Scheduling

Scheduling decides which threads are given access to resources moment to moment

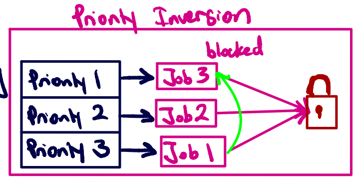
Goals: 1) ↓ Response Time 2) ↑ Throughput (operations/sec), ↓ overhead 3) ↑ Fairness

Waiting Time: time before it got scheduled

Compeletion Time: waiting time + running time

Priority handles differences in importance, watch starving

Priority Inversion: high priority task blocked waiting on low priority thread



Choose	For
FCFS / FIFO	CPU Throughput
SRTF Approx.	Avg Response Time
	I/O Throughput
Linux CFS	Fairness (CPU Time)
Round Robin	Fairness (CPU Wait Time)
EDF	Meeting Deadlines
Priority	Favor important

Policies

1) First Come, First Served (FCFS/FIFO)

Idea: One program scheduled until done

Pro: Least overhead, simple

Con: convoy effect (short processes stuck behind large ones)

2) Round Robin (RR)

Idea: Each process gets small unit of CPU time (quantum)

Pro: with n process, q time quanta, max waiting time (n-1)q

Con: Lots of contex switching, high completion time

3) Shortest Job First (SJF), Shortest Remaining Time First (SRTF)

Idea: Run job with least amount of computation to do

Pro: Optimal!!

Con: Need to be able to see future, know process length

4) Lottery Scheduling

Idea: Give job some # lottery tickets, randomly choose ticket

Pros: On avg, CPU time proportional to # of tickets

Cons: Could choose long jobs, low priority, unfair for less jobs

5) Multiple-Level Feedback Scheduling

Idea: Multiple queues, adjusts queue as process is run; have queues w/ fixed priority scheduling, time slice

Pro: Approximates SRTF

Con: Can counter by requiring I/O and staying in highest

6) Earliest Deadline First (EDF) — Real Time Scheduling

Idea: Tasks w/ deadlines, Computation Times, choose closest deadline

Feasible if n tasks, Computation Time C, deadline D: $\sum_{i=1}^{n}\left(\frac{C_i}{D_i}\right) \le 1$

Pro: For Real Time Scheduling

7) Stride Scheduling, Linux Completely Fair Scheduler (CFS)

Idea: Track virtual CPU time per thread, gets equal share, choose thread w/ least CPU time

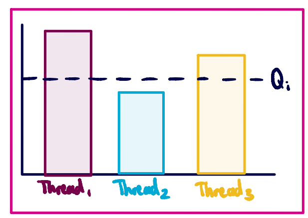
Basic equal share: $Q_i = \text{Target Latency} \cdot \frac{1}{N}$ ← threads

Weighted share: $Q_i = \left(\frac{w_i}{\sum_p w_p}\right) \text{Target Latency}$

↖ allow different rates of execution, ↓ weight ↑ Physical CPU time

Target Latency: period of time where every service gets service

Add min granularity to ensure each process gets to run, min time slice



Deadlocks

Deadlock: cyclic waiting for resources, deadlock ⇒ starvation, starvation ⇏ deadlock

Requirements for Deadlock

1) Mutual Exclusion and bounded resources
 - one thread at a time use resources
2) Hold and wait
 - thread holding resource waits to acquire more
3) No preemption
 - resource released voluntarily
4) Circular Wait
 - set of waiting threads waiting on each other

Deadlock Prevention

1) Provide sufficient resources, VM unlimited
2) Abort requests, acquire atomically
3) Fail if waiting too long, force give up
4) Order resources usage in same order

Deadlock Avoidance

Prevent system from entering unsafe state
↳ Use Banker's Algorithm

Safe space: can prevent by delaying aquisition

Unsafe space: can unavoidably lead to deadlock, with certain aquisition

Deadlocked state: exists a deadlock

Deadlock recovery	Deadlock denial

Bankers Algorithm

- Check if resource request leads to unsafe state
- State max resource needs in advance

Allow thread to continue if available resources − # requested ≥ max

Idea: Allocate resources dynamically
- Evaluate each request & grant if some ordering is deadlock free
- Pretend request granted, run deadlock detection algo

