OS - layer of software that provides sw access to hardware resources
- abstraction, resource management
- provide VM interface for apps

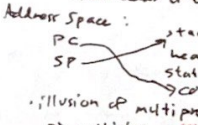
Program VM - supports execution of single program
System VM - execution of entire OS and apps
VM - gives program illusion it owns machine hardware has features you want
PS VM - each process thinks:
- owns all memory/CPU time
- owns all devices
- all devices have same interface
- Fault Isolation
- Protection/Portability

Sys VM - layers of OSes
- OS crash restricted to 1 VM

OS:
- manage sharing of resources, protection, isolation
- provide abstraction, easy to use
- abstraction of physical resources
- common services: networking, storage

4 OS Concepts:
1. Thread
   - single unique execution context, fully describes program state
   - PC, registers, execution flags, stack
2. Address Space
   - programs execute in addr space that is distinct from memory space of physical machine
3. Process
   - instance of executing program
   - consists of addr space, ≥1 thread
4. Dual Mode Operation/Protection
   - only system can access some resources
   - controlling translation from program virt addr to machine physical addr protects user programs from each other, OS, hw

Thread:
- registers hold thread context:
  - SP = top of stack
  - when resident in processor registers, executing
  - PC = addr of executing instr

Address Space:


- illusion of multiprocessing
  - → multiplex
  - → virtual CPU needs to hold thread's PC, SP, registers
  - → switching: save, load
- virtual CPUs share same non-CPU resources: I/O, memory

Process: execution env w restricted rights
- owns memory, fd, file sys context
- protected from each other
  - ⇒ memory protection

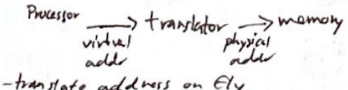
threads: concurrency :: addr space: protection

Dual Mode:
1) Kernel (protected)
2) User

- need state (which mode)
- some ops only allowed in kernel

User → Kernel: set sys mode, save PC, put aside user state
Kernel → user: clear sys mode, restore PC

Base + Bound
- translate addr when program loads
- relocate loader
- basically shift program to execute somewhere in memory

Address Translation


- translate address on fly
- allocate some part of memory to program. Ensure program addr within size of space allocated

Mode Transfer

Syscall
- ps requests system service
- doesn't have addr
- marshall syscall ID and args in register

Interrupt
- external asynchronous event triggers context switch
- e.g. Timer/IO device independent of user ps

Trap/Exception
- internal synchronous event in process triggers context switch

Unprogrammed Control Transfer gets target address from interrupt vector
vector: # ⇒ handler

Process Control Block (PCB)
- represents thread process
- status
- registers, SP
- PID
- execution time
- memory space, translation info

Safe Kernel Mode Transfers
- controlled transfer into kernels (syscall table)
- separate kernel stack
- syscall handler copies user args to kernel space before invoking specific function

Interrupt Control
- interrupts not visible to user process
- no change to process state
- kernel may enable/disable interrupts
- ∃ non-maskable interrupts
  - cannot disable

Safe Interrupts
- handler works regardless of user code state
- atomic transfer of control

Process creates a process
- fork() - create copy of current process w new PID
  - > 0 (return value)
    - running in parent process
    - RV = PID of child
  - = 0
    - running in child
  - < 0
    - error!
    - running in dg PS

State of original process duplicated

exec: change program being run by current ps

wait: wait for ps to finish

signal: sys call to send notification to another process

Shell: job control system
user level equivalent of interrupt
= signal

I/O Design Concepts
- Uniformity
  - file ops, device I/O, interprocess comm thru open, read/write, close
- Open Before Use
  - access control
- Byte oriented
  - addressing in bytes
- Kernel Buffered Reads
  - streaming + block devices look the same
  - read blocks process
- Kernel buffered writes
  - completion of outgoing transfer decoupled from application, allowing it to continue
- explicit close

I/O, storage

High Level I/O	streams
Low Level I/O	handles FILE DESCRIPTOR # (int)
Syscall	registers
File System	descriptors
I/O Driver	commands/data transfers

disks, flash, controllers

File System
- Files live in hierarchical namespace of filenames
- File: data, metadata (info about file)
- Directory: contains files, directories

C File API: streams (sequence of bytes)
- FILE: stdin, stdout, stderr

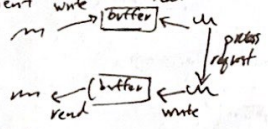
File Descriptor: OS object representing state of file
- user has "handle" on descriptor
- LOW LEVEL I/O

Lower Level Driver
- associated w HW device
- registers/unregisters w kernel
- handler functions for each of file ops

Device Driver
- device-specific code in kernel that interacts w device
- standard internal interface

2 pieces:
- Top half: accessed by sys calls
  - kernel interface w device driver
  - starts I/O to device
- Bottom half: run as interrupt
  - gets input or transfers next block of output.
  - may wake sleeping threads

Communication like File I/O
- connected queues over internet

e.g.


Socket: abstraction of network I/O queue
- embodies one side of communication channel (interprocess comm)

Data transfer like files - w fd

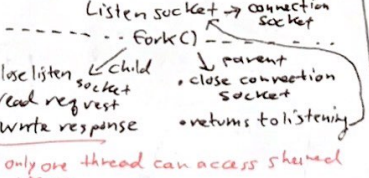
Ports:
0-1023	system ports (↳ SU privileges)
1024-49151	registered ports
49152-65535	dynamic/private

Client: connect() → Server Socket
accept() ← ↓
→ connection → new socket

* new socket for each unique connection

Server has socket that listens for connection

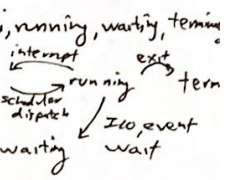
Upon request:



* only one thread can access shared vars

Context Switch - CPU switch from PS A to PS B
- overhead: min practical switching time

