Operating Systems

Chapter 2: Introduction to OS

- The operating system is in charge of making sure the system operates correctly. The primary general technique is uses is virtualization. Virtualization happens when the OStakes a physical resource (e.g. processor, or memory, or a disk) and transorms it into a general, powerful, and easy-to-use virtual form of itself.
    - For example, the OS provides the illusion that it does more than it actually does by virtualizing the CPU (pretending things are running concurrently)
    - For example, when you run the same program twice (concurrently) and print the value of an address, youll likely see the same address (0x2000000, for example), and yet this address is being updated independently by these two separate programs. This is possible because each process has its own virtual private address space, instead of sharing the same physical address space.
- The OS's goal is to manage resources efficiently (CPU, memory, and disk)

Protection

1. The OS segregates access to certain parts of its internals such that users cannot take certain actions. These two tiers of actions are called:
    a. System calls – Only this call can seize access (transfer control to / jump into) the OS. When this call is initiated, a special hardware instruction called a trap is initiated. The hardware transfers control to the trap handler which raises the privilege level to kernel mode. In kernel mode, the OS has full access to the hardware of the system itself.
        i. When the OS is done servicing the request, control is passed back to the user via a special return-from-trap instruction.
    b. Procedure calls – User applications run in user mode and make procedure calls.

Chapter 4: The Process – Abstraction

1. The process is a running program. It takes byutes from disk, contains instructions, some static and non-static data, and transforms them via the program into something useful.
2. The OS creates the illusion of multiple running programs by running one, stopping it, running another, and so on. The technique is known as time sharing of the cpu and each process gets its own time-slice.
3. The OS determines which program to run via the scheduling policy.
4. The abstraction provides by the OS of a running program is called a process. A process can be defined as the different parts of the system it accesses or affects during its execution (machine state).
    a. Memory – the instructions lie in memory, the data the program reads and writes, and thus the address space of the process
    b. Registers – Many instructions are explicitly read or updated from / in registers
    c. Have their own instruction pointer, stack pointer, and frame pointer.

Process Creation

1. Load its code and any static data into memory / address space of the process (programs initially reside on disk).
2. Memory must then be allocated for the process before it can run (we need a runtime stack).
3. The OS will perform IO tasks like opening file descriptors (in UNIX its input, output and error)
4. The OS must allocate some memory for the program's heap.
5. It will start running the program via the entry point (namely Main())

Process States

1. Running – Currently executing
2. Ready – In the ready state, but for some reason the OS has decided not to run it at the given moment
3. Blocked – Not ready to run until some other event takes place. Like certain IO is taking place
    a. The register context will hold the contents of a stopped process' registers. When it is stopped, the contents of its registers will be saved ot this memory location. By restoring these registers, the OS can resume running the process.

- All processes will be stored in a process list (which is a data structure)
- The OS task scheduler is responsible for determining which process runs and which ones don't run despite being ready. For example, if process 0 is blocked doing IO, process 1 will be switched in, when process 0 is ready (unblocked), it will be moved back into ready state. When process 1 finishes, process 0 can proceed to run.

Chapter 6: Limited Direct Execution

- The OS must virtualize the CPU in an efficient manner while retaining control over the system. The technique used to do so (so that a CPU doesn't run away with the machine) is called Limited Direct execution.
- The following are problems that are addressed by limited direct execution
    - **Restricted Operations**
        - If a program wants to perform some kind of restricted operation, such as issuing an I/O request to a disk, or gaining access to more system resources, it must do so via its process mode.
        - All programs operate in **user mode**, where the program is restricted in what it can do. For example, it cant raise an IO request
        - The operating system runs in **kernel mode**, to run privileged operations.
            - To enable a process to perform a privileged operation, a users program needs to make a **system call.** System calls allow the kernel to carefully expose certain pieces of functionality to user programs such as accessing the file system, creating, communicating with, and destroying other processes, etc.
                - It does so via a special **trap** instruction. When it is done, the OS calls a **return from trap** instruction which will resume execution in user mode.

