ready_list, current)
  reschedule
  release(scheduler_lock)
  re-enable_signals

CPU Scheduling on Multi-Processors

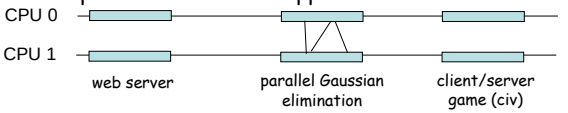
- Cache affinity
  - keep a task on a particular processor as much as possible
- Resource contention
  - prevent resource-conflicting tasks from running simultaneously on sibling processors

Multiprocessor Scheduling in Linux 2.6

- One ready task queue per processor
  - scheduling within a processor and its ready task queue is similar to single-processor scheduling
- One task tends to stay in one queue
  - for cache affinity
- Tasks move around when load is unbalanced
  - e.g., when the length of one queue is less than one quarter of the other
  - which one to pick?
- No native support for gang/cohort scheduling or resource-contention-aware scheduling

Multiprocessor Scheduling

- Timesharing
  - similar to uni-processor scheduling – one queue of ready tasks (protected by synchronization), a task is dequeued and executed when a processor is available
- Space sharing
- cache affinity
  - affinity-based scheduling – try to run each process on the processor that it last ran on
- caching sharing and synchronization of parallel/concurrent applications
  - gang/cohort scheduling – utilize all CPUs for one parallel/concurrent application at a time



Anderson et al. 1989 (IEEE TOCS)

- Raises issues of
  - Locality (per-processor data structures)
  - Granularity of scheduling tasks
  - Lock overhead
  - Tradeoff between throughput and latency
    - Large critical sections are good for best-case latency (low locking overhead) but bad for throughput (low parallelism)

Disclaimer

- Parts of the lecture slides were derived from those by Kai Shen, Willy Zwaenepoel, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).