Process states:
- new, ready, running, waiting, terminated



PCBs move queues as they change state

Thread State
- shared by all in process:
  - content of memory
  - I/O state (fd, network conn)
- private to each
  - kept in TCB
  - CPU registers (PC)
  - execution stack
    - ↳ parameters, temp vars, return PCs

Shared: Heap, global vars, code

Per-Thread: TCB, stack, stack info, saved registers, stack metadata

Dispatching Loop of OS:
```
Run Thread()
ChooseNextThread()
SaveState of CPU(curTCB)
LoadState of CPU(newTCB)
```

Dispatcher gets control back
- internal events, e.g. blocking on I/O, waiting on signal from other thread, yield()

context switching mostly depends on cache limits and process/thread's hunger for memory

External Events
- interrupts: stop running code to jump to kernel
- timer
  - → ensure that dispatcher can regain control

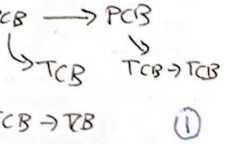
- looks like thread run synchronously but NOT.

- OS keeps track of TCBs in kernel memory

ThreadFork()
- create new thread and put on ready queue
1) sanity check args
2) enter kernel mode and check args
3) allocate new stack + TCB
4) init TCB and put on ready list
   - registers:
     - SP → stack
     - PC ret addr → ThreadRoot()
     - 90, a1 → fcnPtr, fcnArgPtr

ThreadRoot()
- DoStartupHousekeeping()
- UserModeSwitch() // into user mode
- Call fcnPtr(fcnArgPtr) we go
- ThreadFinish

Multithreaded Processes



Process ~~Overhead~~

- Switch HIGH
  - CPU state low
  - Memory/IO HIGH
- Creation HIGH
- Protection
  - CPU ✓
  - Memory/IO ✓
- HIGH sharing overhead

REMEMBER: program can exit before context switch

```
sema game-cap = sema-init(
lock lock = lock_init()
void handle_player()
  sema_down(game cap)
  { lock_acquire
    login to game()
    lock release
  play game
  { lock acquire
    logout
    lock release
  sema_up
```

Threads

- Switch medium
  - CPU state low
- Creation medium
- Protection
  - CPU ✓
  - memory/IO NO
- Sharing low

```
test&set
int (test_and_set (int *valve)
  int result = *valve
  *valve = 1
  return result;

test and test & set
locked = 0
lock ()
  do
    while (locked == 1);
  while
    test&set (&locked)
unlock()
  locked = 0
```

Kernel threads are expensive
- need to go into kernel mode to schedule

Lighter Weight Option: User level Threads
→ User program provides scheduler, thread pkg
→ user threads scheduled non-preemptively
- cannot interrupt each other

Downside: when one thread blocks on I/O, they all do
- Kernel cannot adjust scheduling

Option: scheduler activation
→ Have Kernel inform user level when thread blocks

Threading Models: (user to Kernel)
- 1-1, many-1, 1-many

Multi-core: run multiple threads each on a different core simultaneously

Multiprocessing: multiple CPUs or cores or hyperthreads

Multiprogramming: multiple jobs or processes

Multithreading: multiple threads per process

Why cooperating threads:
- share resources
- speedup
- modularity

Cooperating threads: shared state between multiple threads
- non deterministic, reproducible

ThreadPool - bounded pool of worker threads
- ensures throughput

- threads yield overlapped I/O, computation without having to deconstruct code into non-blocking fragments

Atomic Operation - always runs to completion or not at all
- indivisible
- memory references and assignments of words (load, store) are atomic

Synchronization: using atomic operation to ensure cooperation between threads

Mutual Exclusion: ensuring that only one thread does a particular thing at a time

Critical Section: piece of code that only one thread can execute at once

Lock: prevent someone from doing something, e.g. lock before entering critical section and accessing shared data

starvation: thinking other has shared resource when neither does

solution should protect critical section!

busy-waiting: consuming CPU while waiting

- Can avoid context switching by
  - avoiding internal events
  - preventing external events by disabling interrupts

Better implementation of lock:
- maintain a lock variable and impose mutual exclusion only during ops on that var.

```
int value = FREE
Acquire() {                          disable intr
  disable interrupts;
  if value = BUSY:
    put thread on wait queue
    Go to sleep
    // Enable interrupts?
  else:
    value = BUSY
  enable interrupts
}
Release() {                          enable intr
  disable interrupts;
  if (anyone on wait queue):
    take thread off wait queue;
    place on ready queue;
  else
    value = Free;
  enable interrupts;
}
```

- need to enable interrupts after going to sleep

Alternative: atomic instruction sequence
- read/write value atomically

Test&Set using value

Lock is free, test&set reads 0 and sets value = 1

Busy: 1, Free: 0
- returns 0 when we complete

If lock is busy, test&set reads 1 and sets value = 1 → while loop continues
↳ BUSY WAITING ⇒ priority inversion
- busy-waiting thread has higher priority than thread holding lock

Semaphore - generalized lock
- non-negative int

sema-init (#num) semaphore initial # capacity?

P() - atomic operation that waits for semaphore to be positive, then decrement by 1
> wait() (acquire)

V() - atomic operation that increments semaphore by 1, waking up a waiting P()
> signal()
(release)

- only ops allowed are P, V

2 Uses:
- mutual exclusion
  - initialize val to 1
- Scheduling Constraints
  - init val to 0
  → allow t1 to wait for signal from t2

e.g. ThreadFinish
sema.V()

Thread Join
sema.P()

Finish signals Join

Pintos uses TCB for kernel stack

sema down in critical section may cause deadlock

2 semaphores may lead to deadlock

- use separate semaphore for each constraint

Problem: semaphores are dual purpose
⇒ Locks for mutual exclusion, condition vars for scheduling constraints

Monitor: = lock + ≥0 cond vars for managing concurrent access to shared data

Condition Var: queue of threads waiting for something in a critical section

Key: possible to go to sleep inside critical section by atomically releasing lock when we sleep
- cannot do this w sema

Wait(&lock): atomically release lock and sleep
- reacquire lock later before returning

Signal() - wake up one waiter

Broadcast - wake up all waiter

Must hold lock when doing cond var ops!

