Operating systems provide user programs with a better/simpler/cleaner model of the computer & to handle managing resources throughout the machine.

① WEB BROWSER   EMAIL READER   MUSIC PLAYER } user mode } Software

② user interface Program } user mode } Software

③ operating system } kernel mode } Software

④ HARDWARE

the "levels" of a machine

They are built to evolve due to how hefty they are,
and they manage/protect memory, I/O devices, and resources: keeps track of which program is using which resource, to track resources, grant resource requests, account for usage, and mediate conflicting requests from different programs & users.

Resource management includes multiplexing: sharing resources in time & space

Time multiplexing: programs/users take turns using it (issues of scheduling and fairness)

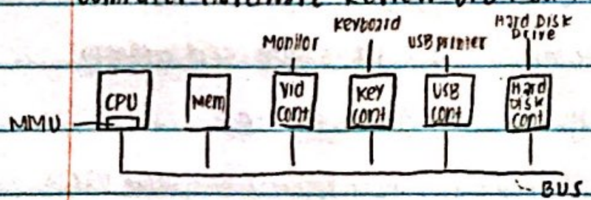
Space multiplexing: resources are divided into parts (issues of fairness and protection)

History of operating systems (briefly)

Vaccuum Tubes
Transistors and Batch systems
ICs and Multiprogramming (integrated circuits)
Personal Computers
Mobile computers

just to get a fun idea of how things progressed

Computer Hardware Review b/c I sucked Ass in 110



CPU: fetches instr from mem, executes them
has general registers (to hold vars / temp results)
& special registers: program counter (mem addr of next instr)
stack pointer (top of curr stack in mem)
PSW (comparison instr for I/O & sys calls)

Now we introduce multithreading (also known as hyperthreading)
It allows the CPU to hold the states of different threads and switch back and forth on a nanosecond time scale. (it doesn't offer true parallelism, only pseudoparallelism)

Memory Overview

Typical Access Time		Typical Capacity	
1 nsec	Registers	< 1KB	smaller
2 nsec	Cache	4MB	
10 nsec	Main Memory	1-8GB	
10 msec	Magnetic Disk	1-4TB	larger

Registers: internal to CPU, programs must manage them

Cache Mem: mostly controlled by hardware, most heavily used cache lines are either located inside or close to the CPU. cache miss goes to memory, so most OS's keep (pieces of) heavily used files in main memory to avoid fetching from disk

BUT, when do we put a new item into the cache? which cache line do we put it on? which item do we remove from the cache when out of space? and where do we put this newly evicted item in memory?

Disks (magnetic/hard disks) are mechanical devices

They have two surfaces per platter, and one read/write head per surface and information is written onto it in a series of concentric circles.

At any given arm position, heads can read tracks, which are divided into sectors (typically 512 bytes/sector). outer disks = usually more sectors.

one cylinder to next: 1msec

Random cylinder: 5-10 msec

once on correct track, sector to rotate under head: 5msec - 10msec

once sector under head, read/write: 50MB/sec - 160MB/sec

SSDs store in flash memory (no moving parts), data not lost when powered off.

Virtual memory: can run programs larger than physical mem by placing them on the disk and using main mem as a kind of cache for most heavily executed parts

It requires mapping mem addrs on the fly to convert the address the program generated to the physical address in RAM where the word is located. done by the MMU

And as a note, switching from one program to another is called a context switch, and sometimes it is necessary to flush all modified blocks from the cache and change the mapping registers in the MMU while doing so (but it's v expensive, so we try to avoid this)

I/O usually consists of: the device itself

- it's controller (chip/ set of chips that controls device)

accepts commands from OS and carries said commands out

The software that talks to a controller (gives commands/accepts responses) is a device driver.

Each controller has a small # of registers for possibly specifying the disk address, memory address, sector count, and direction (read or write). These registers can be mapped into the OS's address space. If this happens, no special I/O instructions are required, and user progs can be kept away from hardware using base and limit registers.

I/O can be done using:

Busy waiting - user prog issues sys call, kernel translates to procedure call, to driver
driver starts I/O & sits in tight loop polling device to see if done.
when I/O = complete, driver puts data (if any) where needed, and returns.
OS then returns control to caller.

Interrupts - starts device & asks to give interrupt when finished. Driver
then returns, and OS blocks caller if needed/looks for other work to do.
when controller detects end of transfer, interrupt is generated to signal completion.