        - The trap instruction saves register state carefully, changes the hardware status to kernel mode, and juumpts into the OS to a pre-specified destination: the trap-table.
        - There is a **trap table** that maps a numeric code to ta trap handler. A system-call number is assigned to each system call.
    - **Switching Processes**
        - If a process is running on the CPU, by definition this means the OS is not running. If the OS isn't running, it cant do anything. So how do we regain control of the CPU so that it can switch between processes?
            - The OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place.
        - The decision to switch is down by a part of the OS called the **scheduler**
            - A switch is known of as a **context switch**.
                - A concept switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process (onto its kernal stack) and restore a few for the soon to be executing process. By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the running process, the system resumes execution of another process.
                - A context switch doesn't only lead to a performance penalty because the OS needs to restore some registers. Programs also build up a lot of state in their CPU caches, TLBs, branch predictors, and other on-chip hardware. When we switch to another job, the state must be flushed and the new state relevant to the job is brought in.

Chapter 7: Scheduling

- There are several scheduling metrics that are used to determine scheduling performance
    - **Turnaround time** – The time at which the job completed minus the time the job arrives at the system.
    - **Response time** – The time the job arrives at the system to the first time it is scheduled.
- There are several scheduling algorithms as well:
    - **FIFO** – straight forward but bad if you have large jobs followed by small jobs due to the convoy effect (where quick jobs are help up by long jobs leading to high average tunaround times).
    - **Shortest job first** – The name describes it well. Still subject to convoy effect if jobs don't arrive at the exact same time though.
    - **Shortest time to completion first** – The scheduler would determine which job takes the least amount of time and, if it takes less to complete than the amount of time remaining on the current task, it preempts the current task with the new task.

- **Round robin** – This algorithm is most sensitive to response time (imagine having to wait 10 seconds for the system to respond to your terminal). The basic idea is that each job runs for a single time slice before switching to the next job in the queue. The length of a timeslice is a multiple of the timer-interrupt period of 10 milliseconds.
  - Context switching does take time though, and we don't want that to dominate the overall system performance. Deciding the time slice length represents a trade-off to a system designer, making it long enough to amortize the cost of switching without making it so long that the system isn't responsive.
  - While the response time is good in RR, the turnaround is poor.

Chapter 8: The Multi-Level Feedback Queue

- This is the most well-known approach for scheduling. The problem it tries to address is two-fold:
  1. Optimize turnaround time
  2. Minimize response time (make the OS more responsive to the user).
- There are multiple queues, each assigned a different priority level. There are several rules:
  - Priorities decide which job to run (rank A > rank B, A runs)
  - Amongst jobs with the same priority, round robin is chosen.
  - When a job enters the queue, its placed at the highest priority
  - If a job uses up an entire time slice while running, its priority is reduces
  - If a job gives up the CPU before the time slice is up, it stays at the same priority.
- MLFQ varies the priority of each job based on its observes behavior.
  - For example, if a job often relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave.
    - If a job uses the CPU intensively for long period of time, the MLFQ will reduce its priority. In this way the MLFQ uses the history of the job to predict future behavior.
- Some programs can game the scheduler. For example, relinquishing to the CPU so they maintain at the topmost queue. This leads to starvation of long-running tasks.
- Some systems allow for user advice to help set priorities. In UNIX you can use the command-line utility **nice** to increase of decrease the priority of a job
  - Positive nice values imply lower priority (this is used by Linux's completely fair scheduler)

Chapter 9: Scheduling – Proportional Share

- Instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain amount of CPU time.
- One approach is a **lottery ticket** approach, where each process is assigned a certain number of tickets. The scheduler draws a number and the process that contains that ticket is the winner for that time slice.
  - Lottery scheduling provides a mechanism to manipulate tickets in different and sometimes useful ways.

