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
③ operating system } kernel mode
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)

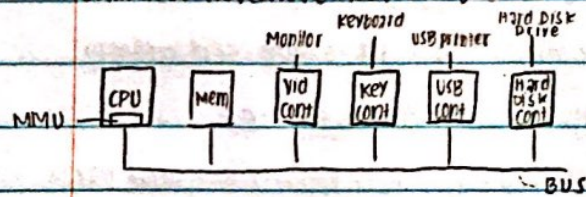
Space multiplexing: resources are divided into parts (issues of fairness and protection)

History of operating systems (briefly)

Vacuum 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



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 - 160 MB/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

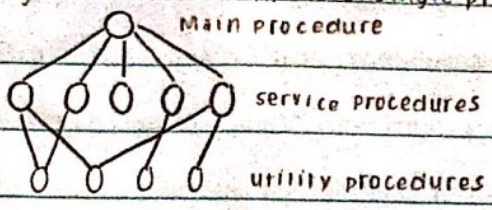
1.6 talks about different calls but it's p straightforward

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.

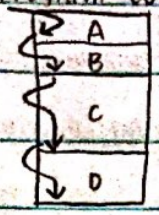
END OF REVIEW/GENERALIZATIONS/OVERVIEWS, let's jump in 😊

Processes and Threads

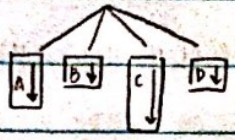
A process is the abstraction of a running program, and in any multiprogramming system, the CPU switches from process to process quickly. The CPU is only ever running one process, but these speedy switches give the illusion of the CPU running multiple processes at a time.

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)

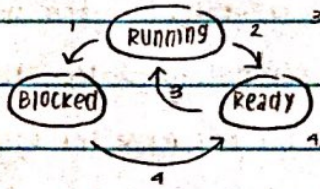
Killed by another process (involuntary)

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

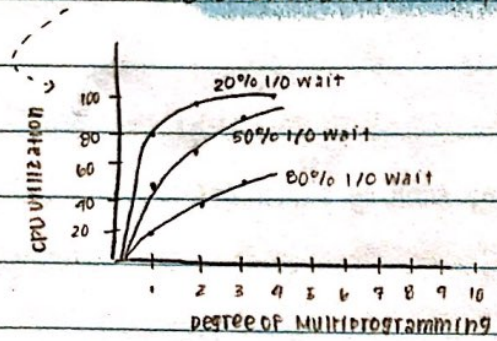
5) scheduler decides which process to run next 6) assembly starts up new curr. process

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 = $1 - p^n$



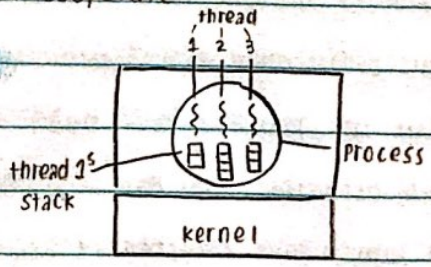
Threads (aka, a kind of process within a process)

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. User 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 bit 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 dangerous, 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.

The producer-consumer problem / bounded-buffer problem

Two processes share a buffer (fixed size). Producer puts info into it, and consumer takes info out. If producer wants to put in item, but buffer is full, it can go to sleep, and be awakened when the consumer has removed items. Variable count keeps track of items in buffer. But a race condition can occur because count access is unconstrained. A wakeup bit could be set to prevent misunderstandings when it comes to processes being "logically" asleep (in a simple scenario).

Semaphores (usually called down & up), change their values as a single, indivisible atomic action (so once a semaphor operation has started, no other process can access the semaphor until the action is completed/ blocked) They can solve our bounded buffer problem with 3 semaphores:

empty: # of slots that are empty

full: # of slots that are full

mutex: stops prod & cons from entering buffer at same time

They also help with synchronisation.