Construction of lock: ++ wait --
- Separate lock var, use HW mechanisms to protect modification of var

Hoare Monitor: use if
- signaler gives lock, CPU to waiter; waiter runs immediately
- waiter gives up lock, processor back to signaler when it exits critical section or waits

Mesa Monitor: most
- signaler keeps lock + processor
- waiter put on ready queue
- need to check condition again after wait

↳ use while loop for wait
↳ this is to account for cases in which another thread runs before it and undoes condition thread 1 was waiting on

Readers/Writers Problem
- e.g. database
  - readers never modify
  - writers read + modify

want many readers
1 writer

→ give priority to writers

In this example, hold a lock when changing AW, WW, AR, WR
- these vars and checks on them inherently ensure protection

Goal of Scheduling
- minimize response time
- maximize throughput
- fairness

First Come First Serve
- thread keeps CPU until it yields

Wait time: amount time job wait until it start

Completion time: amount time job finishes till

+ simple
- short jobs stuck behind long ones

exec v doesn't destroy fd; control transfer thru syscalls (creation, int...)

Round Robin
- each process gets unit of CPU time (10-100 ms): = quantum
- after time expires, process preempted + put back in queue
- no process waits > (n-1) q
+ Better for short jobs, fair
- Context-switching adds up for long jobs

Today, time slice = 10-100 ms
context switching = 0.1-1 ms
~1% overhead

Strict Priority Scheduling
- always execute highest priority runnable jobs to completion
- each queue processed RR

Problems:
- starvation of lower priority tasks
- Deadlock: priority inversion when no priority donation

→ FAIRNESS
- gained by hurting avg response time
- could give each queue a fraction of CPU
- could increase priority of jobs that don't get service

Lottery Scheduling:
- give each job some tickets
- at each time, pick a ticket
- CPU time proportional to # tickets given to each job
- to approximate SRTF
  - short running jobs get more, long fewer
- behaves gracefully as load changes

To evaluate a scheduling algorithm:
- Deterministic modeling
  - compute performance of alg on predetermined workload
- Queueing Model
  - mathematical approach for handling stochastic workloads
- Implementation/Simulation
  - build system to run algorithms against

Mix of different apps
- Burst time used to figure out whether app is interactive or high thruput
- sleep a lot/short bursts ⇒ interactive
- less bursts ⇒ not interactive

Minor Best FCFS - Shortest Remaining Job First

SJF: run job w least computation

SRJF: preemptive version - if job arrives w shorter completion time than running job, preempt
↳ get short jobs out of system
- may lead to starvation of longer jobs

Make

```
task_one:      ← target   (rule)
    echo 1     ← recipe
task_two: task_one   ← dependency
    echo 2
```

perror - print error

strace - print syscalls

ltrace - print library calls

open(path, oflags) → fd
read(fd, *buf, nbytes)
write(fd, *buf, nbytes)
lseek(fd, offset, whence)
- move offset of file
whence:
SEEK_SET: offset = offset
-CUR: curr offset + offset
-END: size of file + offset
dup2(new fd, old fd)
- duplicate fd, closes old
signal(signum, handler)
SIG_IGN, SIG_DFL

p-thread_create: creates + immediately starts new thread
thread, *attr, start_routine, *arg

p-thread_join: waits for thread to stop
thread, &retval

pthread_yield - thread_yield

Niceness: -20 < n < 20
↑ take away CPU time

send()

socket(), listen(), accept

bind(socket_fd, addr, size)

SRTF is not realistic
- can predict length of next CPU burst (adaptive)
→ programs have predictable behavior
- Multi Level Feedback Scheduling
- multiple queues, each w own algorithm/CPU time
- jobs go between queues depending on if timeout expires (I/O Bound stays near top)
- serve all from highest priority queue, ↓

Real Time Scheduling
- enforcing predictability of worst case response times
- Hard Real Time (meet all deadlines)
- Soft Real Time (high probability)

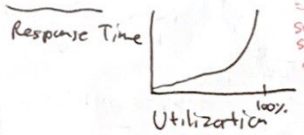
Workload: task (deadline, known computation time)

Earliest Deadline First: schedule active task w closest absolute deadline
response time → ∞, utilization → 100%

Deadlock: Mutual Exclusion
Hold & Wait
No Preemption
Circular Wait

Detect deadlock, then: (Recovery) (may leave system in inconsistent state)
- terminate thread, force resource give up
- preempt resources, don't kill thread
- rewind actions of (deadlocked thread)

Preventing Deadlock:
- infinite resources
- no sharing of resources
- don't allow waiting (retry)
- make all threads request everything they need at beginning
- force all threads to request resources in particular order



Working state of process is defined by data in memory

Memory Multiplexing: protection, controlled overlap
TRANSLATION

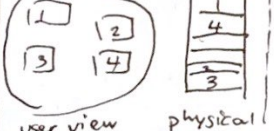
Loading = binding instructions, data to memory addresses can be bound at compile, link/load, execution time

Uniprogramming - no translation/protection
- gets entire physical memory

Base/bound: add base to every address during execution (dynamic address translation)
- error if address > bound
- program gets continous region of memory

Issue: memory gets fragmented (external fragmentation)
no support for sparse address space
hard to do interprocess sharing

Multi Segment Model: Each program gets multiple segments, each has base/bound



external fragmentation

Segment Map: Seg # ⇒ base, limit, valid bit
- base added to offset
- as many chunks of physical memory as entries in segment map

e.g. 4 segments

add base to offset

Seg ID	Base	Limit
0	0x4000	0x0800
1	0x4800	0x1400
2	0xF000	0x1000
3	0x0000	0x3000

