Scheduling - Overview

- Quick review of “textbook” scheduling
- Linux 2.4 scheduler implementation overview
- Linux 2.4 scheduler code
- Modified Linux 2.4 scheduler
- Linux 2.6 scheduler comments

Possible Goals of a Scheduler

- Fairness: each process gets fair share of CPU
- Efficiency: keep the CPU busy
- Response time: minimize response time for interactive users
- Turnaround: minimize the time batch users must wait for output
- Throughput: maximize the number of jobs processed per time unit
- **These goals are contradictory, therefore there can be no “one scheduler to rule them all.”**

How Does the Scheduler Know When To Run?

- When the scheduler starts a process running, it doesn't know how long until the process will suspend itself or finish.
- This problem is typically solved with a built in timer which causes an interrupt periodically.
- This is called **preemptive scheduling**

Quick Review: First Come First Serve (FCFS)

- Uses a FIFO queue
- Runnable processes are inserted at the back of the queue
- Processes at the front of the queue are selected to run on the CPU
- Control of the CPU is only relinquished voluntarily (process goes to sleep or finishes)
- Consider the situation where a short process is queued right behind a long process.

Quick Review: Round Robin (RR)

- FIFO queue used
- Each process is given a time to run called a **quantum**
- If a process is still running when its time quantum is over, the process is put at the back of the queue and the CPU is given to the next process
- The time quantum must be chosen carefully to avoid spending too much time switching between processes
  - Wasted CPU time vs. Response time

Quick Review: Priority Scheduling (PS)

- *Question:* Are all processes equally important?

Quick Review: Priority Scheduling (PS)

- *Question:* Are all processes equally important?
- *Answer:* In general, no. (eg: Interactive processes may need immediate attention)
- Each process is assigned a priority. The runnable process with the highest priority is chosen to run.
- To prevent a high priority process from starving all other processes, the scheduler often adjusts the priority of a process over time

Linux 2.4 Scheduler

Ways to Classify Processes

- CPU bound vs I/O bound
  - Compiling vs Database system
  - Linux 2.4 favors I/O bound processes
- Interactive vs Batch vs Real time
  - Editor vs Scientific computations vs streaming video
- Linux 2.4 does not distinguish between interactive and batch processes, but it does recognize real time processes.
  - Linux 2.4 is not suitable for real time apps with hard real time constraints

Process Preemption

- A process can be interrupted if another process with a higher priority enters the TASK_RUNNING state.
  - This may only happen if the process is not running in the kernel. The Linux 2.4 kernel is NOT preemptive
- A process is also considered to preempted when its time quantum expires
- A preempted process is NOT “suspended” (it is still in the TASK_RUNNING state)

Epochs and Quantums

- CPU time is divided into epochs
- Each process has a specified quantum (max CPU time per epoch).
- Each process' time quantum is computed at the start of each epoch.
  - *Question: What does this mean concerning the time complexity of the Linux 2.4 scheduler?*

Epochs and Quantums

- When a process has used its time quantum, it's preempted
- A process can be selected to run many times per epoch, provided that it hasn't already used up its time quantum.
  - This could happen if a process puts itself to sleep to wait for I/O. In fact, the Linux kernel rewards this behavior
- The epoch ends when all processes have used up their time quantum.

Priority

- Static – not changed by the scheduler
  - Real time processes
- Dynamic – changed by scheduler over time
  - Essentially, base priority + remaining time quantum
- Any static priority is considered “higher” than any dynamic priority, meaning real time processes are only in competition with each other.

Linux 2.4.20 Sourcecode

original:
http://lxr.linux.no/source/kernel/sched.c#L547?v=2.4.20

annoated schedule() function:
annotated_functions.html

When is schedule() called?

- The schedule() function can be invoked in two ways.
  - Direct Invocation
  - Lazy (Deferred) Invocation

schedule() - direct invocation

- The scheduler will be invoked directly when a process needs to be blocked because a resource is not available. In this case, the process will insert itself into the proper wait queue (for the resource that it wants) and invoke the schedule() function to yield the CPU to another process.
-
- When the process is again selected to run it will check to see if the resource is available. If so, the process continues. If not, the process again calls schedule().

schedule() - direct invocation

- schedule() is also directly invoked by device drivers that perform long, iterative tasks. In this case, at the beginning of each iteration, the process checks if any other processes want the CPU and, if deemed necessary, invokes schedule().

schedule() - lazy invocation

- Lazy invocation is probably responsible for most calls to the schedule() function. This occurs when the needs_resched field of a process is set to 1. Understanding the Linux Kernel says, "Since a check on the value of this field is always made before resuming the execution of the User Mode process, schedule() will definitely be invoked at some time in the near future."

schedule() - lazy invocation

- This flag would be set, for example, by a process that has higher priority than the currently executing one (on wake up). It seems to me that this scheme will only work on an honor system based approach (like most OS components), meaning that the need_resched flag is only set under a very strict set of conditions.

Linux 2.6 Scheduler

2.4 Scheduler Weaknesses

- Time to schedule a job increases linearly with the number of jobs on the system – O(n)
- Run queue is shared among all the CPUs
  - Only 1 CPU at a time may access the runqueue
  - Causes processes to be “bounced” between CPUs which is wasteful because caching can not be taken advantage of (affinity problem).

2.6 Scheduler to the Rescue

- No more global run-queue, one per processor
  - Each processor can now schedule processes simultanously
- Run queue is now composed of two queues, an active queue and an expired queue
  - When a process' quantum expires its new quantum is computed, and it is placed in the expired queue
  - When the epoch is over (all process' time quantums have expired), the expired queue becomes the active queue and the active queue becomes the expired queue.
  - This scheduler is now O(1)!

But Wait! There's More!

- In order to allow the system to respond faster to interactive and real time jobs the kernel is now preemptible.
  - There are still critical sections in the kernel that can not be interrupted.
    - These are enclosed by preempt_disable() and preempt_enable()

Where Did All This Info Come From?

- *Modern Operating Systems*
- -http://iamexwiwww.unibe.ch/studenten/schlpbch/linuxScheduling/LinuxScheduling-2.html
- -http://www-306.ibm.com/software/tivoli/features/ccr2/ccr2-2004-02/features-scheduler.html
- *Understanding the Linux Kernel*
- -http://developer.osdl.org/craiger/hackbench/