A mutex (simplified vers of a semaphor) is useful when a semaphor's ability to count is not needed. It's a variable that can either be in a locked or unlocked state. A futex is a feature of Linux that implements basic locking, but avoids dropping into the kernel unless it really has to, and pthreads not only offer mutexes, but also condition variables, which allow threads to block due to a condition not being met.

Monitors are a higher-level synchronization primitive. They're a collection of procedures, variables, and data structures all grouped together in a special kind of module/package. Only ONE process can be in a monitor at any instant. They are an easy way to achieve mutual exclusion, and the condition variables wait and signal allows processes to block when they cannot proceed, and the wait must come before the signal to prevent lost signals.

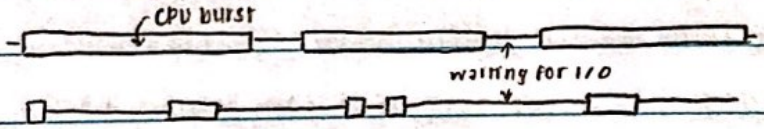
Message passing allows information exchange between machines, using the two primitives send and recieve. If a message is recieved, an acknowledge message is sent back.

The last synchronization mechanism, barriers, is intended for process groups. No processes may be allowed to proceed until they all reach a specific phase. When a process reaches a barrier, it is blocked until all processes reach the same barrier. This allows groups to synchronize.

SCHEDULING

Multiple processes or threads may be compering for the CPU at the same time, and it's up to the OS's scheduler to use scheduling algorithms to determine which ready process will run next. The scheduler must make efficient use of the CPU because process switching is expensive. A switch from user → kernel mode occurs, & the state of the curr process is saved (along w/ storing registers and process table and sometimes memory map (page table)). Next process is selected by scheduler, MMU is reloaded w/ mem map of new process, and mem cache & related tables may now be invalid, so those will be dynamically reloaded from main mem twice.

Processes alternate bit bursts of computing with I/O requests:



↑ more common as CPUs get faster

When to schedule?

① When new process created ② process exits

③ Process blocks on I/O ④ I/O interrupt occurs

Nonpreemptive (only option if no clock available) scheduling algs let a process run until it blocks or releases CPU, while preemptive only lets it run for a maximum of fixed time.

And scheduling algorithm goals differ for each system

Batch systems want to maximize throughput (jobs/hr), minimize turnaround time (average time), and keep the CPU busy at all times

Interactive systems want to minimize response time & meet expectations

Real-time systems want to meet deadlines & avoid quality degradation

& All systems want fairness, balance, and policy enforcement

BATCH SYST.

INTERACTIVE

Scheduling Algorithms:

FCFS: processes assigned to CPU in order they request it, and are allowed to run for as long as it wants to.

SJF: scheduler picks the shortest jobs in the input queue to run first. Optimal only when all jobs available simultaneously.

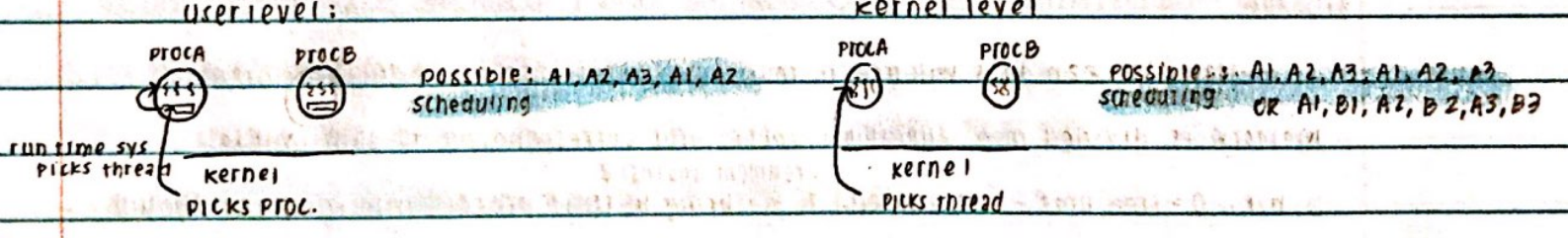
Shortest Remaining Time Next: Again, optimal when all run times known in advance. Allows new short jobs to get good service.