Deadlock Detection Algo [Banker]

add to unfinished
 for each thread unfinished
  if (Request [Max − Alloc] ≤ Avail)
   remove from unfinished
   Avail = Avail + Alloc

Memory

Virtual Memory to multiplex memory, protection, controlled overlap

Pages: small fixed size physical memory chunks

Page Table: one per process, has physical page & permission (R/W/Valid)

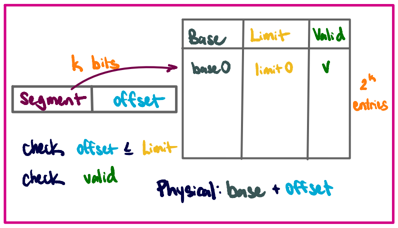
Memory Management Unit (MMU): Translation box converts between virtual & physical address; kernel handles evicting, invalidating, disk

1) Base & Bound / Segment Mapping

Idea: set registers w/ base and limit

Pro: Simple

Con: Internal and External Fragmentation, no sparse address space support or interprocess sharing



2) Inverted Page Table

Idea: use a hash table to map VPN to PPN

Pro: Efficient lookup

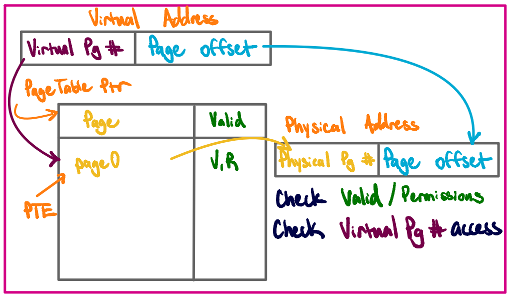
Con: complexity of hash chains, poor cache locality

3) Simple Paging

Idea: Translations in Page Table

Pro: Able to share memory: point to same physical page #, easy to (re)allocate memory

Cons: Page Table too big, Internal fragmentation



4) Multilevel Page Table

Idea: Tree of Page Tables w/ fixed size, save Page Table Ptr (CR3)

Virtual P1 #	Virtual P2 #	Page offset

(10b-10b-12b)

Pros: allocate just needed PTE, easy memory allocation, sharing

Cons: One pointer per page, ≥2 lookups per reference

Page table Entry (PTE): pointer to next level page table or actual page, permission bits

Caching

Translation Lookaside Buffer (TLB): cache for translations

records just end result, recent VPN to PPN, include ProcessID, hardware

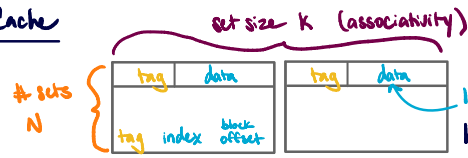
Temporal Locality: Time locality, recently accessed closer

Spatial Locality: Space locality, contiguous blocks

Sources of Cache Misses

1) Compulsory: first access to block
   → clustering, working set tracking
2) Capacity: cache cannot contain all blocks accessed
   → increase cache size
3) Conflict: multiple mem location mapped to same cache location
   → increase cache size, increase associativity
4) Coherence: other process updates memory

Cache



Data = N * L * K

Write Policy / Eviction Policy

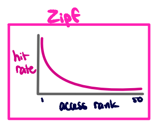
Block: minimum quantum of caching

Index: lookup candidates in cache, identify set

Tag: identify actual copy

Write Through: info written to both block and lower lvl memory

Write Back: info written only to block, write when evict



Zipf distribution: increasing size of cache has diminishing returns

Average Memory Access Time (AMAT)

$$AMAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time}$$

Types of Caches

1) Direct Mapped Cache: single block per set, index

Valid	Tag	Data

2) N-Way Set Associative: N-direct mapped caches

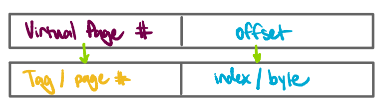
Valid	Tag	Data	Valid	Tag	Data

3) Fully Associative: Every block can hold any line, no index (← TLB typically fully associative)

Tag	Valid	Data

Cache typically physically indexed

Can lookup TLB and cache simultaneously



Demand Paging

Demand Paging: only keep active pages in memory, as cache: fully associative, LRU, write back, 1 pg block

