Note taken from Tannenbaum, OSTEP, modules, Computer Networking book

Written in chronological order of modules

Intro to OS

The **operating system** is the piece of software that allows running programs to execute on the system in an easy to use way. The primary method is **virtualization**, which is the act of taking a physical resource (memory, disk, network) and making a virtual representation. This abstraction allows a set of rules for sharing resources so no two programs can clobber each other. The operating system uses a set of **system calls**, which provide a standard library of interfaces for applications to use underlying resources.

The operating system runs in **kernel mode**, which is effectively unrestricted access to physical resource. Programs run in **user mode** and must request the OS to do kernel-level permission work. This request is called a **trap.** When this occurs a **trap handler** is activated, the OS works to do whatever the system call asked for (eg read 100 bytes from some file X), then a **return from trap** is fired that de-escalates to user-mode and returns control to the running program.

There also exists an **interrupt,** which is a hardware-signal to the CPU that it has urgent work to be scheduled (eg a keypress is an interrupt). When an interrupt reaches the CPU, it will finish the running process and switch back to the OS to run an **interrupt service routine** (ISR) in kernel mode to handle the interrupt.

Processes and Threads

A **process** is effectively a running program (code) alongside data and context. It's an abstraction that allows consistent interface with programs. A process has a **state** at any given point in time (5 states total). A process consists of an **address space**, or a virtualized view into a slice of main memory accessible to the program, registers like the program counter, stack and frame pointers (stateful information about what's the next line of code to execute in the program), and potentially I/O information like open files.

The 5 states a process can be in are

- New: the initialization phase. If a process is created by, eg unix Fork, there is some time needed to build a process and then create some initial values.
- Ready: when a process is prepared to do work, it can indicate to the CPU that it can to be scheduled. This state signifies that.

- Running: when a process is running on the CPU (and being time-sliced accordingly), the process is running
- Blocked: a process might be unable to run on the CPU because non-CPU, non-process work must occur. Eg network IO, if a process makes an HTTP request it takes time. The process is blocked. If a process forks a child and then **wait**s, it is blocked. Once the blocker is resolved, the process moves back to ready
- Exit: The process has finished its work and is ready for cleanup

A process can also be **suspended**, which is a stateful persistence process to take it off main memory into secondary storage to continue later.

We mentioned the associated data of a process. That entire structure is called the **process control block**. PCBs have states, so the CPU can queue them accordingly based on CPU scheduling algorithms.

A **thread** is an abstraction for a single running process. Specifically, a process indicates resource ownership and a thread represents process execution. Multiple threads can access the same resource via the address. Threads exist inside processes, and since they have their own stateful requirements they have their own **thread control blocks** (TCB). The idea of multiple execution threads in a process allows for **multi-threaded programs** to run **concurrently.** For example, a program might have execution paths that require network I/O, while other code can execute. A thread that is blocked on I/O can simply wait, while the rest of the program continues to execute (background saving in google docs, autocomplete dropdowns in UIs).

Threads only have 3 states **-** ready, running, blocked. Threads and concurrency open the door to concurrency problems (deadlocks, out of order execution, etc). Sharing data can yield nondeterministic results since there is no **atomicity** guarantee.

Concurrency and Deadlocks

A **race condition** occurs when N > 1 processes are working with some shared data and the final outcome depends on the order of execution of processes. To avoid race conditions, we need **mutual exclusion**, which prevents processes from working on a piece of data that is currently “owned” by another process. The region of code where shared data is accessed is called the **critical region**. The rules for a good solution to mutual exclusion are (Tannenbaum)

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block any process.
4. No process should have to wait forever to enter its critical region.

Peterson’s algorithm is a software solution to the problem, in which every thread has a bit it can toggle to request access to the critical region. If multiple threads were flagged, they would go sequentially and lower the flag.

From the hardware perspective, one way to do it is to disable hardware interrupts. This would stop a process from being interrupted in the middle of a critical region, but means it could choose to never give up control. Additionally, this really only works in the case of a single CPU core.

There are other low level solutions, such as **test and set lock** or **exchange**, which rely on writing a flag to memory that determines if in a critical region or not.

A common software implementation is a **semaphore**. A semaphore is an object with an integer value, which can be manipulated with two functions: **wait** (down) and **post** (up). The behavior is ass follows

- Wait decrements the value of the semaphore by 1. If the value is negative, the caller suspends execution (waits).
- Post increments the value of the semaphore by 1. If there are one or more threads waiting, wake one.

Typically, you initialize the value of the semaphore to 1 and if only one thread executes, the value will never go negative. If more than one thread trigger the wait, the following threads after the counter goes negative will just queue up and wait for threads to execute, post to the semaphore and allow threads to execute.