Round Robin: each process assigned time interval (quantum) during which it is allowed to run. Too short of a quantum causes too many switches & lowers CPU efficiency, but too long causes poor response to interactive requests.

Priority scheduling: Runnable process w/ highest priority allowed to run. Might want to incorporate RR to prevent starvation.

Lottery scheduling: give processes lottery tickets for various resources. It's highly responsive, and can be used to solve otherwise difficult scheduling problems.

Thread scheduling differs depending on user-level vs kernel-level.



Classical IPC Problems include:

Dining Philosophers → (watch videos online for clarity) use semaphores!

Readers and Writers → see Courtois et al.

MEMORY MANAGEMENT

Two problems have to be solved to allow multiple applications to be in mem at the same time without interfering with each other: protection & relocation

An address space (set of addresses that a process can use to address memory) (and each process has one) is a good solution for this.

It's a bit more difficult to give each program an address space. Let's start with a simpler (but dated) implementation.

we will use a simple version of dynamic relocation to accomplish this, for our solution will map each process' address space onto a different part of physical memory using base and limit registers.

Programs will be loaded into consecutive memory locations wherever there is room.

The base register is loaded with the phys addr where the program begins, the limit register is loaded w/ the length of the program.

Base+Physical Addr → sent on memory bus → memory reference reached

We can't fit ALL of the processes into physical memory, and we have two strategies to help deal with this:

swapping (bringing in each process, running it for a while, then putting it back to disk) (in its entirety)

& virtual memory (allows programs to run even when they're partially in main mem)

Swapping:

It can create holes in memory, and we could either swap into those holes (if space allowed us to), or we could compact by moving all processes downward.

We should leave room for growth, though, due to a program's data segment/stack.

We can manage free memory with bitmaps and free lists:

BITMAPS

A memory of 32n bits will use n map bits. Large allocation = smaller bitmap.

Memory is divided into allocation units, and corresponding to each unit is a bit. 0 = free unit, 1 = occupied. If we bring k-unit process into memory though, we'd have to find a run of k consecutive 0s in the map. (Argument against?)

LINKED LISTS

We can also keep a list of allocated and free segments, where a segment either contains a process, or a hole between two processes.

There's algorithms used to allocate memory for processes, listed below:

First Fit: fast, because it searches as little as possible

Next Fit: finds next available hole, slightly worse performance than

Best Fit: slower, and lots of wasted mem b/c tiny holes

Worst Fit: not a good idea, but new hole kinda useful

VIRTUAL MEMORY

We need to be able to run programs too big to fit in MM, and have systems that can support multiple programs running simultaneously (that might collectively exceed memory).

↳ Virtual memory: the idea:

each program has its own address space, which is broken up into chunks called pages

Page: contiguous range of addresses

the pages are mapped onto physical memory, but they don't all have to be in physical memory. When the program references a part of its address space that's not in physical memory, the OS is alerted to go get the missing piece and reexecute.

But how does it work?

It uses a technique called paging:

program-generated addresses are called virtual addresses, and form the virtual address space. When virtual memory is used, virtual addresses go to an MMU that maps virtual addresses onto physical mem addrs. Virtual address space consists of fixed-size units called pages. Corresponding units in phys mem are called page frames.

virtual addr → MMU → finds page frame → outputs phys addr to bus

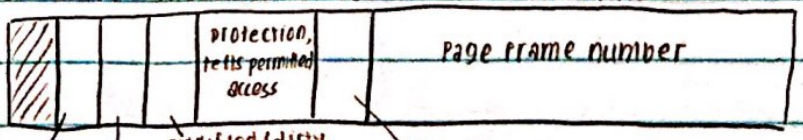
A present/absent bit keeps track of which pages are physically in memory

If a program references an unmapped address, the MMU notices the page is unmapped, and causes the CPU to trap to the operating system (page fault)

OS then picks little used page frame, writes its contents back to disk, then fetches page just referenced into the freed page frame & restarts trapped instr.