If invalid PTE:

1) MMU traps to OS w/ Page Fault
2) Find & replace page w/ page from disk
3) Reset Page Table & restart instruction

Freelist: keep set of free pages by Clock Algorithm

Working Set: group of pages accessed by process recently

Swapping: some or all of previous process moved to disk to make room

- Can share code segment, setting readonly

Page Replacement Policies

1) FIFO: evict oldest page
   Con: evicts heavily used pages
2) RANDOM: choose random page for replacement
   Con: unpredictable
3) MIN: replaces page not used for longest time, optimal
   Con: don't know future
4) LRU: replace page not used for longest time
   Con: too much overhead
5) Second Chance
   Split into Active and Second Chance List
   Pro: few disk accesses
   Con: increased overhead trapping

Approx LRU:

6) Clock Algorithm Replace an old page, partition into old and young
7) Nth Chance N chances to stay in memory

Allocation of Memory for Processes:

Equal allocation, Proportional allocation, priority allocation

- can set lower and upper bound for memory

Thrashing: busy swapping pages in and out w/ little progress w/o enough pages

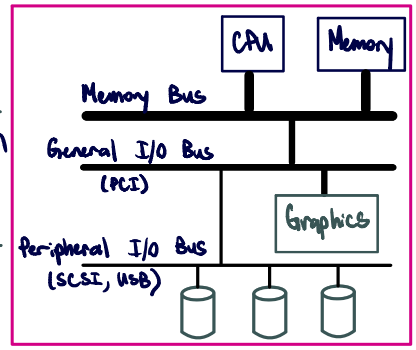
CS 162 Midterm 3 Notes

Jeffrey Shen

I/O

I/O is how the computer communicates w/ the world

Block devices	Character Devices	Network Devices
- Access blocks of data, fs	- single chars at a time	- diff from others, pipes, stream
open(), read(), write(), seek()	get(), put()	sockets, select()
disk drives, DVD-ROM, raw I/O	keyboard, mice, USB	ethernet, wireless, bluetooth



Bus: wires for comm/connecting n devices, protocols for data transfer, one at a time

PCI Express bus: not parallel, fast serial channels, use as many as needed

Ways for process to interact w/ controller: Programmed I/O vs. DMA

3 registers for I/O: 1) status 2) command 3) data

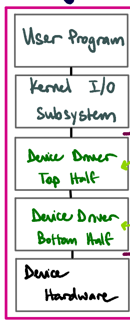
1) **Port mapped I/O:** CPU uses privileged in/out instructions

2) **Memory-mapped I/O:** load/store instructions, in physical address space

Direct Memory Access (DMA): specific device to manage devices

Use hardware interrupts for device I/O, can also poll

1) CPU sets up DMA request, 2) give controller access, 3) DMA interrupts when done



Device Driver: device specific code in the kernel interacts w/ device hardware

Top half: implement std cross device calls (open(), close(), read())

kernel interface to device driver

Bottom half: run as interrupt handler

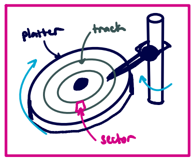
get input, transfer next block of output

Storage Devices

persist memory

Hard Disk Drive (HDD): magnetic disk storage device

block level random access ↑sequential access ↓random access



Request Time: queuing + controller + seek + rotational + transfer

- queuing: software queue in device driver
- controller: hardware controller
- seek: positioning head/arm over track
- rotational: sector rotating
- transfer: transferring block of bits

Response Time/Latency: time to complete task (s)

Throughput/Bandwidth: rate of tasks performed (op/s)

throughput = amount read / time

Startup/Overhead: time to initiate operation (s)

Little's Law: in a stable state, avg arrival = avg depart

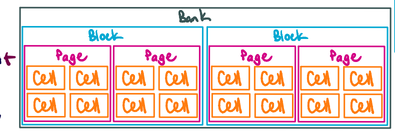
$$N = \lambda \times L$$

- N: jobs, avg len queue
- λ: jobs/s, BW, avg arrival rate
- L: latency, avg time waiting

