EECS-343 Operating Systems
Lecture 9:
Midterm Review

Steve Tarzia

Spring 2019

Announcements

- Project 2 deadline was extended one week (due Monday)
- HW2 is out and due on Wednesday.
- HW1 solutions have been posted.
- Midterm exam is on Thursday.
- Practice Midterm solution Q2 had an error. New solutions are posted.

Operating systems roles

- A ***user interface*** for humans to run programs
- A ***resource manager*** allowing multiple programs to share one set of hardware.
- A ***programming interface*** (API) for programs to access the hardware and other services. System calls are the OS API.

Processes & System Calls

- **Process** is a program in execution
- **Limited direct execution** is a strategy whereby a process usually operates as if it has full use of the CPU & memory.
- CPUs have user and kernel **modes** to prevent user processes from running privileged instructions, thus *limiting* execution.
- **Interrupts** are events that cause the kernel to run
- **System Calls** (or traps) are software interrupts called by a user program to ask the OS to do something on its behalf.
- **Timer Interrupt** ensures that the kernel eventually runs.

Process Creation

- xv6 OS code is written for the Intel x86 CPU architecture, but…
- Linux supports 31 different CPU architectures
  - Low-level ***mechanisms*** are different on each architecture.
  - High-level ***policies*** are the same for all.
- ***Fork*** syscall: run once, exits twice!
- ***Nondeterminism*** is when a program's output is unpredictable
- OS process scheduler can create ***race conditions*** in programs that rely on an interaction of multiple processes.
  - These are tricky to debug, because they are sensitive to timing (***Heisenbugs***).
- ***Kernel panic*** occurs when OS causes an exception and can't recover

Process Scheduling

- Defined two conflicting metrics: ***turnaround time*** and ***response time***
  - Cannot optimize both – must tradeoff, or balance, the two
- Optimized by ***shortest job first*** and ***round robin***, respectively
- Context switching overhead is due to the CPU caches
  - CPU keeps most recently used data in nearby caches, so it's more efficient to let an ongoing process continue.
- ***I/O-blocked*** processes make progress without using the CPU
  - We should prioritize I/O-bound processes
- ***Multi-Level Feedback Queues*** are often used in real OS schedulers
  - Prioritizes "polite" processes that use little CPU time when scheduled
  - CPU-bound processes squander their time quotas and lose priority

Virtual Memory

- Memory is divided into equal-sized ***pages***.
- ***Page tables*** translate virtual page numbers to physical page numbers.
- Showed the details of page table entries (PTEs):
  - High bits translate from virtual page number to physical page number.
  - Low bits in the PTE are used to indicate present/rw/kernel page.
- During a context switch, kernel changes the **%CR3** register to switch from the page table (VM mapping) of one process to another.
- VM is handled by both the OS and CPU:
  - **OS** sets up the page tables and handles exceptions (page faults).
  - **CPU** automatically translates every memory access in the program from virtual addresses to physical addresses by checking (*walking*) the page table.

VM & Paging costs & optimizations

- **Latency cost**, because each memory access must be translated.
  - **Translation lookaside buffer (TLB)** caches recent virtual to physical page number translations.
  - Software-controlled paging removes page tables from the CPU spec and lets OS handle translations in software, in response to TLB miss exceptions.
- **Space cost**, due to storing a page table for each process.
  - Linear (one-level) page tables are large.
  - Smaller pages lead to less wasted space during allocation, but more space is consumed by page tables.
  - **Multi-level page tables** are the only way to truly conserve space.
  - Mixed-size pages reduce TLB misses.
- Copy-on-write fork, demand zeroing, lazy loading, and library sharing all reduce physical memory demands.

Paging overview

- **Virtual memory** addresses are translated to physical memory addresses by the CPU, and the translation is dynamically configured by the OS in each process' page table.
- **Swapping** is the movement of pages between disk and physical mem.
- Page tables also allow several memory management optimizations:
  - **Copy-on-write fork** – delays memory copies
  - **Shared libraries** – read/execute-only code can be shared by several processes
  - **Lazy allocation/demand zeroing** – wait before allocating user memory.
- **Filesystem caching** allows page-sized portions of files to be stored in physical memory.

Swapping gives the illusion of lots of memory

- Disk is slow, but large, and can be used to store RAM's overflow
  - Disks have high ***throughput*** (transfer bitrate) but high ***latency*** (delay)
  - Magnetic disks have even higher latency than SSDs, due to moving parts.
- Paging and swapping work together, using the same CPU mechanisms
  - If a page is marked "not present" it may be either invalid or swapped to disk.
    - Or it might indicate *lazy allocation*, *lazy loading*, or *copy-on-write*
    - High bits of page table entry can store disk location of swapped page.
- ***Page replacement policy*** decides which page(s) to ***evict*** to free memory
  - Swapping can be done ***on demand*** or in the ***background***
  - Having some free physical frames will prevent delays for allocations.
  - ***Accessed bit*** and ***Dirty bit*** in PTEs inform the page replacement policy
- ***Thrashing*** is when swapping prevents the system from doing any work.
- ***Unified page cache*** handles both traditional paging and ***file caching***.
  - Makes filesystem access seem just as fast as memory access.