    - Currency allows a user to allocate tickets between its jobs in whatever way it wants.
        - For example, assume users A and B have each been given 100 tickets. User A can give its two jobs, A1 and A2 each 500 currencies. This means A1 and A2 essentially have 50 tickets.
    - Proceses can also transfer tickets amongst themselves.
- There is also a **stride** approach where we initially randomly pick a process, then increment its 'progress' after each time slice it receives by a large (fixed) number divided by the number of tickets that it is assigned. This figure is called its stride. We always pick the job with the smallest overall stride value first.

Chapter 10: Multi-Process Scheduling

- When we move to the multiprocessor world, the main difference between single-CPU and multi-CPU architecture revolves around caches (in particular, shared caches).
- Caches are based on the notions of spatial and temporal locality. The idea behind temporal locality is when a piece of data is accessed, it is likely to be accessed again in the near future (e.g. a variable in a loop)
    - Side note: **Cache coherence** focuses on the idea that two CPUs may not have the same view of a single piece of information. For example, CPU 1 changes the value of address A from D to E in its local cache. This value is scheduled to be propagated up to main memory but never does since the OS decides to stop running the program and instead move it to CPU 2. It then reloads the value at address A from Main memory only to find that its still D.
        - One way to avoid this problem is **bus snooping**, where each cache pays attention to memory updates by observing the bus that connects them to main memory. When a CPU then sees an update for data it holds, it invalidates its own copy.
    - Side note: Another issue with multi-processor scheduling is **cache affinity.** Cache affinity is the concept that it is more performant to run a job on the same CPU given that it builds up a fair amount of state in that CPU's cahce and TLBs.

Chapter 12: Memory Virtualization

- Every address generated by a user program is a virtual address.
- Memory is laid out as code first, then heap, then stack.
- Code is static (thus easy to place in memory) so it can be placed at the top of the address space (we know we wont need more space as the program runs)
- Memory is virtualized when the running program thinks it is loaded into memory ad a particular address (say 0) and has a potentially very large address space when in reality its loaded from an arbitrary physical address.
    - Side note: when you print out the value of a pointer, you are printing out a vbirtual address.
    - Side note: with regards to memory leaks, the operating system will reclaim all the memory of the process when the program is finished running.
        - Thus short-lived programs, leaking memory often does not cause any operationsl problems. For long-lived programs, leaked memory is a much bigger

issue and will eventually lead to a crash when the application runs out of memory.

Chapter 15: Address Translation

- With virtual to physical address translation, the hardware transforms each memory access, changing the virtual address provided toa physical address where the desired information can be located.
- From the programs' perspective, its address space starts at address 0.
- One technique for address translation is called **dynamic relocation** or **base and bounds**.
    - In this approach we need two hardware registers, the base register and the bounds register.
    - When the OS loads the program, it decides which location in physical memory it will reside in and places the starting physical address in the base register (*physical address = virtual address + base)*
    - *A **bounds*** address ensures that the address is within the confines of the address space.
    - The memory management unit helps with address translation.
        - The MMU generally manages memory using a free list.
    - The CPU generates exceptions in situations where a user program tries to access memory illedally.
- When the CPU needs to context switch, it must save and restore the base-and-bounds pair to and from the per-process structure (such as the process control block).

Page 151 has a useful chart.

- One downside of this approach is internal fragmentation, since the space between the stack and heap are large (and therefore wasted).
    - One way to address this is known as segmentation.
- The same physical segment in memory could be mapped to multiple virtual address spaces as code-sharing is still common in multiple systems today.

Chapter 16: Segmentation

- So far, int the text described above, we're been putting the entire address space of each process in memory.
- Segmentation solves the problem of supporting a large address space with a lot of free space between the stack and the heap.
    - The main idea is to have a base and bounds pair per logical segment of the address space. A segment is a contiguous portion of the address space of a particular length.
- With segmentation, instead of having a single base-bound pair, Code, Heap, and Stack all have their own base and bound register pairs.
    - One way to identify which base-bound pair is being referred to is via the **explicit** approach.
    - In this case, the top two bits are used to refer to which segment (base-bound pair) and the remaining is the offset from the base pair register. (**Page 159** diagram is useful).