Deadlocks occur effectively when threads are all waiting to work on some resource owned by another thread such that the threads never can release control and continue execution.

There are four requirements for a deadlock. Mutual exclusion, hold-and-wait (thread locks resource A, wants access to a locked resourced B, so thread blocks), no preemption (OS cannot remove a resource from a thread), and circular wait (thread A waits for B, B waits for C, C waits for A).

We can try **prevention**, **avoidance,** and **detection** to alleviate the issue

Prevention: remove one of the requirements. Mutual exclusion is a requirement. We can get rid of hold and wait by making a thread declare all its locks at the same time. We can add preemption by adding a callback to resource locks - if a lock is requested and denied, the thread must release all its lock. We can prevent circular wait by ordering resources and preventing a thread holding a high-level resource from locking a low-level resource.

Avoidance: Running the **banker’s algorithm** to simulate all likely resource locks thread would request, and determine that we are not in an **unsafe state.** See wiki

Detection: Rather than avoiding deadlocks, accept that they happen and have some auditing process to see if a deadlock exists, and a rollback / undo process to go back to a working state, crashing / kill threads, or in other destructive ways clear the deadlock.

Memory Management

Memory management exists largely as a safety measure. When we have multiple processes running, we don't want process X to arbitrarily access memory allocated to process Y. We also need to be able to handle relocation - moving processes around in memory based on what's running, what isn't, what needs to be reallocated more space, etc.

A computer has physical memory addresses, ranging from zero to N. We create **logical** addresses, or virtual addresses, to trick all programs into thinking they start at zero and have access to all memory but in reality, a **hardware memory management unit** handles runtime conversion of eg pointers in the program to point to real memory. For example, if a program starts at address 2000 and a pointer in that program is to virtual memory address 100, the REAL location handled by the hardware unit will be 2000+100 = 2100.

We need strategies to allocate memory, or **partition** it. We can use **fixed size** allocation, but that is wasteful for programs that need very little space and just sit on unused, inaccessible memory. Eg allocating 256MB to Notepad that might only need 10MB.

A better solution is **dynamic partitioning.** Dynamic partitioning refers to just allocating enough space needed for a process and allocating more (potentially moving the process). However, this causes gaps (unused holes) in memory. Gaps that are too small to allocate for a process, so they stay unused. We call this **external fragmentation**. The act of moving processes around to remove those inaccessible slivers of memory is called **compacting.**

There are allocation strategies like best fit, worst fit, first fit, next fit:
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf

The **buddy system** is a process of taking all the available memory, taking some process and its memory requirement X, and dividing the number of available memory in half repeatedly until its the smallest multiple that would fit the process. This results in **internal fragmentation** of too much memory allocated, but it's a compromise between dynamic and fixed partitioning.

Memory uses **paging** today. Pages are small fixed-size windows of memory (4kb or so) that are basically windows into main memory. Each page belongs to a **page frame** which is an array of memory. Processes then use up a number of pages. Since pages are fixed size, it's possible to have some internal fragmentation inside a page but it's negligible due to page size being so small. Each process would have a **page table** that serves as a mapping of virtual pages to physical addresses.

The operating system uses **swap space** to move pages onto the hard drive when not in use in main memory. We use a bit set in the page in virtual memory called a **present bit** - is the page in main memory or swapped to disk? If the bit is set to zero, a **page fault** occurs and OS needs to figure out why a page that was swapped out was accessed - a bug, something that should have been in memory, something that shouldn't be accessible by current process, etc.

If we have pages, then we introduce multiple lookups - MMU finds which page frame we're in, then which address that belongs to. To make this faster, CPU designers add a cache called the **translation lookaside buffer** or TLB. The TLB just holds heavily accessed virtual-to-physical address translations.

Inevitably we have too many pages to store in main memory, so we need a **replacement policy**. Moving the wrong pages to disk will cause many page faults. There are strategies like LRU, FIFO, that are obvious caching policies.

The **clock algorithm** pretends all pages are in a circular list. The **clock hand** points to some page. When it's time to swap, we look at the page pointed to. Is the **use bit** set to 1? If yes, it was used recently, so set to zero and move clock hand to next page in list. Keep doing this until we see a zero, either circling the entire list and ending where we started or finding a page in the middle to swap out. (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys-policy.pdf)

Another strategy (The **PFF Algorithm)** is to look at the number of page faults that occur and manage the *ideal* rate. Too many page faults indicate we need more pages in memory, so add frames. Too few faults means we can offload some pages. The downside is it's bad for memory locality. There are adjustments like the **variable interval sampled working set or VSWS algorithm** that add a limit on the max number of time allowed between faults before throwing away unused pages, toggling use bits and letting the process continue.