Solid State Drives (SSD): Flash memory storage device

can erase fixed # times, no hardware move, only transfer (q+c+t)

Operations: 1) read page 2) erase block 3) program page



Throughput approaches μ_{max} bottleneck rate

Memoryless Service Distribution: req arrival time independent

$$T_q = \frac{\rho}{1-\rho} \cdot T_s$$

ρ: utilization (arrival rate / μ)

T_s: mean time to service customer

service rate $\mu = 1/T_{ser}$

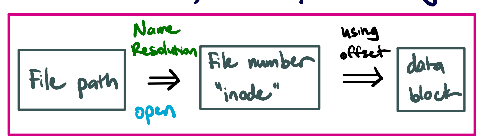
Flash Translation Layer (FTL): Translate logical blocks to Flash layer using indirection and copy-on-write to reduce write amplification & avoid wear out

File Systems

transforms block interface of disks into Files, Directories

Logical Block Addressing (LBA): sector has integer address controller translates addy ⇒ phys pos

- in-memory inode for system-wide open file table
- most files small, most bytes in large files



Disk Scheduling

1) FIFO: fair in requesters, ↓seek time
2) Shortest Seek Time First (SSTF): pick closest req, starvation
3) Elevator Algorithm: closest req in direction of travel
4) Circular Scan (C-SCAN): one direction, skips req going back

File System Designs

1) **File Allocation Table (FAT):** simple, widely used

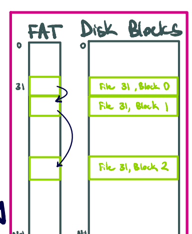
File as collection of disk blocks

FAT is linked list one to one with blocks

File number root of block list for file

Directory is file w/ file_name : file_number mapping

Pro: sequential, no frag, big Con: random, bad locality, internal frag small

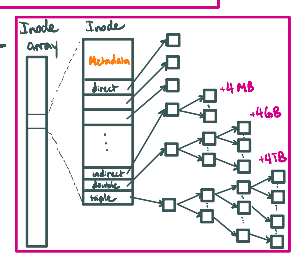


2) **Fast File System:** multi-level tree structure

- File Number index into set of inode arr
- inode corresponds to file w/ metadata
- use bitmap allocation for free

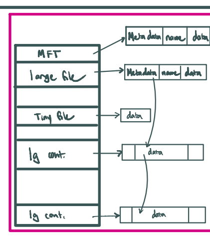
Pro: efficient for small/large files, locality, sequential, random access, no external frag

Con: inefficient for tiny, contiguous, reserve 10% space



3) **Windows New Technology File System (NTFS)**

- Variable size extents w/ 1 KB size entry
- Master File Table: attr:value pairs,
- big files: pointers to other MFT entries
- Supports journaling



4) **Ext 2/3 Disk Layout**

Disk divided into block groups, journaling + FFS

Hard link: mapping from name to file number in dir struct

link never breaks, link()

Soft (Symbolic) link: dir entry mapping name to another name

link could break, symlink()

- can use B-Trees to store name : file_num mapping traversal

Memory Mapped Files: map file into address space, mmap()

Buffer cache: Memory to cache disk blocks / name translations

implemented in OS software, w/ LRU replacement policy

- **Read-Ahead Prefetching:** fetch sequential blocks early
- **Delayed Writes:** writeback, write when full and periodically

Reliability

recovery mechanisms for failures

Availability: probability of system to process req indep of failures
Durability: fault tolerance, ability to recover data
Reliability: ability of system to perform required func
Transactions: atomic sequence r/w, consistent state → consistent state
- if any fail, roll back otherwise commit

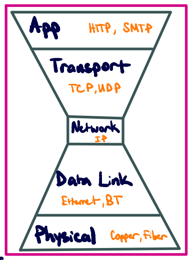
Journaling: log transactions in journal, after logging, apply
Log Structured file system (LSFS): Log is the storage, writes everything sequentially