- Segmentation sometimes leads to external fragmentation, where little holes of free space end up being created in physical memory (since segments are variable sized, the free memory gets chopped up into off-sized pieces). Thus when a request for a segment is made, no contiguous range of address space can be found to service this request. One solution is to compact memory around existing fragments.
    - This is slow since the OS needs to stop whatever process are running, copy the data over to the new physical address, change their segment register values to point to the new physical locations, and thus is expensive.
        - A simpler approach is to use a free-list management algorithm that tries to keep large extents of memory acailable for allocation (there are several algorithms which will be discussed later)
- Segmentation can alo lead to internal fragmentation if allocators hand out chunks of memory bigger than that requested (and hence the space used in this chunk is never used).

Chapter 17: Free Space Management

- How does the allocator know how big of a chunk to free when free is called? Most allocators store extra information in a header block which is kept in memory. The header block exists right before the handed-out chunk of memory. When *free* is called, the library uses pointer arithmetic to figure out where the header begins.
    - Thus, when a user requests N bytes, the library doesn't search for a free chunk of size N, it searches for N + header size.
- There are several mechanisms in allocation libraries:
    - Best fit – Find the smallest block that matches the capacity requirement and return a block of memory from it (leaves many small chunks)
    - Worst fit – Find the largest chunk that matches the capacity requirement and return a block of memory from it (tries to leave big chunks)
    - First fit – Find the first block that is big enough and return the requested amount to the user.
    - Next fit – Always start the search at the last point that it was done searching the last time. The idea is to spread the search throughout the list more uniformly (to avoid splintering the beginning of the list).

Chapter 18: Paging – Introduction

- Allocation of variable sized memory becomes challenging overtime as fragmentation occurs.
- Another approach is to chop the space into fixed-sized pieces. In virtual memory we call this idea **paging**
    - Instead of splitting the address space into some number of variable-sized logical segments (e.g. code, heap, stack) we divide it into fixed-sized units, each of which we call a page.
    - We view memory as an array of fixed-sized slots called page frames, each of these frames can contain a single virtual memory page.
    - To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a per-process data structure called a **page table**. The

main goal of the page table is to store address translations for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides.
    - To load data from the virtual address into a register we need to translate virtual address that the process generated. We do this by splitting the address into two components: the virtual page number and the offset within the page.
        - The first x bits set the virtual page number. If our virtual address space is 64 bytes large, for example, and the page size is 16 bytes, we need to select 64 / 16 (4). 4 can be represented in 2 bits.
        - The lowest x bits are the size of a single page frame (this part is known as the offset within the page). If we have a page size of 16 bytes, we need to know which byte to start at (4 bits can represent 16 bytes)
    - With the virtual page number, we index into our page table and find which physical frame virtual page 1 resides within.
        - For example, VPN -> PFN (physical frame number) of 1->7.

Where are Page Tables Stored?

- Page tables can be very large. Imagine a 32 bit address space with 4kb pages. The virtual address space splits into a 20-bit VPN and a 12-bit offset (10 bits are needed for 4kb but we need to round to the nearest bite).
    - A 20-bit VPN implies that there are 2^20 translations the OS would need to manage (roughly a million). Assuming we need 4 bytes per page table entry (PTE) to hold the physical translation plus any other useful stuff, we get an immense 4Mb of memory added for each page table. If we have 100 processes running, that would be 400mb just for address translations.

What is in a Page Table?