- check offset w limit

Virtual Addresses stored in registers

virt addr space has holes
- efficient segmentation for sparse addr space

- stack/heap grow by addressing outside valid range
- need r/w bits in segment table
- segment table stored in CPU on context switch

Problems: must fit varying size chunks into physical memory
- may move ps multiple times to fit
- limited options for swapping to disk
- wasted space (Fragmentation)
- external: between chunks
- internal: within chunks

Solution to Fragmentation:

Pages - fix sized chunks of physical memory
- pages are small
- use vector of bits to handle allocation: 1 → allocated, 0 → free

One Page Table per process
- in physical memory
- contains physical page #, permission bits

Virtual Addr (what data is in phys memory)

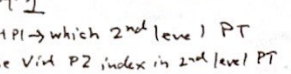
$2^{\text{offset bits}}$ = page size
e.g. 10 offset bits, page size = 1024

Check Virtual Pg # against page table bounds/permission
Virt Pg # ⇒ (Phys. Pg #, permission)
offset - where within page?

Sharing:
- page tables for different processes can point to the same physical pg
- if stuff grows in virtual addr space, allocate new pages in physical memory where room

Fix for Sparse Address Space:
- two level page table

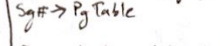
Virt Addr:



PT 1

Virt P1 → which 2nd level PT
Use Virt P2 index in 2nd level PT

Segments + Pages



Seg # → Pg Table
Virt # → Phys Pg

On context switch, contents of top-level segment registers need to be switched
- pointer to top-level table

Sharing: Segments point to same Page table

Page Table Entry Bits
31-12: Physical Pg #
#11-9: Free
L: L=1 ⇒ 4MB page
D: Dirty
A: Accessed (recently)
PCD: Page cache disabled
PWT: Page write transpar (external cache write thru)
U: user accessible
W: writeable
P: present (valid)
Hardware/software translation
HW: fast, inflexible
SW: flexible, every translation must invoke fault

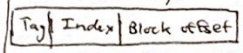
Cache

Average Access Time =
(Hit Rate · Hit Time)
+ (Miss Rate · Miss Time)

Temporal Locality:
- recently accessed

Spatial Locality:
- contiguous blocks

Misses:
- Compulsory: 1st access
- Capacity: cache doesn't fit all blocks
- Conflict: multiple memory locations mapped to same cache location
- Coherence: other process updates memory



On a miss, replace:
- random
- LRU

Write Thru: info written to both block in cache + block in lower level memory
+ read miss cannot result in write
- processor held up on writes

Write Back: written only to block in cache
+ repeated writes not sent to DRAM
processor not held up on writes
- read miss may require WB of dirty data

On TLB miss:
HW: (hw)
if PTE valid, fill TLB
else, page fault

SW: → processor gets TLB fault
if PTE valid, kernel fills TLB
else, internally call pg fault handler

*on context switch, TLB is invalid since virtual addresses map to diff phy addr between processes

On Page Fault, must perform instruction again
- save faulting instruction, partial state
- save state of processor

Precise Exceptions
- all prev instructions completed
- offending and following not as if they haven't started

In TLB, cost of conflict miss time >>> slightly increased cost of hit time (fully associative)

Thrashing - continous conflicts between accesses

CPU → TLB → Cache → Memory
Virt → Phys Addr
Phys Addr → Data

Parallelizing TLB, cache access

Virt Addr	VPN	offset
Phys Addr	PPN / Tag	Index	Byte

- check TLB w VPN
→ returns PPN / Tag
- index into cache w Index
→ returns PPN / Tag
- check if return the same

locks are for threads

Virtual Addresses
TLB
Page Table = cached mapping "in physmem from virt → phys"
Physical Addresses
cache
Physical Memory - address = physical address

What happens when: cannot translate to physical address?

Working set: subset of address space program uses to run

Model of Locality: Zipf
- long tail

⇒ substantial value from tiny cache
⇒ substantial misses from very large cache

Demand Paging
- programs spend 90% of time in 10%, so wasteful to put all of code in memory
Solution: use memory as cache for disk

Transparent level of Indirection:
- flexible placement of user data

- 1 page block size
- fully associative - expensive conflict miss
- find page: TLB, page table traversal
- on miss, go to disk to grab page
- write back

(Handling Page Fault: No physical page for virtual page (valid)
1) reference missing page - "invalid"
2) trap (not in page table)
3) OS grabs page from disk
4) bring page into phys memory
5) reset page table
6) restart instruction

PTE helps w demand paging
- valid: page in memory, PTE points at physical page
- invalid: page not in memory, use info in PTE to find on disk

Page Fault Handling:
- replace old page
- if dirty write contents to disk
- change PTE and cached TLB to "invalid"

"invalid" means not in memory

Effective Access Time:
- Hit Rate · Hit Time + Miss Rate · Miss Time
- Hit Time + Miss Rate · Miss Penalty

Replacement Policies for Demand Paging:
- FIFO - minimum (replace page that would be used for longest)
- random - LRU

LRU performs badly in cyclic reference

Belady's anomaly: adding memory doesn't always help fault rate
e.g. FIFO: A B C D A B E A B C D E

Implementing LRU
- Ideally: timestamp each reference
order by timestamp
- expensive
- Clock Algorithm
- set "use" bit on reference

Dirty Use Wired Valid

Page Fault:
- advance clock hand
- 1: clear and advance
- 0: replace

(3)

N^{th} clock
- give page N chances
- counter per page: # sweeps
- 1: clear use, counter
- 0: increment counter; if = N, replace

large N for better LRU approx.
smaller N for efficiency

For dirty pages: give extra chance before replacing
- Clean: use N = 1
- Dirty: N = 2 (WB on N = 1)

