Scheduling

CS 161: Lecture 4
2/9/17
Where does the first process come from?

BRACE YOURSELVES...

STORY TIME IS COMING
The Linux Boot Process

Machine turned on; BIOS runs

- BIOS: Basic Input/Output System
- Stored in flash memory on motherboard
- Determines which devices are available, loads Master Boot Record (MBR) of the bootable storage device into RAM, jumps to it

<table>
<thead>
<tr>
<th>Metadata</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition 1</td>
<td>446 bytes</td>
<td>Partition 1 metadata</td>
</tr>
<tr>
<td>Partition 2</td>
<td>64 bytes</td>
<td>Partition 2 metadata</td>
</tr>
<tr>
<td>Partition 3</td>
<td></td>
<td>Partition 3 metadata</td>
</tr>
<tr>
<td>Partition 4</td>
<td></td>
<td>Partition 4 metadata</td>
</tr>
<tr>
<td>Magic number</td>
<td>2 bytes</td>
<td>Magic number: 0x55AA</td>
</tr>
<tr>
<td>Remaining Sectors</td>
<td></td>
<td>Remaining Sectors</td>
</tr>
</tbody>
</table>
The Linux Boot Process

• The stage 2 bootloader loads the OS kernel into RAM and then jumps to its first instruction
  • The stage 2 bootloader code is larger than a single sector, so it can do fancy things
    • Ex: Present user with a GUI for selecting one of several kernels to load
• The kernel starts running and does low-level system initialization, e.g.,
  • Setting up virtual memory hardware
  • Installing interrupt handlers
  • Loading device drivers
  • Mounting the file system
• But how do user-level processes get started?
The first process that runs

- **init**
  - The first process that runs
  - Responsible for launching all other processes via `fork()+exec()`
    - Desktop window manager
    - **sshd**
    - Printer daemon
  - Reaps all zombie processes whose parents did not `wait()` on them

*Daemon processes run in the background, without being directly controlled by a user via a GUI*

 Or the equivalent on your Unix-like system, e.g.,

- **systemd** on many Linux distros (Ubuntu, Debian, Fedora)
- **launchd** on Mac OS
Scheduling: Which Process Should Run Now?

- Different processes have different behaviors
  - IO-bound: A process mostly waits for IOs to complete
  - CPU-bound: A process issues few IOs, mostly does computation
  - A process may change its behavior throughout its execution—the scheduler must notice and adjust!

- Often a good idea to prioritize IO-bound processes
  - If IO comes from user (e.g., keyboard, mouse), we want interactive programs to feel responsive
  - Network IO may take tens or hundreds of milliseconds (Comcast! Verizon!)
  - IO is typically slow, so start it early!
Inside a Computer

Socket 0

L1 i-cache  L1 d-cache  L1 i-cache  L1 d-cache
L2 cache
L3 cache

Socket 1

L1 i-cache  L1 d-cache  L1 i-cache  L1 d-cache
L2 cache
L3 cache

RAM
IO Is Usually Slow: Start It Early!

1 CPU cycle (1 register access): 0.3 ns
L1 cache access: 0.9 ns
L2 cache access: 2.8 ns
L3 cache access: 12.9 ns
RAM access: 120 ns
SSD access: 50—150 µs
Disk access: 5—10 ms
Network RTT: 10—500 ms
User input: 200 ms—seconds
Mechanism versus Policy

• Policy: A high-level goal (e.g., “Prioritize IO-bound tasks”)

• Mechanism: The low-level primitives that are used to implement a policy
  • Ideally, a single set of mechanisms are sufficiently generic to support multiple policies
  • Designing a minimal (but expressive) set of mechanisms is often tricky!

• Basic scheduling mechanisms
  • Run queue: the set of threads that are ready to execute on the CPU
  • Wait channel: a set of threads that are waiting for an event to occur
  • Traps: opportunities for the OS to run and make a scheduling decision
First-Come, First-Serve (FCFS)

• Basic idea: Run a task until it’s “finished”
  • “Finished” is typically defined as “willingly blocks” (e.g., due to an IO request)
  • The blocked task is placed in the relevant wait queue
  • When a task unblocks, it is placed at the end of a single FIFO ready queue

• Advantages
  • Enables parallel use of the CPU and IO devices
  • Simple!

• Disadvantages
  • Seems unfair AF: A single CPU task can monopolize the processor!
“AF” means “As Fuzz”
For example:

THAT FUZZ IS IRRITATING AS F**K
Response Times

• Ideally, a scheduler would maximize both CPU utilization and IO device utilization

• So, we should overlap computation from CPU-bound jobs with IO from IO-bound jobs
  • Important consequence: When IO-bound jobs are ready to use the CPU, we should prioritize those jobs (i.e., minimize the response time needed to assign them to a core)
  • Otherwise, devices lay idle: a sadness
Ex: An IO-bound disk grep, and a CPU-bound crypto calculation on a single-core machine

- **FCFS screws over IO-bound tasks**

- **CPU and disk at almost full utilization!**
Round-Robin

• Insight: After a task has run for a while, the OS should forcibly preempt the task
  • Time slice: the maximum amount of time that a task can run before being taken off the CPU
  • Timer interrupts provide a convenient mechanism to enforce time slices

• If a task is forcibly preempted, it goes at the end of the ready queue
  • Voluntary blocking places the task in the appropriate wait queue

• Advantages:
  • CPU-bound tasks must share the processor
  • No starvation!