Paging allows the capability to share memory/structures - that is, the Unix **fork** command has no reason to allocate new pages when a child process is created. Instead it flags memory as **readonly**, creates a child process and if the child process tries to write, the OS duplicates the pages, creates page table entries and performs **copy on write** semantics such that duplication of shared data is postponed until required.

Computer Networks

What is the internet? **Host** devices are connected by **communication links** and **packet switches**. Links can transmit data at some rate of bits / second, and we call that rate the **bandwidth.** Hosts send data as small segments, annotated with header bytes. We call these segments **packets**. A switch takes a packet from some sender, and sends it to some outgoing link to get it closer to the destination. These switches are devices like **router** and **link-layer switches**. The **route** of the packet is the trace of all the switches it passes through to get to the end.

Protocols dictate how information is sent and received. **TCP (Transmission Control Protocol)** and **IP (Internet Protocol)** are the two most commonly used protocols. The principal protocols of the internet are known as **TCP/IP.**

Applications that involve multiple devices that exchange data are called **distributed applications.**

How does one program on one system communicate with a program on another system?

A **socket interface** specifics how a program on a device asks the Internet (infrastructure) how to deliver a packet. That interface includes **IP addresses** and **port numbers.**

When we discuss the **network edge** we think about hosts broken down into subcategories of **clients** and **servers** - clients might be my laptop or phone, servers would be some VM in AWS serving Netflix. These devices exist at the edge of the network.

Access networks connect an edge router (the device in my bedroom) to somewhere else. A **home access** network could be my FIOS connection, DSL or cable. There also exists satellite and dial up. **Enterprise access** such as ethernet and wifi for corporate or university campuses. This **local area network** (LAN) can be constructed wirelessly to serve a **wireless local area network** (WLAN). For mobile device access, we have **wide-area wireless access** such as 3G or LTE for my iPhone.

Packet switching

Most switches use **store-and-forward transmission** - the switch needs the entire packet before it can start transmitting the first byte. My router needs to **buffer** the packet as it comes through until the entire packet is available. Switches have some **output buffer** or **output queue**. Since switches have multiple outbound links, they have a buffer per link. This then adds **queueing delay** and if the queue is full when another packet tries to buffer, we have **packet loss**.

Client server architecture is a design that has two entities - clients and servers. Servers are always-on hosts with a permanent IP and sit in data centers to scale. Clients communicate with server, intermittently connected and have dynamic IPs (maybe). They don’t communicate with each other directly.

Peer to peer architecture has no always-on server. Each client peer and connect to each other to request data (BitTorrent).

We also have **process communication**. Processes run within a host, and the processes use **inter-process communication** (IPC) to communicate. Processes send and receive to and from

sockets. A process has an identifier - that identifier is the IP address of the host and a port number. Eg gmail.com:443 is TLS web server process for gmail.

Application Layer

Application layer protocol define the types of messages exchanged, syntax, and semantics. Protocols include HTTP or FTP. The **transport layer** transports application-layer messages on top of it. Protocols include TCP or UDP. These protocols have rules around throughput, reliability, transmission rate, etc.

The web uses **HTTP** or **hypertext transfer protocol**. Clients are browsers (usually) that send requests, servers return responses. HTTP is stateless, the server knows nothing about past client requests. We use TCP and make a connection (socket) on port 80. Server accepts socket. HTTP messages (application layer) are sent between, then the socket is closed.

HTTP connections can be **persistent** or **nonpersistent**. If nonpersistent, need to keep opening new connection per object of data. Persistent connection allow for multiple objects.

Client initiates a TCP connection to server. Server accepts and returns notification of acceptance to client. Client gets connection, sends a request message in the TCP socket. Server receives it, sends a message back. Client parses and does whatever while the server closes the connection.

The **round-trip time (RTT)** is time to server and back. HTTP response time is 2xRTT + file transmission time. (1 RTT to open socket, 1RTT to send request/response, add time for whatever file needs to be built). Since persistent HTTP leaves the socket open, we can reduce to 1RTT for subsequent requests.

HTTP continued

The HTTP spec defines **status codes**, such as 200 for success, 404 for File Not Found, 301 Moved Permanently, etc.

Cookies are used to create server-side user association. It's made up of a cookie header line in the response of the first request, the cookie header in all requests after the first, the cookie file on user host (in the browser), and persisted in the backend server database. Cookies are used for authorization, shopping carts, recommendations, ads, etc.

Web caches or **proxy servers** sit in between clients and servers to store some server data and serves some requests faster. For example, hitting a CDN to get static assets. Caches are both clients (to origin server) and servers (to you, the user client).

We don’t want to use a stale cache, so HTTP defines a **conditional GET** - a cache header called **if-modified-since**: **<date>** that expects some degree of recency.