Second Chance List, FIFO / LRU
- memory in 2: Active List (RW), SC List (invalid)
- access pages in active list at full speed
- ow page fault, access SC list

Alternative to SC list: Free list
- filled in background by clock algo
- dirty pages copy back to disk when enter list
- faster for page fault

Page Frame Allocation
Global: process selects replacement from set of all frames
Local: selects from only its frames
Types of Allocation:
Equal: every process gets same amount
Proportional: allocate according to size of PS
Priority: proportional according to priority
- take lower priority process' frames on page fault

PS not enough pages ⇒ high page fault rate
- low CPU utilization
- OS spends lots of time swapping to disk

Thrashing: PS busy swapping pages in + out
Core Mapping: physical page → virtual page
- when you push physical page out of page table to disk, must invalidate all PTEs

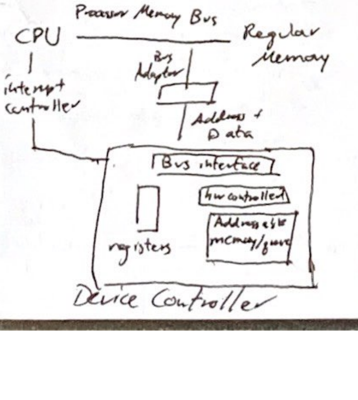
---

I/O
- interact w devices using I/O controllers
- r/w to I/O registers like they're memory

Byte or Block data transfer
Sequential vs Random access
Transfer Mechanism: I/O and DMA
Uniform interface despite different devices
- block devices: disk drives, DVD-ROM
- character devices (byte): keyboards, mouse
- network devices: socket interface

Timing:
- blocking interface (wait)
  - sleep process till data ready
- non-blocking (don't wait)
  - immediately return from r/w request w # bytes successfully transferred
- asynchronous (tell me later)
  - get pointer to user's buffer
  - take data and notify when done

CPU interacts w controller
- controller has registers that can be read or written
- may have memory for request queues



Processor Accesses Registers:
I/O instr: in/out instr
Memory Mapped I/O: load/store
- device registers/memory in physical addr space

Transferring Data To/From Controller:
- Programmed I/O:
  - use instruction
  + easy, simple hardware
  − consumes processor cycles proportional to data size
- Direct Memory Access
  - give controller access to memory bus
  - transfer directly

↳ device driver → disk controller
  - initiate DMA transfer
- DMA controller transfers bytes to buffer
- DMA interrupts CPU to signal completion

Notifying OS: completion, error
- interrupt (high overhead)
- polling: OS checks specific device status register
  + low overhead
  − wastes cycles

Device Driver in kernel
- device specific code that interacts w device hardware

Top half: kernel's interface to device driver
Bottom:
- does actual transfer
- wake sleeping threads on completion

Latency: time to perform an operation
Bandwidth/Throughput: rate at which ops are performed (op/s)
Overhead: time to initiate an operation
Latency(n) = overhead + $\frac{n}{\text{Bandwidth}}$
Bus speed, device transfer bandwidth determine peak bandwidth for I/O

Storage Devices
Magnetic Disk:
1 unit on track (ring) = sector
R/W:
Seek time: position arm over track
rotational latency: wait for sector to rotate under r/w head
transfer time: transfer block of bits
Disk Latency = queueing time + controller time + <r/w 3>

request → SW Queue → HW controller → Media time → result

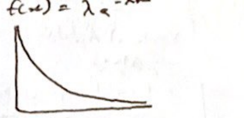
Key to using disk effectively is minimizing seek and rotational delays
∃ error correcting codes, bad sector remapping
→ rarely corrupted → large capacity, cheap
→ slow for random access → better for sequential

Flash Memory
→ capacity at intermediate cost
→ block level random access (both)
→ good for reads, slower for writes
→ erasure requirement in large blocks
→ wear patterns issue
- no moving parts (no seek/rotational delay)

Read is fast, write slower
bc: can only write empty pages in a block
may need to erase a block...
Writes 10x reads, erasure 10x writes
+ low latency, high throughput
+ lightweight, low power, silent
+ read at memory speeds
− [small storage, expensive]
− asymmetric block write performance

Effective BW per Op:
$n / \left(S + \frac{n}{B}\right)$, n = # ops, S = fixed overhead, B = time per op
↳ $\frac{\text{transfer size}}{\text{response time}}$

Model burstiness of arrival using exponential distribution
$f(x) = \lambda e^{-\lambda x}$



Likelihood of an event occurring is independent of how long we've been waiting
Poisson - completely random
Queueing Theory:
Little's Law:
in any stable system, average arrival rate = average departure rate
jobs: λ (jobs/s) × L (s)
arrival time · response time

λ: mean number of arrivals/s
T_{ser}: mean time to serve
C: squared coefficient of variance
μ: service rate = $1/T_{ser}$
u: server utilization = $\lambda/\mu = \lambda \cdot T_{ser}$
T_q: Time spent in queue
L_q: Length of queue = $\lambda \cdot T_q$

Memoryless service time distr (C = 1) exp distr.
$T_q = T_{ser} \cdot u/(1-u)$
General Service Time Distr
$T_q = T_{ser} \cdot \frac{1}{2}(1+C) \cdot u/(1-u)$

---

lock outside while loop that checks condition
- MUTEX: protect access to shared variables

Make scheduler fair: downgrade when thread uses up quanta, upgrade when it yields or gets blocked

---

Page Table
- # entries determined by size of virtual address + size of page
- each entry consists of PPN and valid bits

e.g. virt mem size doubles, # entries doubles
~~Thrashing~~ Paging - exchanging data in memory for data in disk

---

Banker's
- look at remaining resources & check against what each thread needs

---

phys mem lower than aggregate working set sizes ⇒ thrashing

---

memory isn't actually allocated until you begin using it

Memory Protection - use base/bound for each process

In PintOS, all syscalls share a single interrupt

Bankers: unsafe state doesn't necessarily imply deadlock
- evaluate each request
- grant if some ordering of threads is still deadlock free afterwards

guarantees safe state

Remember: killing something hold synchronization primitive will leave system in inconsistent state, e.g. won't "up" semaphore

Given 64 bit address space + 12 bit offset, we have $\frac{2^{64}}{2^{12}} = 2^{52}$ pages.