A virtual address is a series of bits, and a portion of these is the page number, (virtual page # & offset) which is used as an index into the page table (yields # of page frame of that virt page)

A page table entry (though it varies) can look like this:



and note: Page table doesn't contain info about disk address of unreferenced page. that's kept in software tables.

Two major issues must be faced in any paging system:

1) The mapping from virtual to physical address must be fast (b/c virtual-to-physical must be done on every memory reference)

2) If the virtual address space is large, the page table must be large (b/c virt addr ≥ 32 bits, ex: 4KB page size → 32-bit addr = 1 million pages!)

Most programs make many references to a small number of pages....

So let's equip computers with TLBs (translation lookaside buffer) to map virtual to physical addresses without going through a page table. They contain a small # entries, & contain information about most recently referenced pages.

soft miss: page not in TLB, but in mem

hard miss: not in TLB, not in mem, in disk

Multilevel Page Tables

contains PT1 field as an index into first PT, and entry located there yields addr/page frame number of a second page table, indexed into with the PT2 field. Page frame number there + offset = phys addr.

Inverted Page Tables

one entry per page frame in real memory, instead of one entry per page of virtual addr space. the entry keeps track of which (process, virtual page) is located in the page frame. (Hash table)

They save space, but virtual-to-physical translation is harder.

Page Replacement Algorithms

Pages are chosen to be evicted when faults occurr, but how should we determine this? The optimal alg is to remove the page that will be referenced latest in the future, but this is unrealizable.

Realizable ones are:

Not recently used: status R = referenced, M = modified, and periodically, R bit is cleared. When page fault occurs, OS inspects pages. They will be: 0) not referenced, not modified 1) not referenced, modified (happens when R is cleared)

2) referenced, not modified 3) referenced, modified

The alg then removes a page at random from the lowest numbered (nonempty) class. Easy to understand, moderately efficient to implement, adequate performance.

lots of page faults: thrashing

First-In, First-Out

The OS maintains a list of all pages currently in memory, with most recent arrival at the tail, least recent at head. Upon a page fault, page at head is removed & new page is added to tail.

BUT, the oldest page may be useful!

Second-Chance

A slightly modified FIFO. We will inspect the R bit of the oldest page, and only if it is unreferenced will it be replaced. Otherwise, back to the tail of the list it goes, and we continue looking for an old & unreferenced page to boot. All pages referenced? Pure FIFO, so alg will always terminate.

Clock-Replacement

Second chance, but without moving pages around on its list. Instead, circular list, where hand points to oldest page.

Least Recently Used

Throw out page that's been unused for longest amount of time.

Not cheap though, because we have to keep a linked list of all pages in memory, and update the list with every reference.

Not Frequently Used

counters roughly keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement. We can shift counters right one bit with every tick to help simulate LRU.

000 —referenced→ 100 —not referenced→ 010 —not referenced→ 001 —and so on→ ....

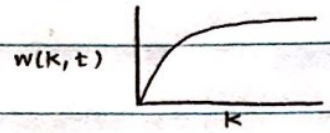
this is a local page rep. algorithm

The Working Set

Processes are started with none of their pages in memory, and the first fetch causes a page fault. Pages are brought in, and eventually a process has most pages it needs and runs w/ relatively few faults. This is *demand paging*, for pages are not loaded in in advance.

The set of pages that a process is currently using is its *working set*, and the working set model is when a paging system tries to keep track of each process' working set and make sure it's in memory before letting the process run. (prepaging)

At any instant in time, t, there exists a set consisting of all the pages used by the k most recent memory references. This set, w(k,t) is the working set. The limit of w(k,t) as k becomes large is finite b/c a prog cannot reference more pages than its working set contains.



→ various approximations are used to compute the working set:

- set of pages used during past 100msec of execution time
- R & M bits, R cleared every tick, page fault → find to evict
  R bit = 1? virtual time written to "time of last use"
  R = 0? compare to a time τ.
  All pages in working set? Evict one with greatest age.