DMA (Direct Memory Access) - special hardware in form of chip that can control
the flow of bits b/t mem and some controller w/o constant CPU intervention.
CPU sets up DMA by telling it how many bytes to transfer, the device/mem addrs
involved, and the direction. when DMA chip is done, it causes interrupt (process discussed later)

Overviews of operating system concepts

Processes are programs in execution. They have their own address space (contains executable program, program's data, and stack), and registers.
Related processes cooperating to get a job done often need to communicate and synchronize activities (this is called IPC, interprocess communication)

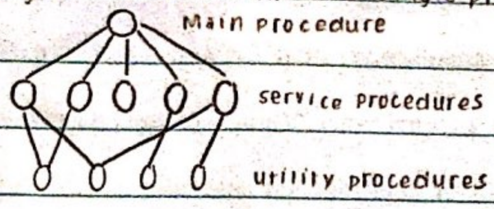
Address space (typically 2^{32} or 2^{64} bytes each), further discussed later

1.6 talks about different calls but it's p straightforward
important ones to note are link, fork, and the rest are common sense

we'll go more into files and file systems later

Operating system structures

Monolithic: (most common organization) OS runs as a single program in kernel mode.



Layered: organize the OS as a hierarchy of layers, each one constructed upon the one below it

layer	function
5	the operator
4	user programs
3	input/output management
2	operator-process communication
1	memory and drum management
0	processor allocation and multiprogramming

Microkernel: put as little as possible in kernel mode. Let microkernel handle interrupts, processes, scheduling, and IPC.

Client-server model: servers provide some service each, and clients use these services. Communication occurs w/ message passing.

Virtual machines: exactly that, virtual machines that can each run different operating systems, all on the same computer

Exokernels: instead of cloning the actual machine (like VMs), you can partition it (give each user a subset of the resources). Exokernel allocates resources to VMs and checks attempts to use them to make sure no machine is trying to use another's resources.

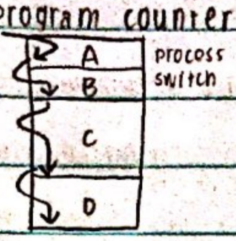
END OF REVIEW/GENERALIZATIONS/OVERVIEWS, let's jump in :)

Processes and Threads

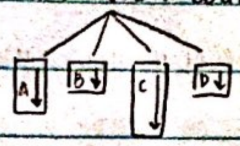
A **process** is the abstraction of a running program, and in any multiprogramming system, the CPU switches from process to process quickly. The CPU is only ever running one process, but these speedy switches give the illusion of the CPU running multiple processes at a time.

There's only one physical program counter, so when each process runs, it's logical PC is loaded into the physical one. When finished (for time being), it's stored in the process' stored logical PC in memory.

One program counter



Four program counters



Please note, only one PC/program is active at once

Processes may be created due to four principal events:

System initialization

Execution of a process-creation system call by running a process

A user request to create a new process

Initiation of a batch job

and processes that stay in the background to handle some activity are called daemons

And on the opposite end, processes will terminate due to:

Normal exit (voluntary)

Error exit (voluntary)

Fatal error (involuntary)

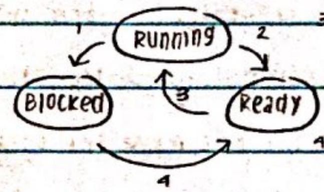
Killed by another process (involuntary)

Processes may be in these three states

1) Running (actually using the CPU at that instant)

2) Ready (runnable; temporarily stopped to let another process run)

3) Blocked (unable to run until some external event happens)



1) processor blocks for input

2) scheduler picks another process

3) scheduler picks this process

4) input becomes available

To maintain the process model, the OS maintains a process table (w/ one entry per process)

Each entry contains that process' state, program counter, stack pointer, memory allocation, status of its open files, accounting/scheduling information, & everything else about the process that must be saved when switched from running to ready/blocked state.

When a disk interrupt happens, the running process' PC, program status word, and sometimes register(s) are pushed onto the (current) stack by the interrupt hardware.

The computer then jumps to the address of the interrupt service procedure. which decides what to do next

Processes are interrupted thousands of times during execution, but after each interrupt, the interrupted process returns to precisely the same state it was in before the interrupt occurred.

Skeleton interrupt - 1) hardware stacks PC, etc 2) hardware loads new PC from int vector

3) assembly saves registers, sets up new stack 4) C interrupt service reads/buffers input

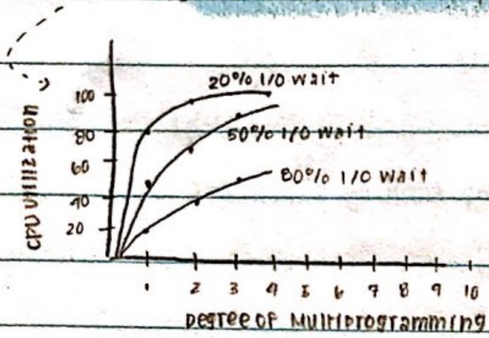
5) scheduler decides which process to run next 6) assembly starts up new curr. process

When multiprogramming is used, CPU utilization can be improved.

