Advanced OS Refresher

A short summary and study guide

Matt Chung (https://blog.mattchung.me)

2020-10-20

Contents

Getting Started

About Refresher Course

If you are thinking about taking advanced operating systems (AOS) course at Georgia Tech, you'll be recommended to take the Georgia Tech - Refresher - Advanced OS course offered on Udacity. This book does not replace the course. Rather, this book serves as a supplement.

Maybe you just don't have enough time to watch all the lectures (that's okay, life gets in the way). Maybe you have a pretty strong foundation and you just want to quickly skim through the contents. Whatever the case, this book should help.

How to read this guide

Each of the three sections — memory systems, file systems, multi-threading — starts with "Can I skip this section". If you can successfully answer most of the questions listed, you're probably safe with skipping that particular section.

Memory Systems

Can I skip this section?

If you can answer most of the questions below, you can probably skip the memory systems section:

- Do you understand the memory hiearchy (i.e. L1, L2, L3 cache)
- How does direct mapping work?
- What's the difference between a write-through policy and a write-back policy?
- What is a fully associative set?
- What problem does a translation lookaside buffer solve?
- What's a page fault? How is a page fault resolved?

Summary

During this course refresher, my brain started slowly recalling the various system concepts of memory systems. Here's a quick recap of what I was able to remember from taking four system courses (i.e. computer organization, operating systems, high performance computing architecture, and compilers):

- **Naive Memory Model** – Normally we think that memory access is fast. But that's an oversimplification and it is much more nuanced. In fact, underneath the hood, there are cache systems (i.e. L1, L2, L3 cache)
- **Cache Motivation** – We can speed up memory access by a factor of 10 by introducing a cache. That speed up can reduce the number of cycles from 100 down to a meager 4. Continuing with this example, a 99% hit rate will yield 4.96 cycles on average, 90% hit rate of 13.6 cycles (huge difference)
- **Hierarchy** – Given that we want the cache rate to be high and fast, how do we make the right trade offs? By creating a memory hierarchy: registers, L1, L2, L3, then memory, which can also serve as a cache for disks too. As we move down the hierarchy, speed reduces as well as the cost.
- **Locality and Cache Blocks** – To populate our cache, we apply two heuristics: temporal (i.e. address we just accessed we are more likely to access again) and space locality (i.e. addresses adjacent to most recent address are more likely to be accessed). We store addresses in cache blocks, which can vary in size depending on design
- **Direct Mapping** – Each memory address can live in only one cache line. This is easy to manage but one downside is thrashing, a situation in which one cache line continues to get evicted over and over despite other cache lines unused.
- **Set Associative Mapping** – One way to overcome the limitation of direct mapping (mentioned previously) is to use set associative mappings: a cache line can live in multiple cache lines. The

trade off? Need to implement some sort of cache evication algorithm (e.g. least recently used) and will need to look up tag (most significant bits of the address) in multiple cache lines
- **Fully Associative set** – Any address can live at any cache line. Basically, the index bits (equal to number of cache entries) is now 0 (when it was previously equal to number of cache entries). Similiar to both the direct and set associative sets, we still use the offset (n bits, where n is equal to number of bytes within each block)
- **Write Policy** – write-through (write to both cache and memory, keeping two consistent) and write-back (only writes to cache, beneficial when writing to same block many times before being evicted) and write-allocate (read into cache, then choose one of the two write strategies) and no-write-allocate (bypass cache and just write to memory). The relationship between these policies?

Naive Memory Model

Normally, we think that memory access is fast. But it's much more nuanced. In fact, underneath the hood, there are cache systems (e.g. L1, L2)

Cache Motivation

adding a cache has a factor of 10 speed up. Say we have cache hit that's 4 cycles, and a memory access that's 100. For 99% rate we get 4.96 on average; for 90%, 13.6. Huge difference)

Memory Hierarchy

We want cache hit rate to be high and fast, but how do we make right trade offs? Create a memory hierarchy: registers, L1, L2, then memory, which can serve as cache for disk too. Also, there might be room for L3 cache if the processor is multi-core)

Locality and Cache Blocks

We apply a heuristic of temporal and space locality. Things that we just accessed, put in cache; things adjacent or nearby, throw that in cache too. And, be consistent, using cache blocks)