DESIGN ISSUES FOR PAGING SYSTEMS

What else do we have to consider in order to get good performance from a paging system?

Local & Global Allocation Policies - a local page replacement algorithm effectively corresponds to allocating a fixed fraction of memory, while global algs dynamically allocate page frames among the runnable processes. ----> these generally work better ---->

If local used & working set grows → thrashing
" & working set shrinks → wastes memory, but

If global used, system continuously decides how many page frames to assign to each process.

We can use PFF (page fault frequency) algorithm to determine whether we should include a process' page allocation if using a global algorithm. Make sure the fault rate is neither too high or too low.

Load control is all about swapping processes out to relieve the load on memory. This is because, all things considered, the system can still thrash, since there's only so much space in main memory. There might be some processes that need more memory, but no others that need less, so the only solution is to temporarily swap some processes back to the disk, without making the # of processes in mem so low that the CPU is idle a lot.

Pagesize is a parameter that can be chosen by the OS, and determining it requires balancing some competing factors (which means there's no optimum).

A randomly chosen text/stack/data segment will not fill an integral # of pages. On average, 1/2 the final page will be empty, & the extra space is wasted. (internal fragmentation)

n segments in memory, page size p bytes, $np/2$ bytes wasted on internal fragmentation (which argues for a small page size)

In general, large pages will cause more wasted space to be in mem, but small page size means programs need many pages & in turn a large page table.

i.e. 32 KB program needs four 8KB pages, but 64 512-byte pages.

In addition to this, transfers to/from disk are a page at a time, and a small page transfer takes almost as long as a large one, and the TLB has space used invaluably by small pages (b/c TLB entries = scarce)

i.e. 1MB prog, 64KB working set → 4KB pages = 16 entries in TLB
↘ 2MB pages = 1 single entry

Due to all of this, OS's might use larger pages for kernel, smaller for processes.

And for one last analysis, let avg process size be s bytes, and page size be p bytes, & assume each page entry requires e bytes.

Approx. # pages per process = s/p, occupying se/p bytes of page table space.

Wasted mem due to internal fragmentation = $p/2$

Total overhead due to PT & internal frag = $se/p + p/2$ — large when page size large

PT size, large when page size is small

From this, we can derive the optimum page size = $p = \sqrt{2se}$

We could seperate instructions & data (address) space with I-space and D-space.

Each address space can be paged individually with their own page tables.

Rather for normal address spaces, they're used to divide the L1 cache.

We can also share pages (to avoid having two copies of same page in mem at a time)

But we can't search through all PTs to see if a page is shared, so we use special data structures instead. Sharing read-only text is much easier than sharing data, because if a process updates a memory word in shared data, a trap is made to the OS, and a copy must be made of the offending page. Unmodified pages need not be copied, and this is called copy-on-write

Mapped files present the idea of a process issuing a system call to map a file onto a portion of its virtual address space. In most implementations, no pages are brought in at the time of mapping, but as pages are touched, they are demand paged in one at a time, using the disk file as the backing store. When the process exits, or explicitly unmaps the file, the modified pages are written back on the file on disk.

Mapped files → an alternate model for I/O.

↳ → instead of reads/writes, file accessed as big char array in mem.

Paging daemon is awakened periodically to inspect mem state.

Too few free frames? Evict w/ page replace alg.

When does the OS have page-related work to do?

1) Process creation time
2) Page fault time
3) Process execution time
4) Process termination time

At [] time, the OS...

1) determines how large program/data will be, creates page table, allocate/initialize space in mem for PT, and in swap area on disk. record info about PT & swap area in process table
2) MMU reset for new process, flush TLB, make new process PT current.
3) read out hardware regs to determine which virt addr caused fault. compute page needed, locate on disk, find avail. page frame to place page (evict old page if needed). Read into page frame, back up PC to point to faulting instr to execute again.
4) release process' PT, pages, & disk space process occupies on disk.