Multilevel page table will NOT always take less storage space than single level
e.g. all pages in virtual memory allocated

In n-way set associative, num index bits decreases by $\log_2 n$ compared to direct mapped.

1KB Cache w 32B blocks

DM:

tag	index	byte select
22	5	5

2-Way

tag	index	byte select
23	4	5

Full

tag	byte select
27	5

- copy on write: mark pages as read only
- process try to write → fault → OS creates writable copy in new physical frame ⇒ new TLB

#VPNs = # entries

possible for FCFS to have lower turnaround time than RR, e.g. jobs arrive shortest to longest

size of PTE determined by how many physical pages we have

0x0442F - 18 bit addr
first 4 bits are "0001" so we look at index 1 in "0x3000" PT ptr which is 0x4203. Last 2 bits are "11" so it is valid and a directory. Top 7 bits are taken (bc # bits of PPN in PTE) and we 0 out offset to go to address 0x4200. Next 5 bits are 00010 so we look at index 2, which is 0x5601. Last bit = 1 = valid, so we take top 7 bits of 0x5601 and add that to offset 0x02F which gets us 0x562F.
- virtual address loaded into register

I/O
- controlled, managed by OS
- logic to interact w device in I/O controller

Data granularity: block, byte
higher throughput

Access Pattern: Sequential vs Random
How to Transfer: Programmed I/O, DMA

I/O: Uniform interface for devices

Block Devices:
- open(), read(), write(), seek()

Char:
- get(), put()

Threads - shared fds

Network:
- socket interface
  - socket() - create socket - then bind()
- open(), close(), bind(), listen(), select()
  - select: not block on one socket, wait on all sockets
  - read/write to ~~----~~ fd that represents connection
  - create new thread for each connection

Dealing w Timing
1) Blocking - wait: puts to sleep till ready
2) ~~Nothing~~ Non-blocking - return quickly polling
3) Asynchronous - tell me later
   - kernel fills buffer, notifies user when ready. fn called when op complete

I/O devices connected by controller
- connected by bus: data moved in/out of PCI bus, connected to CPU by bridge/memory controller
- all controllers connected to PCI bus

Processor/Device Interaction
- inside device controller are hw controller, memory, bus interface, registers
  - gets command, sends info

2 ways to interact w device
1. I/O instruction
   - particular register to put instruction for I/O device
2. Memory Mapped I/O
   - regions of memory reserved for I/O devices

Transfer To/From Controller (Data)
- Programmed I/O
  - each byte transferred via processor
- Direct Memory Access
  - give controller access to memory bus
  - CPU tells I/O controller, DMA how much data, where to store

Notifying OS
- interrupts: high overhead, good for unexpected events
- polling: OS checks register periodically
  + low overhead
  - may waste many cycles on polling if infrequent I/O

Latency - time to perform operation
Throughput/~~latency~~ bandwidth - rate at which ops are performed
Overhead/StartUp - time to initiate an operation
Latency: $S + \frac{n}{B}$, S = startup cost, B = bandwidth, n = # bytes/bits

Bandwidth: $\frac{n}{(S + \frac{n}{B})}$

Peak BW for I/O: (bottleneck)
- bus speed
- device transfer bandwidth

Magnetic Disk:
- sector (unit of transfer)
- platter, track, cylinder, head, arm

Seek (position head over proper track)
Rotational (wait circle)
Transfer

Disk latency = Queueing time + Controller Time + Seek Time + Rotation Time + Transfer time

PS: owns mem, fd, filesys context

Threads: TCB, stack, stack info, CPU registers, stack metadata

Shared: heap, global vars, code

- minimize seek + rotation delays

SSD - no moving parts
Queueing + Controller + Transfer
Sequential, random reads take same time
Writing ≈ 10x reads
↳ only write on empty pages
Erase ≈ 10x writes

Effective BW per Op: transfersize / response time

Queueing leads to big increases in latency as utilization increases

Service Rate $\mu = 1/T_S$ - time to serve
Arrival Rate $\lambda = 1/T_A$
Utilization $U = \lambda/\mu$, $\lambda < \mu$

Model Burstiness:
exponential distribution
$f(x) = \lambda e^{-\lambda x}$
λ = average arrival rate
interval btwn arrivals
- lots of short arrival intervals, few long gaps

Variable Time T
mean $m = \sum p(T) \cdot T$
σ^2 (variance) $= \sum p(T) \cdot (T-m)^2$
$C = \sigma^2/m^2 = \sum p(T) \cdot T^2 - m^2$
$C = 0 \Rightarrow$ no variance
$C = 1 \Rightarrow$ memoryless, Poisson, completely random

Little's Law -
in stable system:
- average arrival rate = avg departure rate

$N(\text{jobs}) = \lambda \cdot L$ (response time)
jobs in system = arrival time × throughput × response time
arrival rate · how long job is in sys

stable - distribution doesn't change over time

λ: arrival rate
T_{ser}: mean time to serve a customer
C: σ^2/m^2: squared coeff of variance
μ: $1/T_{ser}$: service rate
u: server utilization: $\lambda/\mu = \lambda \cdot T_{ser}$
describe our system $L = T_q + T_{ser}$ (T_{sys})

Compute:
T_q: time spent in queue
$L_q = \lambda T_q$ (length of queue by Little's law)

M/M/1 queue (Little's)
Poisson arrival, service, 1 server
C = 1 (memoryless service time)
$T_q = T_{ser} \cdot \frac{u}{1-u}$ = $\frac{u/\mu}{\mu - \lambda}$