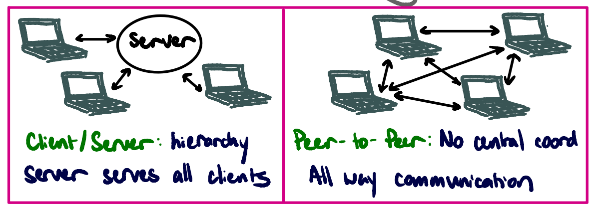
Redundant Arrays of Inexpensive Disks (RAID): Reliable Disk Storage
RAID 1: Disk Mirroring/Shadowing: disk fully duplicated
RAID 5+: High I/O Rate Parity: Data stripped across multiple disks
RAID 6: allow 2 disks in replication stripe to fail
Careful Ordering and Recover: step builds structure, recover scans looking for incomplete actions
Versioning and copy-on-write: version files, creating new structure by linking back to unchanged

Distributed Systems

world becoming large distributed system w/ microprocessors in everything

Scalability: add resources to system to support more work
Transparency: mask complexity behind simple interface ex) location
Protocols: agreement on how to communicate, syntax, semantics
The Internet Allows apps to function on all networks
End-to-End Principle: Implement if can correctly w/o any burden lower layer only for performance enhancement
Hosts: all layers, access data, run applications
Switches: physical/data layer, connects hosts on small network
Routers: physical/data/network layer, route packets cross-network
Internet Protocol (IP): network layer "Best Effort" packet delivery
IP Address: 32 bit integer, destination of IP packet
Subnet: network connecting hosts w/ related IP addresses
Domain Name System (DNS): hierarchical mechanism for naming, name → IP





TCP

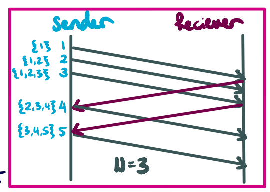
transport connection, ordered reliable delivery w/ congestion control

Transport Layer: E2E comm between processes, demultiplex port
UDP: connectionless service, "best effort"
Sliding Window: send set of n packets in window

Handling Errors

Throughput = N * packet_size / RTT

1) **Go-Back-n (GBN)**: recv only in order, cumulative ACK on timeout/NACK, resend n packets
2) **Selective Repeat (SR)**: selective ACK, resend only lost packet



TCP Properties
- seq nums are byte offsets, GBN, don't drop out of seq packets
- detect congestion using packet loss, AIMD, bad packet checksum
1) Increase rate on ACK 2) Half rate on packet loss

Distributed Decision Making

General's Paradox: impossible to achieve simultaneous acknowledgement over unreliable network

Two-Phase Commit: decide if all processes commit or abort a transaction eventually

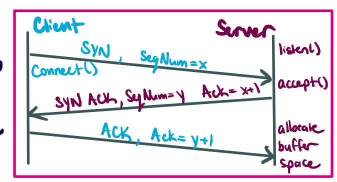
set one coordinator, rest participants

1) Coordinator asks all processes to vote VOTE-REQ
2) Participant vote VOTE-COMMIT/VOTE-ABORT, log
3) If all VOTE-COMMIT, GLOBAL-COMMIT, otherwise GLOBAL-ABORT, log
4) Participant commit or abort on recieve, log

Failure

any participant error, coord votes abort
if all voted commit, wait on coord to recover

3-way handshaking: open conn, congestion control, prevent delayed packets
Round Trip Time (RTT): time for packet sender → reciever → sender

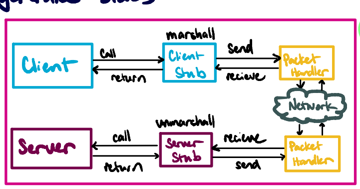


Remote Procedure Call (RPC)

↓ translation complexity

Serialization: expressing object as sequence of bytes
Big/Little Endian: first bit in address most/least sig bit
Marshalling: converting values to canonical form, serializing obj,
Binding: converting user-visible name to network endpoint
dynamic binding allows flexibity w/ servers
Stub generator: compiler that generates stubs

interface def lang → stub code
parameter ↔ req message
result ↔ reply message



Properties of reliable transactions: ACID

1) **Atomicity**: occur in entirety or not at all
2) **Consistency**: one consistent state to another
3) **Isolation**: Concurrent transactions do not interfere, serialized
4) **Durability**: effect persists despite crashes

Distributed Filesystems

Mount remote files on local fs

Virtual Filesystem Switch (VFS): Virtual abstraction of fs, syscalls pass through VFS