Page Fault Handling, more detailed

1) hardware traps to kernel, saves PC on stack, instr state saved on reg
2) save general reg, volatile info
3) find virtual addr, sees if valid, and finds free page frame
4) frame dirty? disk transfer. once clean, finds disk addr of needed page.
5) loads page in. PT updated. frame marked normal, and PC reset to prev fault instr
6) reload regs, volatile info, execute

INPUT/OUTPUT

The OS must issue commands to devices, catch interrupts, and handle errors. It should also provide an interface bt devices & rest of the system, and this interface should be device independent.

Principles of I/O Hardware

I/O devices can be divided into the categories

- block devices: stores info in fixed-size blocks, each w/ its own address (hard disks, USB sticks, Blu-ray discs)
- character devices: delivers or accepts a string of characters, w/o regard to any block structure, not addressable, no seek operation. (printers, mouses, network interfaces)

Data Rates for common devices:

Keyboard	10 bytes/sec	USB 2.0	60 MB/sec
Mouse	100 bytes/sec	Ethernet	125 MB/sec
56k Modem	7KB/sec	Ultra 5 Bus	640 MB/sec
Camcorder	3.5 MB/sec	SONET Network	5GB/sec

faster ↓

I/O units consist of mechanical & electronic component

Device controller/Adapter is the electric component

Device itself is the mechanical part

A serial bit stream comes off the drive that consists of a preamble, the 4096 bits in a sector, & checksum (or error-correcting code (ECC)).

The preamble is written when disk is formatted, contains sector #, size, & synchronization info.

The controller converts this stream into a block of bytes, & performs necessary error correction. It's assembled bit by bit in a buffer in the controller, and after checksum is verified, it can be copied to MM.

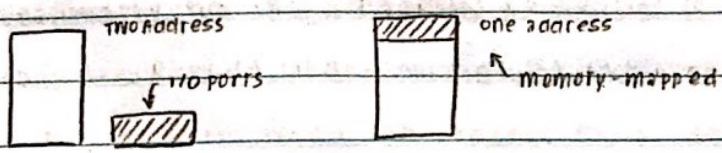
Controllers have a few registers used for communicating w/ the CPU. OS can perform actions by writing to them, & learn about devices states and commands by reading them.

Devices have a data buffer that the OS (programs) can read/write.

But how does the CPU communicate w/ these things?

Approach 1) control register assigned I/O port number, I/O ports form I/O port space, and use special instr to access it, like 'IN REG, PORT'.

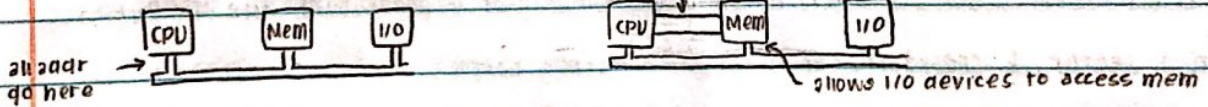
Approach 2) Memory-mapped I/O, where we map all the control registers into a memory space, and each control register is assigned a unique mem addr.



Memory-mapped advantages: no assembly code required, device driver can be written in C. No protection mechanism to keep user processes from performing I/O, b/c we can simply have the OS refrain from putting that portion of the addr space containing the control regs in any user's virtual space. Also, every instr that can ref mem can also ref control regs. (no mem-map means reg must be read into CPU first

But, we have to selectively disable caching b/c caching a control reg would be BAD. In addition to this, if there's only one addr space, all mem modules & I/O devices must examine all mem refs to see what to respond to.

↳ we could have a dual-bus mem architecture where high-speed mem bus is tailored to optimize mem performance w/ no compromizes for slow I/O devices.



But w/ two busses, I/O has no way of seeing memory addresses as they go by on the bus & can't respond to them.

We could:

First send all mem refs to mem. No response? CPU tries other busses.

Put snooping device on mem bus,

Filter addresses in memory controller.

