9/25/14
Lecture 08
Caching, Prefetching, Buffer Cache: Akshar Bonu, Daniel Leichus, Graham Lustiber

Topics:
1) Process isolation
2) Caching

Start by coding a program that will break the machine.

- * (char *) 0x100000000 = 0: accessing an address that the program is not allowed to address

What should a processor do when it doesn't know what to do?

- Going further might expose a secret to a spy so the program should explode (joke)
- What happens: segmentation fault (core dumped), while everything else keeps working
    - This is weird because this breaking doesn't affect anything else
    - This is done by the OS, not the processor, as we want to be able to do process isolation

Virtual machine inside virtual machine - the kernel program he ran

k-hardware.c: interfacing with hardware of the system
kernel.c: initialize hardware, clear console, print hello world, spin forever; kernel is most privileged part of the operating system

Virtual machine has grabbed keyboard, technical difficulties, had to reboot ubuntu.

What would happen if I added the * (char *) to the kernel.c?
- Rebooting every time that line is executed
- Very dangerous error; in class example, no progress at all forever
- Unmodified hardware reboots when it gets an instruction it can't understand

How is it that when we do it this is not caused?
- Modularity: one module does not affect of another; combination of modules behaves better than individuals
- This brings us to:

Process Isolation

Process:
- Program in execution (i.e. emacs, test001); ./ turns a program into a process; multiple processes of same program (i.e. many emacs running at the same time)
- Operating system's implementation of an abstract machine; OS has full access to the real machine

Operating system:
- The program with full privilege over machine operation (i.e. rebooting)
- Goal: make the computer more robust by process isolation

Process isolation:

- Property that each process runs without affecting any other process; except through explicit sharing
- Every process has its own memory; process a cannot look or modify process b's memory; they each behave as if they own the entire machine (disk, processor, etc.)
- Sharing is mediated and enforced by the operating system (needs a cooperation between hardware and the operating system)
- Operating system gets control when illegal instruction or bad memory is accessed

Time slicing:
- Every certain amount of time (using system clock) gives the kernel back control of the processor; kernel will give it to someone else (reassign)
- This stops an infinite loop
- Cons: time wasted in switching processes
- Pros: divides processing time between multiple processes
- compare to the pros and cons of traffic lights

Kernel:
- Windows: referring to window kernel and ancillary set of programs that give appearance of the operating system
- Kernel is the single program with most privilege
- When the kernel has nothing to do it is in a while loop (old machines)
- Now, when the kernel has nothing to do (not enough instructions for 100% usage), it is put to sleep (saves power)

Strace:
- Strace is a snoop that sits on boundary below and watches every system call made by the process to the kernel
- Prints system calls (API to the kernel)
- Line below between User and Kernel is the system call interface
- System call: explicit request from the process to the kernel (i.e. read, write, exit)
- Necessary to cause explicit sharing between processes (only safe when mediated by the kernel)

User kernel boundary:

[ ] ← Process [ ] ← cache

User

-------------------------------- ← Barrier is mediated by the system call interface (explicit requests from the process to the kernel)

Kernel

[ ] <- cache Disk -> [ ]

unix top:
- Task manager to see CPU usage

Sleep system call:
- process requires no processor power for next x seconds so don't devote any

Polling

- Repeatedly uses the CPU
- Usually not good, but, notices changes faster
- For example, a while { ; } loop
- Resource intensive, but it can detect changes faster

Blocking

- Explicitly asks kernel to wake it up when something happens; waiting for something to happen that they're interested in
- Good: leaves more resources available for other programs
- For example, infinite while loop that sleep();'s

Caching

O SYNC flag

- Write and only continue after response

W_03 vs W_01; former batches so it is quicker

File descriptors:

- 0: stdin
- 1: stdout
- 2: stderr
- 3, 4, 5... any other processes that are run
- type int: type of these are FILE*;

Why does Kernel prefer integers over pointers?

- Fundamental problem: must validate arguments to a system call; if it doesn't check it then it can cause a security hole; integers are easier to check than char* strings, just index into an array of file structures

W_01

- Write a single byte to the disk; kernel does not respond until disk responds; mile long track with a snail on it in comparison to postman analogy (slow); disk has very high per call cost; disk has small per unit cost;

W_03

- 1000 times faster; before sending to disk it is sent to memory (called a cache) and the entire batch is sent to disk (batching); not waiting for disk as opposed to waiting for memory
-

W_05

- Code is written to write one byte at a time; system call is doing 4096 at a time; there is a cache in the processor as well (exists in user side & kernel side); act of making a system call (because kernel must validate all arguments) is expensive and this avoids this

Cache:

- Local fast storage used to speed up access to slow storage (Kernel memory is faster than disk; higher in storage hierarchy); if cache is full then blocking will occur
- when writing we **delay operation to batch it** - absorb writes into a cache & delay into the next level of storage

- when reading we **do the operation early to prefetch it** (opposite of writing diagrammatically as well);
- doing reverse/random reading is slower because operating system can't prefetch as it isn't linear data

There are actually many tiers of caching in the entire system (beyond memory vs. disk), at the lowest level is a 32 bit array

R_01

- 1 byte at a time

R_05

- 4096 bytes at a time (prefetching)

To cache even quicker we cache in the processor (the CPU) before the main memory (RAM) using L1, L2 (CPU cache) - save for far off future lecture