Scribe Notes (Sept 25, 2014) - Serguei Balanovich, Nhu Nguyen, Emily Houlihan, Alex Wang

Announcements

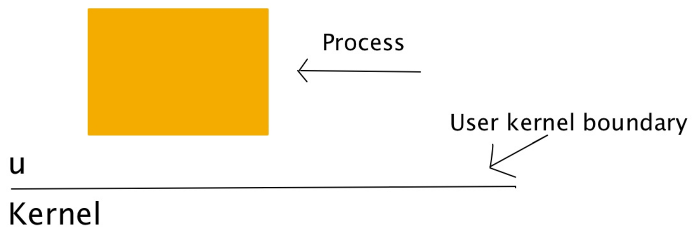
Pset 2 has been released, there IS an intermediate check-in
encourage people to make an associative cache
Deadlines for all other Problem Sets Posted

Process isolation - the great idea of computer systems

- start by coding a program that messes up the machine
  - sets the 0x10000000 byte to 0
  - Even though we know this won't execute, let's pretend we are the computer – when we encounter something like this (accessing an address the program is not allowed to access) – what should we do?
  - this is a memory address we're not allowed to access, we're trying to get something that doesn't exist
  - The processor is being asked to access memory that doesn't exist, so what should the processor do?
  - this is where the idea of halt and catch fire instruction happens - if the next line that you do will go terribly wrong, you have to stop
  - expectation: system will shut down
  - reality: segmentation fault
- This seems really odd
  - It's weird that this program only killed itself and did not affect anything else on the machine.
  - everything that's running never seems to conflict with each other
  - our computers don't need to restart because of modularity
- Now we're going to run a very bare bones operating system
  - Cd into the ../0s00/ directory
  - In there is a very small operating system in there (the smallest OS you've ever seen)
  - hardware.c → it's full of stuff we really don't understand
  - kernel.c - it has grabbed access of his keyboard, and he can't do anything
    - and since the os was designed to do anything, he cannot escape
    - what if I add the same 0x1000000 = 0 to the os we made
    - then the computer starts rebooting constantly, every time the machine does something wrong, it has to entirely restart
    - NASAL DEMONS ARE DANGEROUS
- Why then does this not happen in C?
  - So why is it that this doesn't happen to us on a more regular basis? How do we get a seg fault and the rest of the computer is still functioning?
  - so now that we've seen the difference between software crashes and hardware crashes is very different (software is more modular)

    - one thing doesn’t matter between another
- Process
    - Program in execution. Starting emacs in lecture over and over ends up with *one* program (emacs) but *many* processes
    - OS (Operating System)’s implementation of an abstract machine.
        - Operating System - is the program with full privilege over machine operation.
            - Kernel – the most privileged component
        - The goal of the OS is to make the computer as a whole more robust by enforcing Process Isolation
        - enforces and mediates the sharing of processors and resources
        - hardware detects illegal operation, informs OS, OS kills the process
        - kernel: most privileged part of the OS
            - saying "Windows" refers to the Windows kernel and the ancillary function
        - the reboot is called a triple fault
            - created by design of Intel engineers
        - timer interrupt: every increment of time of the system clock, gives control back to kernel
            - otherwise infinite loop would beat process isolation
            - tradeoff between programs stalling and handling interrupts in determining time increment
            - controlled by kernel
- Process Isolation
    - The property that each process runs <u>without</u> affecting any other processes, except through explicit sharing
    - Every process has its own memory – can process A affect process B? No. Can A access B’s memory? No.
    - Each process pretends as if it owns the entire machine as a result of the Process Isolation abstraction
    - In fact, there is just *one* processor, but a bunch of processes are running at once, all thinking that they have full control of the disk, memory, etc.
    - The sharing comes from the Operating System.
    - Processes act as if they have access to the entire machine (you just don’t)
    - It means you don’t have to worry about being interrupted by another program (yay working in a vacuum)
    - the sharing is being mediated and enforced by the operating system
        - this is why os’s are soooo important
        - bugs in os’s then are VERY bad!! because things will shut down
    - process isolation is technically sort of inefficient (but it stops us from the computer stopping anytime there is an error)

- Let's go back to our original program, but replace the false memory access with an infinite loop. This program would not do anything illegal, so why would the Operating System ever get control of this?
    - this is technically a safe action
    - so when can the os regain control of this program
    - in a bad os this would freeze you machine
    - but other programs still can use the machine while this is happening
    - This is because of **time slicing**, which kicks the kernel, and gives the kernel the control of the process
        - how to determine the length of the time slice?
        - you can vary it, but usually it's .001 seconds so we can give other programs abilities
    - the kernel and the os are super similar!!! the kernel is the single program that has the most privilege
    - during process shifting we hide the fact that a process has been interrupted from the process