The CPU can request data from an I/O controller one byte at a time, but that's very time consuming, so we use DMA (direct mem. access)

DMA can be used if DMA controller is present, and it has access to the system bus independent of the CPU. It contains a mem addr reg, a byte count reg, and one or more control regs.

control reg contains I/O port to use, direction of transfer (read vs write), transfer unit (byte at a time, or word at a time), & # bytes to transfer per burst

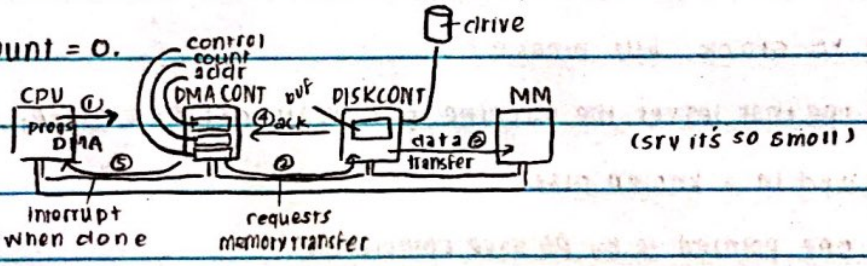
Disk read w/o DMA:

Disk controller reads block bit by bit from drive until entire block is in buffer. Next verifies checksum to ensure no errors, then controller causes interrupt. When OS starts running, reads buffer byte or word at a time w/ loop, and stores it in MM.

W/ DMA:

CPU programs DMA controller so it knows what to transfer where, and issues cmd to disk controller telling it to read data from the disk into its buf & verify checksum.

DMA controller initiates transfer by issuing read req over bus to disk cont, and data is transferred to mem. When write complete, disk cont sends acknowledgement signal to DMA controller & over bus. DMA cont incr mem addr to use and decrements byte count. If byte count > 0, repeat until count = 0.



(srry it's so small)

cycle stealing is where the device controller occasionally steals cycles from the CPU, delaying it slightly in

burst mode (block mode, DMA cont tells device to acq. bus, issue series transfers, and release; can block for a while if long burst though).

fly-by mode (DMA tells device cont to transfer directly to MM).

Disk first reads data into internal buffer before DMA can start.

Why doesn't controller just store bytes in MM as soon as it gets them from disk? (why internal buffer?)

↳ so it can verify checksum before transfer

& once disk transfer has started, bits keep arriving from disk at constant rate (whether controller is ready or not). When block is buffered, bus isn't needed until DMA begins, so DMA transfer to mem isn't time critical.

Not all comps use DMA b/c the main CPU is often much faster than the DMA controller, and can get the job done much faster. It also saves money.

Interrupts Revisited

When an I/O device has finished the work given to it, it causes an interrupt. It sends a signal on the bus line that's detected by the interrupt controller, which asserts an interrupt signal on the bus until it's serviced by the CPU. Interrupt cont puts num on the address line specifying the device & asserts signal to CPU.

Num on addr line is used to index into interrupt table vector to fetch a new PC, which points to the interrupt service procedure.

After it starts running, procedure acknowledges the int. that tells controller it's free to issue another interrupt.

→ Hardware saves info before procedure, like the PC, and possibly other visible and internal registers. But where to? could save to internal registers, but it's slow & can lose data. Usually saved to stack, but whose?

A precise interrupt is one that leaves the machine in a well-defined state.

① The PC will be saved in a known place

② All instr before one pointed to by PC have completed

③ No instr beyond " " " " " has finished

④ The execution state of instr pointed to by PC is known

Imprecise interrupts do not meet these requirements, & often vomit tons of internal state onto the stack. It's super complicated to restart.

We could have a bit that has the CPU log everything to make all interrupts precise, but it would kill performance.

I/O SOFTWARE

The key concept of the design of I/O software is device independence, so that we can write programs that can access any I/O device w/o having to specify. The OS should take care of handling device differences. Another is uniform naming (name of file/device should be string or int), and error handling should occur as close to the hardware as possible. If errors are done transparently at a low level, the upper levels won't even have to know about it happening.