Types of page faults *(new slide)*

- **Minor/soft**: Page is loaded in memory, but PTE is not configured:
  - OS just wants to be informed when the page is accessed, so it *pretends* to evict the page (just mark it *not present*). Useful if CPU has no accessed/dirty bit.
  - Memory can be shared from another process (eg., copy on write, shared library)

  Response: update the PTE.
- **Major/hard**: A disk access will be needed:
  - Anonymous page (process data) may have been swapped out.
  - Lazy-loading program executable.

  Response: load the page from disk
- **Invalid**: User program misbehaved:
  - Dereference null or invalid pointer.
  - Write to page that is read-only.
  - Execute code on a page that is not executable (for security).

  Response: terminate the process.

Free Lists

- Freed memory is put on a ***free list*** to be reused for later allocations.
- A single header can be cleverly used and re-used for two purposes:
  - As a linked list node when the block is free/available
  - To store the size of the allocated block to help service *free* calls.
- Free space management ***policy*** determines:
  - which free blocks to choose for an allocation, and
  - When to ***coalesce*** (join) adjacent free blocks
- Free block choice policies include:
  - **First**, **next**, **best**, and **worst** fit.

$2^{14} = 16,384$

Q4) xv6 uses a two-level page table and ~~4096~~ byte pages. How much space is consumed by an xv6 page table for a process that uses just the lower 140 megabytes (140*1024*1024) of memory?

Q5) How much space would be consumed by the process above if a linear (one level) page table was used?

Q4:

$$\#pages = \frac{\#\ \cancel{m}bs}{\cancel{m}bs/page} = \frac{140 \cdot 2^{20}}{2^{14}} = 140 \cdot 2^{6}$$

$$= 8,960 \text{ pages}$$

$$\#page\ table\ pages = \left\lceil \frac{\#PTEs}{\#PTEs/page} \right\rceil = \left\lceil \frac{8,960}{4096} \right\rceil = 3$$

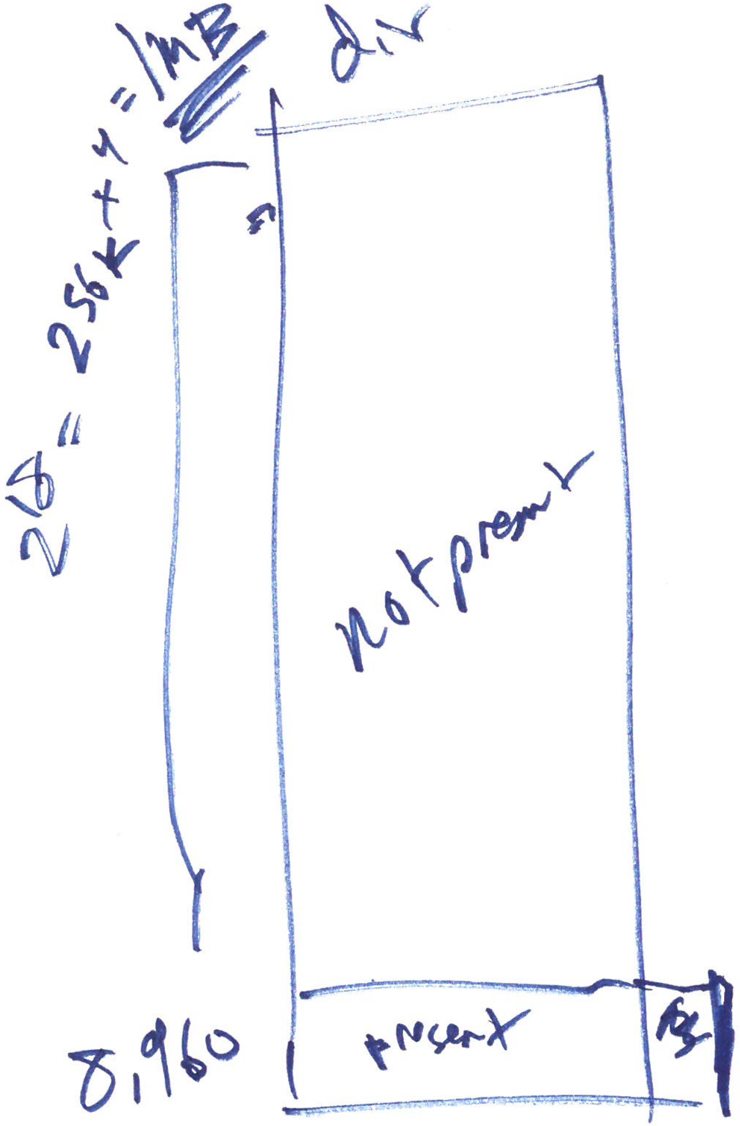
$$\frac{16,384\ \cancel{page\ size}}{4\ \cancel{PTE\ size}} = \frac{bytes/page}{\cancel{PTE}\ bytes/PTE} = \frac{PTE}{page}$$

$$total\ size = \#\ dir\ pages + \#\ table\ (2nd\ level)\ pages$$

$$= 1 + 3 = 4 \text{ pages}$$

$$= 4\ pages * \frac{16384\ \cancel{pages}\ bytes}{page} = \underline{65,536\ bytes}$$

Q5:



$$\frac{2^{32}}{2^{14}} = 2^{18}$$