- system calls are the API for the kernel
    - system call interface process requests from processes to kernels
    - strace watches

Demonstration of time slicing

- using the top command to let you see processes
- in one you have a loop that does nothing and uses 99% of cpu
    - called polling - repeatedly uses the cpu through many time slices
    - polling generally not good, but notices changes quicker
- the other one calls sleep() which talks to the OS - which doesn't use any of the cpu
    - blocking - when a program asks kernel to wake up when something happens, but pass by otherwise

What we can see is when we run an infinite loop to do nothing forever, the program takes up ~100% of the CPU (83.7% in this case because Chrome is rendering bubble animations in the background)

Now change the infinite loop to `sleep(1000)` on each while iteration

Now a.out is not running or using any of the CPU

The difference between the versions is:

Polling: When a program repeatedly uses the CPU

Usually not good but tends to notice changes faster

Blocking: When a program asks kernel to wake it up when something happens

Leaves more resources available for other programs

Caching

Memory refresh

- strace the programs we wrote last time
    - run syncbyte - gets a look at the system calls we got from that
    - writing a byte at a time using system calls (we don't have to wait during the write to the disc)
    - and then run with stdio, which is super fast
- How is Stdio fast?

WERE USING CACHES
use diff to find the difference between the two programs we wrote, and the only diff was o-sync

Let's run *syncbyte* v1, which writes one byte at a time

Run *syncbyte* v3, which does the same thing but we don't care when the data gets written to disk (better performance)

Run *syncbyte* v5, which writes one byte at a time using *stdio*.

If we look at the .out for v1 and v3, we'll see an output of bytes that are nearly identical to one another. The difference is that in v1, an o_sync flag is present (don't proceed until you write to disk). V3 *batches* the disk access.

But looking at the .out from v5, it’s very different. We have multiple writes. One of them shows us how fast the program is executing and the other shows the writes happening to the disk. This version calls *fwrite* instead of *write*.

There are really different results between stdio and syncbyte

- writes to three are going to disc, the writes to 2 are the ones we see on the screen
- w0 is calling directly write to disc
- w03 writes using batching, doesn’t use O_SYNC tag so it doesn’t need to wait to hear back from the disk
- w05 writes using fwrite, or stdio calls (which makes efficient system calls)

things you see in the calls
write (2, <arge>, <arg> …….) and write(3, <arg>, <arg>)

- file descriptors
    - 0 = standard input
    - 1 = stdout
    - 2 = stderr
    - 3 = next file we write
    - all of these values have type integer rather than pointers
    - because the kernel rather deals with integers than pointers, if it gets ints rather than pointers, it will be less likely to create a security hole
- this is one of the things you have to think about when creating a file call structure

If the kernel mistreats a file, it can cause a security hole. These integers are easy to work with which makes it easy for a kernel to validate its interface.

When the processor says “write” in w01, it waits for the disk to respond before moving on (The disk is like a long trail and the write snail has to make it to the right spot before it can “write” and proceed)

In w03, there’s a *cache* in memory that’s used to collect writes and allows us to write and come back over and over (with no delay) until we run out of space or the OS feels necessary, the entire cache is written to disk in one go.

The process is not waiting for disk, it’s waiting for availability of memory for the *cache*. So what’s a cache?

- *Cache* – Local, fast storage which is used to speed up access to slow storage. The memory that the kernel is using to store the cache is a lot faster to access than the disk.

Disk has a *large* per call cost (every call takes a long time) but a *small* per unit cost (more units to write does not increase writing time)

Eventually the cache might be filled up, so then you empty it, and then can start the process back up again
this is the same idea as doing byte writes as blockwrites

In w05 the user level code is written to write one byte at a time, memor is actually being written 4096 bytes at a time

This is a really deep fundamental type of process for systems

- We are using memory as a local copy before writing all that data to disc
- There are many layers of different caches

This is caching for writing. Can we use caching for reading?

- In this case you bring a huge chunk of info out of the disc and then give it to the process when we need it
- Use strace to see the different between stdio read and a single byte read
- It's asking for data before we need it, we call this pre-fetching
- How does it know what to prefetch?
- If we have an enormous file, and we start reading it, we will try to fetch the file sequentially, if you start jumping around, it stops prefetching
- If we access the file in reverse order, it is >10x slower than reading the file forward.