Quiz: fill in the table

Took me much longer to answer this quesiton, about 15 minutes to re-jog my memory. But the offset determines which byte in the block. Since the block contains two bytes based off of the photo, we

know that the offset bit (n=1), 0 represents first byte 1 represented the second byte. Then the next m bits link to the index, the remaining bits representing the tag)

Quiz: How many bits

Remember Matt, to calculate the number of cache lines, we take the total size of the cache (say 512 bytes) and divide it by the block size (say 64). This would be 512/64 = 8. So, now we have the index value. And calculating the offset is easy: take the block size (64 bytes) and derive the number of bits. So for 64, we need a total of 6 bits. The rest go to the tag).

Set Associative Mapping

We might run into a situation of thrashing, one cache line being evicted over and over, based off of the access pattern)

Write Policy

write-through often paired with write-allocate and write-back often paired with no-write allocate. Elaborate on this more Matt … don't think you truly understand)

Address Translation

We employ a common CS trick: indirection. This technique promotes three benefits: large address space, protection between processes, sharing between processes. Each process has its own page table)

Paging

Because we can have more virtual memory than physical (thanks to indirection, above), we need a mechanism to translate between the two. This mechanism is page translation table. And although the VPN and PFN, virtual private number and physical frame number respectively, can differ, the offsets into the page itself will always be the same)

Page Table Implementation

To map a virtual private number to physical private number, we need a page table. If we start with a single array, it will be huge. Take the page size, covert this to bits, subtract page size (in bits) from virtual address number (in bits). Take that number in bits, multiply it times the size of the PTE, page table entry, and you get your page table. Huge. Instead, using multiple tables to reduce size, at the cost of lookup time)

Accelerating address translation

To avoid hitting memory when looking up the page table, architects added a translation look aside buffer. But it's not free. I Learned that with a TLB, there is a problem of using virtual page numbers: how do we tell if that virtual page is valid for a process? We can tackle this by either 1) flushing the TLB during a context switch or 2) adding some unique identifier to the TLB entry)

Page Table Entries

Take the number of bits of the physical address, subtract the page size (convert to bits), that that gives us some remaining bits to play with: metadata. Inside of this location, we can store access control, valid/present (again: reminding me of flickers of HPCA code), dirty, control caching)

Page Fault

If a process requests (or reads) an address that is not assigned a physical address, a page fault will occur (not a big deal), and the page fault handler will run. This handler will start by checking the free-list (reminds me of computing systems from a programmer's perspective) and if address is available, then use the address, then restart the process. But if no page is available, need to evict, then start up again)

Virtually Indexed, Physically Tagged Caches

optimization the TLB and cache lookup, from sequential to parallel

Quiz

When calculating the max number of entries in the cache, for a virtually indexed, physically tagged cache, we need not worry about the virtual or physical address size. Just need to think about the page

size and size of cache blocks

File Systems

Key words: unified buffer cache, inode, journaling, direct memory access DMA), mmap, file allocation table

Can I skip this section?

If you can answer most of the questions below, you can probably skip the file systems section:

- What does the `mmap` system call do?
- What's the purpose of an inode?
- What problem does a unified buffer cache solve?
- How does journaling work? What are the trade offs with journaling?
- What is DMA?

Although I've covered file systems in previous courses, including graduate introduction of operating systems, I really enjoy learning more about this concept in a little depth. The file system's module is packed with great material, the material introducing a high level introduction of file system and then jumps into the deep end unveils what an operating system does behind the scenes.

Main Take Aways

The high level purpose of a file system provides three key abstractions: file, filename and directory tree. A developer can interface with the file system in two ways: a position based (i.e. cursor) and a memory based (i.e. block). Also, in `C programming language`, the function strol converts a string to a long.

The `mmap` system calls maps a file on the file system to a contiguous block of memory (the second method a developer can interface with a file systeme).