VFS object types:
1) superblock: specific mounted fs
2) inode obj: specific file
3) dentry obj: directory entry
4) file obj: open file

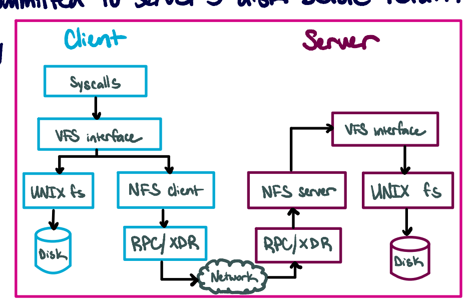
Stateless Protocol: all info to service request is included w/ request HTTP
idempotent operations: repeating operation is same as executing once

Network File System (NFS)

common distributed file system

NFS Protocol: RPC file operations on server, stateless, idempotent
Write-through caching: modified data commited to server's disk before return
Weak Consistency: client only polls periodically
Want sequential ording similar to running on single machine

Pros: simple, portable, efficient
Cons: sometimes inconsistent, doesn't scale



Operating System Overview

Purpose: Special layer of software that provides application software access to hardware resources

1) Illusionist: Provide simple abstractions of physical resources (infinite memory, virtualization)
2) Referee: Manage protection, isolation, and sharing of resources (resource allocation, communication)
3) Glue: Common Services (Storage, Networking, sharing, look and feel)

Four Fundamental OS Concepts

Abstraction	
Processor	→ Threads
Memory	→ Address Space
Disks, SSD	→ Files
Networks	→ Sockets
Machines	→ Processes

1) Threads

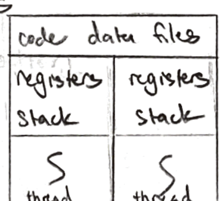
- Single unique execution context
- has own Program Counter, Registers, Execution Flag, stack, Memory State
- When executing and resident on processor: running
- When not loaded in: suspended
- In order to execute multiple processes, multiplex in time, virtual cores
  - Store other threads in Thread Control Block (TCB)

2) Addresses

- Address Space: the set of accessible addresses + state associated w/ them
- OS must protect user programs from one another & protect itself from other programs
  1) Base & Bound: have base register and bound register to check address (base ≤ address < bound)
  2) Address Space Translation (Page Table): Program operates in virtual (pages) translated to memory add

3) Process

- Execution environment with restricted rights
- Protected Address Space w/ 1+ threads
- Running program w/ memory protection
- Processes provide protection, isolation, thread provides concurrency



4) Dual Mode Operation

User Mode: certain operations prohibited. Interrupt handler to direct next steps

Transfers:
1) Syscalls: process requests system service
2) Interrupt: external async event
3) Trap / Exception: internal sync event

Kernel Mode: privileged mode
- Modes provide memory level isolation between processes

Threads & Processes: Programmer POV

- Allow parallel programs to be run
- Multiprocessing: Multiple CPU (cores)
- Multiprogramming: multiple jobs / processes
- Multi threading: multiple threads / processes, same CPU
- Threads have non-determinism: can run in any order, leads to race cond.
- Process fork: copy current process: page table
  1) copy, new process has pid 0, parent pid > 0
- Each process/thread has kernel segment with PCB/TCB, kernel stack. ksp stores kernel stack pointer in order to reduce I/O blocking in kernel

Thread States
- Running - currently in CPU
- Ready - eligible, not running
- Blocked - ineligible to run

	Process	Thread
Creation	fork()	pthread_create()
Page Table	Distinct	Same
Registers, ip	Distinct	Distinct
Stack	Separate & inaccessible	Separate but accessible
Heap, static var	Separate	Shared
File descriptors	Separate	Shared
Synchronization	wait(), waitpid()	pthread_join(), semaphores, locks
Overhead	Higher	Lower
Protection	Higher	Lower

parallel ⇒ concurrent

concurrency ⇏ parallel

File I/O, Devices

Everything is a file: open, read, write, close

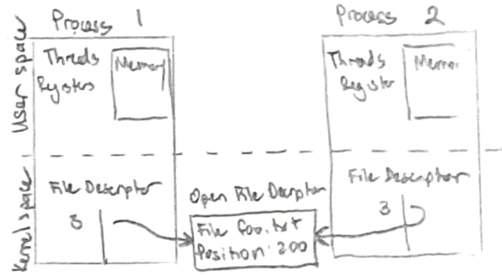
High level FILE: buffered, has fd, buffer, lock

