LECTURE 1.2
CPU SCHEDULING BASICS

COP4600

Dr. Matthew Gerber

1/27/2016

Fundamentals (6.1)

- A system can only run as many processes at once as it has processors
- If processes just idle while waiting, CPU time is wasted
- We only want processes to hold the CPU until they have to wait
- Hence, multiprogramming
- Processes run until they:
  - Have to *wait*
  - Are *terminated*
  - *Yield*
  - Are *preempted* (more on this in a bit)
- Other processes can then take a turn
- Most of the time we say “processes” here, you can also insert “threads”
  - We’ll get into some particular issues with thread scheduling

The Burst Cycle (6.1.1)

- Processes alternate bursts of CPU activity and I/O waits
- Types of processes differ in their burst patterns
- *I/O bound* processes have relatively short CPU bursts interspersed with I/O waits
- *CPU bound* processes have relatively long CPU bursts and less I/O waiting
- Which kind of processes you have can affect the best scheduling algorithm to use
- Keep in mind that *what we are actually scheduling* is not a process, but *the next CPU burst* for that process

Preemptive Scheduling (6.1.3)

- For reasons we've already discussed, we don't generally allow processes to just run as long as they feel like
- Under non-preemptive, or cooperative multitasking, processes run until they:
  - Have to *wait* (on I/O)
  - Are *terminated* (by themselves or by the OS)
  - *Yield* (voluntarily give another process a turn)
- Under preemptive multitasking, they also stop if they:
  - Are *preempted* (by the timer interrupt)
- We will discuss some non-preemptive, or *cooperative* multitasking algorithms, but they're basically only of theoretical interest in the modern era

The CPU Scheduler (6.1.2-6.1.4)

- The *CPU scheduler* selects a process from the ready queue to run.
- The ready queue is *conceptually* a queue - it may not actually be a FIFO queue.
- The scheduler runs when:
  - A running process waits (I/O call)
  - A running process becomes ready (yield or preemption)
  - A waiting process becomes ready (I/O completion)
  - A process terminates
- Note that under the second and third, the scheduler may or may not decide to actually change which process is running

The CPU Scheduler (6.1.2-6.1.4)

- The *dispatcher* is the part of the CPU scheduler that actually does the business of context switching once the new process is selected
- Performance of the dispatcher is just as important as performance of the scheduling algorithm
- The time the dispatcher takes to stop one process and start another is called the *dispatch latency*
- The total time taken every time the CPU scheduler runs is the dispatch latency plus the time taken by the actual scheduling algorithm

Scheduling Criteria (6.2)

- What are scheduling algorithms trying to achieve?
- CPU Utilization
  - We want to keep the CPU(s) as busy as possible
  - Can range from 0-100%
  - For an optimally loaded system, you want 40-90%
  - Optimizing CPU utilization avoids wasting hardware time via lack of use or via overload and its corresponding overhead
- Throughput
  - The number of processes completed per time unit
  - Originally, and obviously, a figure used for batch-processing systems
  - Still perfectly adaptable to many tasks today, especially if you think of it in terms of CPU bursts instead of processes
  - Maximizing throughput results in the highest number of complete processes getting done in a given time period

Scheduling Criteria (6.2)

- Turnaround Time
  - The process-individualistic version of throughput
  - For a given process, how long it takes to execute that process, from start to termination
  - Also originated in back processing days, but also applicable to modern tasks if you think of it flexibly
  - Minimizing turnaround time results in the best experience for users waiting on (whole) processes
- Waiting Time
  - The amount of time a process is ready without running
  - Since waiting time is the only component of a process’s total run time that the CPU scheduler directly affects, it is considered a good overall measure of the efficiency of the CPU scheduler in specific

Scheduling Criteria (6.2)

- Response Time
  - The time between the submission of a request *by a user* and the first output *to that user*
  - Fundamentally irrelevant to overall scheduler performance – in fact, working to minimize response time has a negative impact on waiting time and throughput
  - ***Easily the most important measure in terms of user satisfaction for most tasks***

Multiple-Processor Scheduling (6.5)

- Analogous to single-processor scheduling, but raises a few questions
- The first question is whether to have one processor do the scheduling or allow each processor to be an equal participant
  - Almost all modern systems do the latter, known as *symmetric multiprocessing*
- The CPU scheduler is run by, and for, each processor
- Each processor selects its own process from the ready queue to execute
- This means we can use pretty much the same algorithms
- That doesn’t mean there aren’t issues

Multiple-Processor Scheduling (6.5)

- Need to avoid multiple processors choosing the same process to run
  - This is fundamentally a synchronization issue, just at a *very* fundamental level of the system
- Need to avoid moving processes between processors
  - Doing so can invalidate large chunks of cache memory at once
  - The concept of keeping a process (at least mostly) on the same processor it started on is called *affinity*
- Single ready queue, or ready queue per processor?
  - Most systems use the latter
- If a ready queue per processor, how to maintain a balanced load?
  - Under-loaded processors may *pull* processes *from* other processors
  - Overloaded processors may *push* processes *to* other processors

Thread Scheduling (6.4)

- Scheduling threads is mostly like scheduling processes
- The main difference is the concept of *contention scope*
  - Do we schedule threads just as we would schedule processes, hence using a *system contention scope*?
  - Or do we schedule processes then schedule threads within them, hence using a *process contention scope*?
- This is only really relevant on systems that implement threads in user space
  - For performance reasons, this is now rare
- Windows, Linux, Solaris and Mac OS X all use system contention scope

Processor-Level Threading (6.5.4)

- Some processors – especially multi-core processors – present one physical processor as multiple logical processors, allowing multiple threads to be “running” on it
- The processor then decides internally how to schedule those threads, running a second-level, simpler CPU scheduling algorithm of its own
- Intelligent use of this capability can overcome some of the “micro-waits”, or *stalls*, involved in memory access
- Naïve use of this capability can greatly decrease performance by scheduling two processor-intense threads on the same processor
- More detailed discussion of this capability spans multiple advanced OS, architecture and optimization issues, and is *far* beyond the scope of this course

Real-Time Scheduling (6.6)

- Real-time scheduling is divided into two different classes
- *Soft real-time* scheduling guarantees only that real-time processes get priority over non-real-time processes
- *Hard real-time* scheduling actually attempts to guarantee that a task will be serviced by its deadline
  - Failure to service a task by deadline is considered complete failure
- Hard real-time systems run the gamut from radios to life-safety-critical systems

NEXT TIME: SCHEDULING ALGORITHMS