We also have **File Transfer Protocol (FTP)** to send files over port 21 from one host to another.

Email supports different protocols. **SMTP (Simple Mail Transfer Protocol)** lets servers connect with other servers to send emails. Servers have mailboxes and queues of outgoing mail. SMTP is built on top of TCP over port 25. To retrieve emails, we also have **POP (Post Office Protocol)** or **IMAP (Internet Mail Access Protocol)** or even HTTP. POP3 was designed to download-and-delete emails, then later download-and-keep. POP3 is stateless, so read-status is not saved. IMAP keeps messages on the server and maintains user state across sessions.

DNS

Domain Name System (DNS) is how we map IP addresses to domains. It’s a distributed database in a hierarchy of **name servers** implemented in the application-layer (so it exists at network edge). The core is a root DNS server for .COM, .ORG, .EDU. Each DNS server then might have eg yahoo.com, nthomas.org and inside where we might go to the yahoo.com server to get mail.yahoo.com

Top Level Domain (TLD) servers are responsible for the TLD (.com, .org, .us) domains. **Authoritative DNS Servers** provide DNS mappings for an organization, either owned by an entity or a service provider (eg godaddy). **Local DNS Name Server**s can be owned by ISPs, companies. Also called the **default name server** - serves as a cache and can read upstream.

Records are insert into DNS by **registering** with a registrar. We provide the IP address and names of authoritative name server, then set up name records (A record) and mail records (MX records).

DNS can be attacked with redirects like man-in-the-middle, DNS poisoning (breaking the cache), and other strategies.

Peer-to-Peer

Clients connect to a network and are noted by a **tracker. Client** is registered, and begins downloading chunk from a subset of peers. Once the client has the file they (hopefully) stay in the network to keep uploading the file.

The **distributed hash table** is a database for distributed P2P that has mapping of filename to IP address. A client request some file from the DHT, the DHT returns all matching IP addresses.

Transport Layer

Transport layer provides protocols for **logical communication** between applications on different hosts - making it seem as if the processes are running on the same machine. We send **segments** or **packets** of data.

The internet has two distinct transport-layor protocols - UDP and TCP. UDP is connectionless, no guarantees, faster. TCP has handshake, kinda-guarantees, slower, congestion control.

The IP Service model is a **best-effort delivery system.** No guarantees but does everything it can. As a result it is **unreliable.**

IP delivery is **host-to-host**. UDP and TCP delivery is **process-to-process.**

Delivering data to the correct socket (host IP, port) in the transport-protocol is called **demultiplexing**. Adding headers, socket into and passing off to network protocol is called **multiplexing**.

Example of UDP? DNS - no need to handshake

Pros of UDP? Fine grain control at application level of data, no connection establishment, no connection state, small packet header overhead.

UDP segment - source port #, destination port #, length, checksum, application data

Reliable data transfer - creating a framework to ensure no corruption of data, missing packets, out-of-order messages. Role of the protocol to implement on top of an unreliable protocol, eg TCP is an RDT built on something unreliable in IP.

http://www2.ic.uff.br/~michael/kr1999/3-transport/3_040-principles_rdt.htm

Keywords - pipelining (queue up messages rather than wait on ACK of each), go-back-N (unroll the pipeline by N and use a sliding window),

Network Layer

Two roles, forwarding and routing

Forwarding - move a packet from router input link to router output link
Routing - determine the route for a packet from source A to destination B

Data plane - area where per-router function operate on datagrams (packets) forward from an input link to output link
Control plane - network wide logic of routing packets

Network layer provides **guaranteed delivery (with bounded delay), in-order packet delivery, guaranteed minimal bandwidth, security**.

Routers have input ports, switching fabrics (network router to get inputs to outputs),, output ports

Destination-based forwarding - given a packet and its specific destination, make decisions on which output port. Use some lookup table that takes, eg the prefix of the destination address to an output port, needs to be fast at gigabit speeds. Special hardware and in-memory data structures.

Switching fabrics can be switching via memory (a routing processor writes into memory, looks up output port, then writes into output), switching via bus (write to bus, let the correct output pull it and the rest ignore it), or switching via interconnection network (use a crossbar switch to push into bus that only feeds to the right output port)

Queueing can occur at the input (too slowly writing it out), called **head-of-line blocking** (output is free but input is too slow). Can occur at output (can cause dropped packets with memory issues)

Packet scheduling algos - FIFO, priority queueing, weighted fair queueing,

Network Address Translation - what do we do since IPv4 addresses are running out? Create a private network where the router attaches metadata to outgoing requests (a port) and let all devices on the network appear to be on the same IP to the outside. Uses **DHCP** (Dynamic Host Configuration Protocol) to assign private IPs.