Suppose that a process spends a fraction "p" of its time waiting for I/O to complete.

With "n" processes in memory at once, the prob that all processes are waiting for I/O is "p^n"

so **CPU utilization = $1-p^n$**



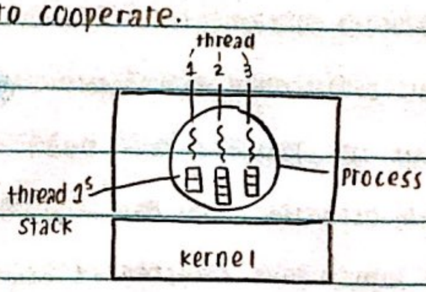
Threads (aka, a kind of process within a process)

Threads are a miniprocess that help make the programming model simpler.

They add the ability for parallel entities to share an address space and all its data (aka they share common memory) amongst themselves. Not only that though. They're lighter weight than processes, so easier/faster to create and destroy. Threads yield no performance gain when they're CPU-bound, but when there's substantial computing & I/O, they speed up the application.

All threads have the same address space, which means they also share the same global variables. Every thread can access every mem addr within the process' addr space, so one thread could read/write/wipe out another thread's stack. No protection is necessary b/t threads though b/c it's impossible and unnecessary. They're built to cooperate.

Per-process items	**Per-thread items**
addr space	program counter
global vars	registers
open files	stack
child processes	state
pending alarms	
signals/signal handlers	
accounting information	



so... threads... do they go in the kernel?

At the user-level?

BOTH???

You CAN put the threads package entirely in user space.

The kernel will know nothing about them, and will see them as one single-threaded process, and this can be good for implementing threads on an OS that doesn't support them.

If managed in user space, each process needs its own thread table to keep track of the threads within it. *keeps track of each thread's PC, stack pointer, registers, state, etc*

Thread scheduling/switch is quite fast, & user level threads allow each process to have its' own custom scheduling alg. Usr lvl threads are also more scalable.

User level threads interfere with the implementation of blocking system calls though, because one blocked thread shouldn't affect another.

Code placed around the system call to do checking (a jacket/wrapper) is inefficient, but helps to see if a call will block.

User level threads: pointless for CPU bound applications

You can also put the threads package in the kernel. (this tends to be the better/more popular option)

The kernel has a thread table that keeps track of all threads. If a process wants to make or destroy a thread, it makes a kernel call. The cost is more expensive, so thread recycling is utilized.

Kernel threads: don't require any new, nonblocking system calls, & page faults are more efficient to work around, BUT overhead is incurred b/c of high-cost system calls.

Hybrid approaches also exist. Kernel-level threads can have user-level threads multiplexed onto them.

Scheduler Activations mimic the functionality of kernel-level threads, but with better performance & flexibility like those in user space.

They use upcalls so that the run-time system can reschedule threads when a thread has blocked.

Pop up threads can be created to handle incoming messages.

They're quick to create, and reduce the latency b/t message arrival and the start of processing.

Issues with making single-threaded code multithreaded: - stack management

- variables global to a thread but not global to the entire program, because other threads should leave them alone
- many library procedures are not reentrant (not designed to have 2nd call made while prev has not finished)
- some signals are thread-specific (where to direct by kernel? who catches what)

IPC, Interprocess communication

Processes need to communicate with other processes, so (in a well-structured way not using interrupts)

How can one process pass information to another

How can we prevent two processes from getting into each other's way

and How do we sequence processes if dependencies are present?

Race conditions

Processes that are working together may share common storage, so race conditions can arise when two or more processes are reading or writing some shared data and the final result depends on who runs precisely when.

To prevent trouble, we will find ways to prohibit more than one process from reading and writing the shared data at the same time.

This is called mutual exclusion (making sure that if one process is using a shared variable/file, other processes will be excluded from doing the same thing)

Critical regions/sections are the part of the program where shared memory is accessed. To avoid races, we will prevent processes from ever being in their critical regions at the same time. Conditions:

1. No two processes may be simultaneously inside their critical regions
2. No assumptions can be made about speed / # CPUs
3. No process running outside critical region may block any process
4. No process should have to wait forever to enter its critical region

We can achieve mutual exclusion by:

Disabling interrupts: disable interrupts just after entering crit region, re-enable just before leaving it. This ensures no process switching will occur, but this can be dangeous, or difficult with multiprocessors.

Lock variables: test lock, if lock = 0, set to 1 & enter region. if lock = 1, wait until lock = 0 to enter. but what if two processes see lock = 0 before first sets it to 1? race occurs.

Strict Altercation: turn = 0, P0 sees this and enters crit. P1 also sees this, and sits in a tight loop until it's 1. Continuously testing a variable until a value occurs is busy waiting, and a lock that uses busy waiting is a spin lock.

It can violate the third condition above, though.