General service time:
M/G/1
$T_q = T_{ser} \cdot \frac{1}{2}(1+C) \cdot \frac{u}{1-u}$

Disk Performance best during sequential reads
- reorder work during bursts
- waste space for speed

Disk Scheduling
- FIFO - may have long seeks
- Shortest Seek Time First
  - pick request closest on disk
  - may lead to starvation

SCAN: closest request in direction of travel
- not fair
- requests in the middle get better treatment

C-SCAN - SCAN in only one direction
- skip requests on "way back"

block: logical transfer unit
sector: physical transfer unit
Fetch ALL blocks corresponding to desired bytes
i.e. if block contains 1 of desired bytes, fetch

File: collection of blocks
- arranged sequentially in logical space

Dir: index mapping names to files

Disk: linear array of sectors
- every sector has integer address (Disk Management)
- controller translates from address ⇒ physical position
- need to track free disk blocks
  - bitmap

open() has lots of overhead: checking, etc

Path ↳ Directory structure → file # (inode) → offset, index structure → block

File: permanent storage
- data, metadata

From user space, get file name (file control block) in directory structure in kernel memory, grab corresponding ~~dir struct~~ ~~----~~ from SS to get proper file control block

File Allocation Table
- which file is allocated which blocks
- each file number is index in FAT
- file number ⇒ linked list of pointers to nth block, indices of disk blocks
- FAT is LL 1-1 w blocks
  - file number is index of root of block list for file
- FAT free list - unused blocks
- FAT stored on boot sector, backup copy on disk
- blocks allocated everywhere bc new blocks fetched from free list
- files usually didn't have blocks in sequential order

Directory: list of entries
Access /my/book/count
- read header for root
- read block for root
- file header for "my"
- block (first) for "my"
  - search for book
- read file header for "book"
  etc

FAT:
- no access rights
- no header in file blocks
- file metadata in directory

Unix Filesystem
- inode format
- file number is index into inode array
- metadata w file

For every dir, read in file header, then data block

Inode
I metadata
12 I data blocks (direct ptr) 48 KB
Indirect block +4 MB
Dbl indirect ptr +4 GB
Tripl indirect ptr +4 TB

Worst case: 4 accesses to get to data for random access
- Bitmap allocation for free list

BSD 4.2: find blocks in contiguous region to expand file
- limit fragmentation
- advantage of bitmap ^

Skip sector positioning to solve rotational delay
Also, prefetching/read ahead

Inode for file stored in same cylinder group as parent directory of file
- limit seeks
10% free, reserved fragmentation

How to Expand File:
- allocate blocks by filling in small holes at start of block group
- allows for contiguous region for later - bigger files can use this

Links - DAG
link entry in one dir to file in another (file #)
Hard - put on disk pointer to inode you refer to (new dir entry)
- maintain ref cnt for each file
Soft Link - contains path + name to file

B-Tree - organizing files
- each entry points to start of range for children
OR
its children are smaller than that entry
- enables quick random access

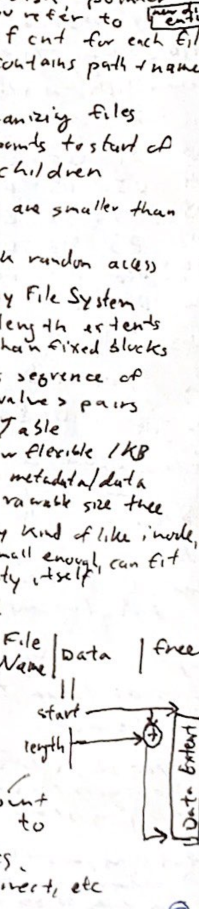
New Technology File System
- variable length extents
  - rather than fixed blocks
- everything is sequence of (attribute: value) pairs

Master File Table
- database w flexible 1KB entries for metadata/data
- extend w variable size tree
- each entry kind of like inode, except if small enough, can fit all in entry itself

e.g. Record

Std. info (metadata)	File Name	Data	free

$\frac{T_{ser} \cdot u}{1-u}$



Some amount of ptrs to extents
- can use indirect, etc

Memory Mapped Files
- file mapped to file
- write directly to memory
- paged w/ OS under hood

Upon mmap()
- create PT entries for (file blocks in memory)
Read file contents from memory

mmap() returns addr in memory of file

Availability: probability that system can accept and process requests
Durability: ability to recover data despite faults
Reliability: give me right answers
- up + working correctly

- disk blocks contain error correcting codes (Reed-Solomon)
- use special RAM - NVRAM for dirty blocks in buffer cache
- replicate to ensure that data survives long term

RAID - redundant arrays of inexpensive disks
- layer of s/w on top of h/w to manage

RAID 1 - disk mirroring
- duplicate disk
- very expensive
- writes more expensive - duplicated
- reads a little better - can read from either

Recovery: hot spare - idle disk already attached

RAID 5 - High I/O Rate Parity
- each disk has blocks, 1 parity block + some data blocks

D0	D1	D2	D3	P0
D4	D5	D6	P1	D7
D8	D9	P2	D10	D11
D12	P3	D13	D14	D15
P4	D16	D17	D18	D19
D20	D21	D22	D23	P5
1	2	3	4	5

P_n = XOR of blocks in row

D2 = D0 ⊕ D1 ⊕ D3 ⊕ P0

Overhead = parity block

$\frac{1}{\#\text{disks} - 1}$ $\frac{1}{n}$, n = # blocks associated w/ parity block

Write: ↑
Read: 0 if there ∄ fail
high if reconstruct

Disambiguate for read: 80% of time, don't need to reconstruct
"Load balancing" (*)

Replication good for reads
- not as good for write, must wait for slowest one

RPC: client passes to RPC, passes to client stub that marshals args then sends to server
- shared fate (local)