- The simplest form is called a linear page table, which is just an array. The OS indexes the array by the VPN and looks up the PTE (page table entry) at that index in order to find the desired PFN.
- The page table's first entry is located in the **pagetablebaseregister**. To get the PTE for a VPN (and thus be able to actually execute a fetch instruction) we do:

    VPN = (Virtual Address & VPN_MASK) >> SHIFT

    PTE = PageTableBaseRegister + (VPN * sizeof(PTE))

When the PTE is available the PhysicalFrameNumber (PFN) can be extracted from it. Now we can get the physical address corresponding from the virtual address as such:

    Offset = virtualaddress & offset_mask

    PhysAddr = (PFN << SHIFT) | offset

- Hence, ever instruction fetch will generate two memory references: one to the page table to find the physical frame that the instruction resides in, and one to the instruction itself to fetch it to the CPU for processing.

Chapter 19: Paging – faster Translations (TLBs)

- To speedup lookups and reduce the memory footprint of a page table, we are going to add a translation-lookaside buffer, or TLB.
- The TLB exists in the chip's memory management unit and acts as a hardware cache of popular virtual to physical address translations.
- Upon the first virtual memory reference, the hardware first checks the TLB to see if the desired translation is held there. If so, the translation is performed quickly without needing to reference / consult the page table.
    - If there is a miss, the hardware has to know exactly where the table table resides in memory (via the table base register) as well as their exact format. On a miss the hardware would walk the page table, find the corresponding page-table entry, and extract the desired translation. It would then update the TLB (virtual page number -> physical frame number) and retry the instruction.

What is in the TLB?

- The TLB is a fully associative cache that might have 32, 64, or 128 entries. A TLB looks like: VPN | PFN | Other bits
    - Other bits include the following:
        - Valid bit – whether a valid translation exists. In the beginning when a process is first created, there are no address translations cached, hence existing entries are not valid.
        - ASID (address space identifier)
        - Dirty bit

TLB and Context Switches

- The virtual-tophysical translation in the TLB are only valid for the process currently running.
- VPN 10 on Process A may map to PFN 100, but on Process B to PFN 170
    - One approach is to flush the TLB on context switches (setting the valid bit to 0).
    - Another approach is to set the ASID (address space identifier) field in the TLB so that each processs has its own unique ID.
        - With this approach, processes can share a TLB (therefore lower memory footprint)
        - Side note: Its possible two different processes share the same PFN despite having different VPNs. This occurs when two processes share a page (code page, for example).

TLB Cache Replacement Policy

- Since the TLB is a hardware cache, it must contend with cache replacement just like any other cache. One common approach is LRU

Chapter 21: Beyond Physical Memory

- Up till now everything we've talked about assumes that our code and data fits into physical memory. This is unrealistic.

- To support large address spaces, we cant assume that all pages reside in physical memory. The OS needs a place to stash away a portion of address spaces that currently aren't in great demand.
- Beyond just a single process, the addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently running programs.

Swap Space

- We need to reserve some space on disc to move pages back and forth. In operating systems we generally refer to this as **swap space**, because we *swap* pages out of memory to it and swap pages into memory.
    1. The OS reads and write to swap space in page-sized units. TO do so, the OS needs to remember the disc address of a given page.

Managing the machinery

- Now that we have an understanding of swap space, and a system with a hardware-managed TLB, lets talk about how swap is used:
    1. On a memory reference, the running process generates a virtual memory references (for instruction fetches or data access) and, in this case, the hardware translates them into a physical address before fetching the desitred data from memory.
    2. The hardware first extracts the VPN from the virtual address, checks the TLD for a match and produces the resulting physical address. It then fetches it from memory. Hopefully this is the common case (since its fast)
    3. If the VPN is not found in the TLB, the hardware locates the page table in memory (using the page table base register) and looks up the page table entry for the page using the VPN as an index.
        - If the page is valid and present in physical memory, the hardware extracts the page frame number (PFN) from the PTE, installs it in the TLB and retries the instruction. A TLB hit occurs and the resulting physical address is produced.
        - If the page is not present, and we wish to allow pages to be swapped to disk, we must add even more machinery. We know that the page is not present in physical memory based on the page table entry's present bit.
            - The act of accessing a page that is not in physical memory is commonly referred to as a page fault.
            - **Side note**; a page fault more generally can refer to any reference to a page table that generates a fault of some kind. This could include the page-nnot-present fault, but sometimes can refe to illegal memory accesses. When people say 'page fault' they usually mean 'page miss' (the program is trying to access parts of the virtual address space that the OS has swapped out to disk)

Page Fault

- When there is a page fault, the OS is invoked to service the page fault (a page-fault handler)
- If we're trying to access a page that has been swapped out, the OS will need to swap the page into memory in order to service the page fault.