FAT (file allocation table) glues and chains disk blocks together. It is essentially a linked list (persisted as a busy bit vector table) that is great for sequential reads (i.e. traversing the linked list) but awful for random access (i.e. to get to the middle, need to traverse from head)

EXT linux file system is based on inodes and improves random access using 12 direct pointers (13th pointer provides first level of indirection, 14th pointer second level of indirection, 15th pointer third level of direction) Learned about the buffer cache (i.e. write-through) and how we a journal can help stabilize the system while maintaining decent performance Linked list and busy bit table

Introduction

Will introduce File Systems and what an OS does behind the scenes

File System Concept

Universal interface of file system contains three key abstractions: file, filename, and directory tree

Access Rights

Learned why we need the execute bit on directories: this permission affects whether you can pass through the directory.

Quiz: Permission Error

Again, cannot `cat` another file within the subdirectory underneath the directory because parent directory lacks the execution bit.

Developer's Interface

Two ways to interface with file: cursor (i.e. position based strategy) or as a block of memory

Quiz: Sabotage

Learned that there's a function called strtol that converts a string to long, and then I also opened up the man pages for fcnt)

MMAP

Can treat a file like a memory buffer and then persist memory map to file by calling munmap).

Quiz: Shuffle

Skipping over quizzes for writing C code … I do this enough at work)

Allocation Strategies

Block represents continuous amount of space on disk, analogous to a page in virtual memory. We can keep track of said block using either free list or busy bit vector. In next sections, we'll explore how FAT accomplishes the following: simple & fast, flexible in size, efficient use of space, fast sequential access and fast random access

File Allocation Table

FAT is the glue that chains the blocks together. Directory files capture hierarchy and starting blocks for the files

Quiz: Values in FAT

We can identify the starting blocks by eliminating any block that has another block pointing to it. Also I'm curious if defragmenting improves the linked list somehow or the FAT table, moving sectors closing together?

File Allocation Table continued

Great for sequential access like copy files to a USB drive. But has fallen out of popularity due to poor random access given to hit the middle, we need to traverse the entire linked list)

Inode Structure

Instead of Fat table, we have an inode that contains metadata and 15 pointers., the first 12 are direct, 13 is first layer of indirection, 14 is the second layer (contains 2 levels) and 15 contains three levels.

Quiz : Data Blocks

When calculating the maximum size without second level of indirection, we take 4 times 12 then multiply 1024 * 4 (i.e. 1024 contains number of entries))

Inode Structure

When compared to FAT, Inode offers much better performance for random access memory.

Buffer Cache

We buffer disk data in something called a `unified buffer cache`. I learned that if you want to really persist to disk, call fsync: otherwise, data sitting in unified buffer cache may get lost. Also, this is the reason why OS warns you not to remove storage device without ejecting them first. I had no idea that's the reason why. Again, how neat)

Journaling

Very neat idea to handle the trade off of using the unified buffer cache. By adding a journal, we have robustness but trade off is that for every write there is another write. So two writes. But this journaling mechanism offers a nice trade off of only writing dirty pages to disk when the time is more convenient.

Direct Memory Access

Hardware optimization: Direct Memory Access. This way, CPU does not need to copy data over the bus. Disk controller can access memory directly)

Multithreaded programming

Can I skip this section?

If you can answer most of the questions below, you can probably skip the threading section:

- What does the `pthread_join` call do?
- What's the difference between `pthread_exit` and `exit` when called by the main thread?
- What memory (e.g. heap, global, stack) is shared versus not shared between threads?

Summary

The virtual address space is divided into several sections, including the heap, globals, constants, and code. Threads, within a process, share all these sections **except** that it has its own stack.

Writing multi-threaded code is difficult and requires attention to detail. Nonetheless, multi-threaded offers parallelizing work — even on a single core!

When compared with process context switches, thread context switches costs less (i.e. they are cheaper).

Joinable vs Detached

A created thread falls into two categories: joinable or detached. Joined thread must explicitly be cleaned up, whereas a detached thread will deallocate its stack automatically upon exit.

Thread layout design

When using threads, there are a couple different design patterns: team, dispatched, pipeline. Selecting the correct design depends on the application requirements.

Finally, when writing multi-threaded programming, the program must keep in mind that there are two different problems that they need to consider: mutual exclusion and synchronization. Regardless, for the program to be semantically correct, the program must exhibit: concurrency, lack of deadlocks and mutual exclusion of shared resources/memory.