• Subtlety: What’s the timer period?
• Problem: What if some tasks are more important than other tasks?
Priority-based Round-Robin

- Insight: Maintain several ready queues, one for each priority level
  - Each queue is FIFO
  - Scheduler finds highest-priority non-empty queue and runs the first task in that queue
- Advantage: Allows higher-priority tasks to receive more CPU time
- Problem: Low-priority tasks may starve!
  - Solution: aging (the longer a priority waits without getting the CPU, the higher its priority becomes)
  - We’ll discuss specific aging approaches next lecture!
- Related problem: IO-bound tasks may suffer if not given high priorities
- Problem: priority inversion
Priority Inversion

- Assume that a system has three tasks T1, T2, and T3
  - Priority: T3 > T2 > T1
- Imagine that T1 and T3 both use the same lock . . .

T3 is the highest priority thread, but it’s blocked by T2 and T1!
Priority Inheritance

A task which owns a lock inherits the highest priority of any task that wishes to acquire the lock.
Shortest Time to Completion First (STCF)

- Goal: Minimize the amount of time that a runnable task has to wait before it actually runs
  - Define “completion time” as the length of a task’s next CPU burst
  - Scheduler estimates each runnable task’s completion time (e.g., using the average length of the task’s recent CPU bursts)
  - Scheduler keeps a single run queue sorted by estimated completion time
  - The front of the queue gets to run next

- STCF can be used with or without preemption
  - Non-preemptive STCF: Once a task is running, it does not relinquish the CPU until the CPU burst is finished
  - Preemptive STCF: The currently-running task can be kicked off the CPU if a new task arrives with a shorter burst time
<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival time</th>
<th>Burst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>T1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>T2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>T3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Non-preemptive

0 7 8 12 16
Shortest Time to Completion First (STCF)

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival time</th>
<th>Burst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>T1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>T2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>T3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

STCF minimizes average response time, but, unlike RR, does not prevent starvation! So, aging is necessary.

T1 shows up and has shortest burst (4 vs 5)

T2 shows up and has shortest burst (1 vs 2)

T2 done

T0: 5  T1: 2  T3: 4
Context switches are pure overhead!

- Direct cost: CPU cycles devoted to bookkeeping
  - Save and restore registers
  - Invoke scheduler logic
  - Switch address spaces from old process to new process

- Indirect costs
  - L1/L2/L3 caches are polluted by kernel code and data; new task must warm the caches with its code and data
  - TLB entries become invalid
CPU Affinity

struct task_struct{ //From the Linux kernel
    volatile long state; //TASK_RUNNING,
    //TASK_ZOMBIE,
    //etc.
    void *stack;         //Kernel stack
    int exit_code;
    struct mm_struct *mm; //Address space info
    unsigned long cpus_allowed;
        //Bitmask representing which
        //cpus the task can run on
};
...
CPU Affinity

T1: Read data

Socket 0

L1 i-cache | L1 d-cache | L1 i-cache | L1 d-cache
L2 cache | L2 cache
L3 cache

Socket 1

L1 i-cache | L1 d-cache | L1 i-cache | L1 d-cache
L2 cache | L2 cache
L3 cache

RAM
CPU Affinity

- L1 i-cache
- L1 d-cache
- L2 cache
- L3 cache

Socket 0

Socket 1

T1: Read data

T2: Read data

RAM
CPU Affinity

Socket 0
- L1 i-cache
- L1 d-cache
- L2 cache
- L3 cache

T1: Read data

Socket 1
- L1 i-cache
- L1 d-cache
- L2 cache
- L3 cache

T2: Write data

INVALIDATE

T1: Read data

RAM
CPU Affinity

Socket 0
- L1 i-cache
- L1 d-cache
- L2 cache
- L3 cache

Socket 1
- L1 i-cache
- L1 d-cache
- L2 cache
- L3 cache

T1: Read data
T2: Write data
INVALIDATE DONE
CPU Affinity

Socket 0

T1: Read data

Socket 1

T2: Write data

L1 i-cache | L1 d-cache | L1 i-cache | L1 d-cache
L2 cache | L2 cache | L2 cache | L2 cache
L3 cache | L3 cache

RAM
Your Machine is a Distributed System!

- Components are connected by a network
  - Some components talk directly (e.g., core/registers)
  - Others require multiple hops to communicate (e.g., core and L3 cache; two cores on different sockets)
  - More hops = more communication latency!

- Ideally, the OS scheduler can:
  - Avoid network latencies by co-locating related threads on the same subset of cores (or at least on the same socket)
  - Keep all of the cores utilized (to avoid convoy effects on a small set of highly-utilized cores)