    - The OS will use the bits in the PTE normally used for data such as the PGN of the page for a disk address. When the page fault occurs it will look at the PTE to find the address, issues the request to disk to fetch the page into memory.
    - When the disk I/O completes, the OS will then update the page table to mark the page as present, update the PGN field in the page-table entry to record the in-memory location of the newly fetched page and retry the instruction.
        - The next attempt may generate a TLB miss, which would be serviceds and update the TLB with the translation. Finally, a last restart would find the translation in the TLB and thus proceed to fetch the desired data or instruction from memory at the translated physical address

What if memory is full?

- Most of the time we can page-in a page from swap space. But, memory ay be full, so we need to page out one or more pages to make room for the new page the OS is about to bring in. This is known as the page-replacement policy.
- The optimal replacement policy is the one that replaces the page that will be accessed furthest in the future (throw out the page that is needed furthest down the line). You can never achieve the optimal, but you can get close. These are the three most important ones:
    - FIFO
    - LRU – Takes advantage of the principle of locality (uses historic accesses to predict which pages are 'hotter' and hence preserves those 'hot' pages in cache, kicking out older (cold) pages that weren't references as much.
        - Actually tracking this is computationally intensive, as every memory access now needs to update some internal data structure. Instead of LRU, most OS use LRU-approxtimation algorithm. This is done by setting a 'use-bit' (there is one use-bit per page)
            - Whenever a page is referenced, the use bit is set by hardware to 1. The hardware never clears the bit (sets it to 0), that is the responsibility of the OS.
            - To approxtimate the LRU, a clock algorithm is used. Imagine pages ordered in a clock circle. A random starting point is picked. The clock hand then iterates through all pages until it hits one with a bit of 0. On its way to that one page, it sets the use bit from 1 to 0 before moving to the next page. When it hits the one with 0, it evicts that page.
                - One small modification to this algorithm is the dity-bit. The dirty-bit is an additional piece of information. We recognize that paging out modified (dirty) pages would require the page be written back to dist post-eviction, which is expensive. If the page hasn't been modified, its eviction is free (can be reused for other purposes without additional I/O). Thus some virtual memory managers prefer to evict clean pages over dirty ones.
    - Random
- There are a couple of interesting properties of these algorithms:

1. When pages are accessed randomly, all these algorithms perform the same (hit rate is determined by the size of the cache)
2. When the cache is large enough to fit the entire workload (working set of pages), it doesn't matter which policy you use..
3. Algorithms that rely on the principle of locality (LRU and FIFO) benefit when the access pattern is non-random.
    a. Imagine the edge-case where the workload is 50 pages and the cache size is 49, a looping-sequential workload of 50 pages would never result in a cache hit. LRU for example, will always evict the page we're just about to access (e.g. to accommodate page 49 it evicts page 0, which is the next page itll access)

Thrashing

- What should the OS do when memory is simply oversubscribed, and the memory demands of the set of running processes simply exceeds the available physical memory?
    - In this case the system will constantly be paging. This is known as **thrashing**.
- Given a set of processes, a system that is thrashing could decide not to run a subset of processes, with the hope that the reduced set of processes' working sets fit into memory and can thus make progress. This approach is known as **admission control**.
    - This approach recognized that it's preferable to do less work better than do everything poorly. Other systems take a more draconian approach and just kill a process that is memory intensive, reducing the memory in a not-subtle way.
- Since paging to disc is expensive, the cost of frequent paging is prohibitive. The best solution to excessive paging is often simple, but more memory.