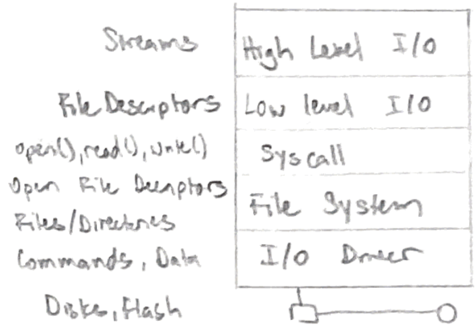
Low level file: returns fd (not buffered)

Drivers: device specific code in kernel that interacts directly w/ device hardware

Top half: accessed through syscalls, initiate I/O, put waiting thread to sleep

Bottom half: runs as interrupt routine, wakes up sleeping threads when I/O complete





Can have same file diff descriptors

IPC, Pipes, Sockets

Interprocess Communication (IPC)

- communication between protected environments (processes)

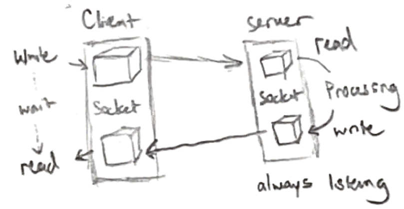
Pipes: act as single queue between processes

- write only on one side, read only on other

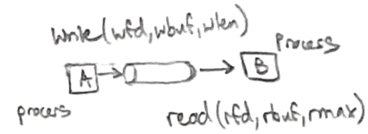
Sockets: allow two queues, communication between

- communication between multiple processes on different machines, socket / bind / connect / listen

Sockets



Pipes



Synchronization

- Many different solutions to fixing synchronization issues, want least busy waiting
- Atomic Operation: operation that always runs to completion or not at all
- Mutual Exclusion: ensuring only one thread does particular thing at a time, excludes the other
- Critical Section: piece of code only one thread can execute at once
- Locks: synchronization mechanism for enforcing mutual exclusion on critical sections to construct atomic operations
- Hardware atomicity primitives disabling interrupts test & set, swap, compare & swap, load-linked & store conditional
- Separate lock variable, use hardware mechanisms to protect modifications of that var

Semaphores: synchronization primitive

- Down() or P(): waits for semaphore to become positive then decrements by 1, like (wait)
- Up() or V(): atomic operation increments by 1, wake up waiting P (signal)

Monitors: a lock and zero or more condition variables managing concurrent access to shared data

- locks for mutual exclusion and condition var scheduling constraint
- condition variable: queue of threads waiting for inside critical section var

Hoare Monitor

```
if (isEmpty(&queue))
    cond_wait(buf_CV, &buf_lock)
```

Mesa Monitor

```
while (isEmpty(&queue))
    cond_wait(&buf_CV, buf_lock)
```

- Wait(&lock): Atomically release lock and go to sleep
- Signal(): wake up one waiter
- Broadcast(): Wake up all waiters

Reader's/Writers Problems

```
while (test & set(guard))
```

Lock: prevent others from changing critical section

acquire(&lock): wait till lock is free, then grab, run critical section

release(&lock): unlock, wake up any one waiting

Implementation:

```
Acquire():
  disable interrupts
    if (value == BUSY)
      put thread wait queue
      go to sleep
  else: value = BUSY
  enable interrupts

Release()
  disable interrupts
    if anyone on wait queue
      take thread off queue
      Place on ready queue
  else: value == 0
  enable interrupts
```

futex

Kernelspace wait queue attached to user space atomic integer

Faster, no syscalls, FUTEX_WAIT, FUTEX_WAKE

write(fd, char *, strlen() + 1)
word_count * wc
ref_count?

char buffer[size]

intr_disable()

thread_current(): get current thread

strcpy(dest, src, len): copy from src to dest strlen() + 1

ex) list_entry(e, word_count_t, elem) list_init(&cv->waiters)

[illegible]