Write involves updating multiple blocks: inode, indirect block, bitmap

Guarantee consistency
- threats to reliability:
  - interrupted operation
  - loss of stored data (NVRAM crashes)

Approach 1: careful ordering
- can check if any ops in progress after crash

- allocate data block
  - write
- allocate inode
  - write
- update free bitmap
- update dir w file name → inode num
- update modify time

Revert/reclaim if crash
Recovery:
- scan inode table
- delete or put unlinked files in lost + found
- compare free block bitmap against inode trees
- scan dirs for missing update locations

Approach 2: Copy on Write
- create copy, write on that
- swap new block w old
- update in batch, disk writes can occur in parallel

ZFS/Open ZFS
- swap out old version w new versions (ptrs) in batches

General Soln
Transactions: atomic updates
- all or none of updates
from one consistent state to another

BEGIN
UPDATE
- fail or conflict, roll back
COMMIT

Properties:
Atomicity
Consistency - transactions maintain data integrity ≈ constraints
Isolation - execution of one isolated from all others; no concurrency issues
Durability: commit = persistent effect beyond crash

Reliability w Log Structure
Journaling - log for recovery
- done with op, commit
- commit guarantees that they happen
- recovery reads log to see if everything is done

ASSUMES - updates to sectors are atomic + ordered

1) Write changes to log
2) Commit: Single disk write to make transaction durable
3) Copy changes to disk
4) Reclaim space in log

Recovery: - read log
- do ops
- GC

Log Structured - data stays in log form
Journaling - log used for recovery
- need transactions for writes

- apply to disk, move tail

if Ecommit is there
- redo everything
else
discard log

Centralized System
- major fns on one computer
Distributed - separate computers working together

Why:
- more compute power, storage
- cheaper/easier to build lots of simple computers
- easier to expand
- user has control over some components
- easier collaboration

Promise:
- higher availability
- better durability: store data in multiple locations
- more security: easier to make secure

Reality: ~~always~~ 3 fails all

Goal: transparency
- mask complexity behind simple interface

MAC - 48 bit assigned by card vendor
IP - 32 bit assigned when comp joins nw
Endpoint identified by port

Functionalities:
1) delivery
2) reliability: tolerate pkt loss
3) flow control: avoid overflowing rcv buffer
4) congestion control: avoid overflowing buffer & router along path

Layers provide set of abstractions for nw functionality + technologies

Properties of Layers:
Service - what a layer does
Service Interface: how to access service

Protocol: how peers communicate to implement service

1 - Physical
- move info between 2 sys connected by phys link
- specifies how to send/rcv bits
- coding scheme used to represent bit, voltage level

2 - Data link
- enable end hosts to exchange frames (atomic msgs) on same link
- send frames to other hosts

CAN - group of hosts on same link
- switch finds frames to intended recipients

Protocols:
- partition channel
- token turn
- random access (listen then speak)

Switches

3 - Network Layer
- deliver pkts to specific nw addr
- construct fwding table

WAN - nw that covers broad area, connects multiple link layer nw
- connected by routers

IP - Best effort pkt delivery

4 - Transport Layer
- end to end communication between processes
- reliability, time, flow/congestion control
- demultiplexing of communication between hosts via port #

UDP: multiplexing/demux of processes
TCP: connection set up/tear down, pkt corruption, retransmission, control

7 - Application
- any service provided to end user

Lower 3 everywhere, top 2 only at hosts

Hourglass: all nw can interoperate applications that run IP can use any nw
simultaneous innovations above/below IP

E2E: where to implement functionality
Conservative:
- don't unless can be completely implemented
- don't unless relieve burden from hosts

Moderate:
- only if performance enhancement AND no additional burden on apps that don't need it

server: socket, bind, listen, accept
client: socket, connect
named pipe - unidirectional; UNIX socket - bidirectional
- computers can share IP using network address translation
- selective ACK (TCP) - adds ability to acknowledge individual pkts beyond current con ack
- advertised window in TCP for new data

Log Filesys
- + write - group writes before flushing to disk, improves performance
- read - must read thru log to ensure we have most up to date version but can just cache this

buffer cache holds filesys data in memory
Simple Filesys - disk = 1 big array. files one after another.
- external fragmentation

External Fragmentation - free space is unusable bc each block is so small
Internal - space wasted in allocated memory blocks bc of restriction on min size of allocated blocks
FAT dir Entry: Name - Attr - Index of 1st block - Size
~~Kernel~~ - top half: kernel starts I/O ops
device driver bottom: services interrupts produced by device
FAT32 has 4 GiB file size limit
idempotent - repeated without affect after first iteration.
2PC: V slave replicas and ACTION from master:
Master [vote-request (ACTION)] → slave
slave [vote-abort/commit] → master
Master [global-commit/abort] → slave
slave [ACK] → MASTER

- 1 slave aborts, master sends global-ABORT
- else, GLOBAL-COMMIT
- response from slave ⇒ request has been recognized + committed
- ACK for global commit ⇒ action has been executed/log is now fault tolerant
- log of slave contains proposed transaction and COMMIT/ABORT

Last Byte Written - last byte put in send buffer
Last Byte Sent - last byte sent by sender to rcv
Last Byte Acked - last byte ack rcvd by sender
Last Byte Rcvd - last byte rcvd by rcvr in buffer
Next Byte Expected - last in-sequence byte expected by rcvr
Last Byte Read - last byte read by rcv-ing PS
Advertised Window = Max Rcv Buffer - (Last Byte Rcvd - Last Byte Read)
Send Window = Advertised Window - (Last Byte Sent - Last Byte Acked)
Write Window = Max Send Buffer - (Last Byte Written - Last Byte Acked)
ACK sent out before rcvd by PS

Recursive Query - faster but less scalable
R + W ≥ N + 1: Quorum Consensus
Prepare - timeout: global abort
Commit - timeout: resend msg

2PC - only AC