int *p p is pointer to int (has addr to int)

p=&y value of p is addr of y (p is pointer to y)

z=*p val at addr p is assigned to z

Memory

- malloc((strlen +1) · sizeof(char)) → ptr to allocated mem
- calloc(# blocks, size per block) → ptr, set to 0
- realloc(ptr, new size)
- free(ptr)

- Int strcmp(const char *str1, const char *str2)
  - return <0 if str1 < str2
  - return >0 if str1 > str2
  - return = 0 if str1 = str2

Dereference

*(a+i) ⟺ a[i]

foo->bar ⟺ (*foo).bar

Memory

0xFFF	Stack - local vars
	↓
	↑
	Heap - malloc
	static - global, static string 1st
0x000	text - code, struct, instructions

char * strcpy(char * dest, char * src)
- copy src into dest
- return ptr to dest

memcpy(void * dest, const void * src, size_t num)

fread(void *ptr, size_t size, size_t count, FILE * stream)

fopen(const char * filename, const char * mode)

Process and Threads Summary

	Process	Thread
creation	fork()	pthread_create()
data/mem	Ind. addr space	shares addr space w/ other thread in same process. Ind. registers and stack
communication	pipes, files, socket	read data of another thread
synchronization	wait, waitpid()	pthread_join(), sema, lock
FDs	separate	shared
heap/static var	separate	shared
stack	separate & inaccessible	separate but accessible
reg	distinct	distinct
page table	distinct	same
	higher overhead	lower overhead

Process: one or more threads w/ private address space
- fork() → pid of child to parent
  → 0 to child
- dup everything same virtual addr, diff physical addr.
- wait(int * status): wait for one child to finish, return id of process that just finished
- waitpid(pid of child, &status, 0): wait for specific child
  -1: any child
- exec(): replaces current process (don't finish curr process) takes in name of file to be executed
- exit(): kill process

OS Basics

Referee, Illusionist, Glue

Signals

- Asynchronous notification sent to a process
- OS interrupt process to deliver the signal
- signal(Int signal, func) to change handler ↗ can be user defined func that takes in an int

SIG_IGN: ignore signal, SIG_DFL: default action

Files

File Descriptor: int to reference a file

FD	File
0	stdin
1	stdout
2	stderr

system calls
- open(const char * path, int flags)
  - ↳ open file and return file descriptor (next available int)
  - offset = 0
  - int close() return 0 success -1 fail
- read(int fileds, void *buf, size_t nbytes)
  - read nbytes of data from offset of file into buffer
  - return # bytes read <size_t>
  - offset += nbytes
- write(int filedes, void * buf, size_t nbytes)
  - write nbyte data from buf into file at offset
  - return # byte written <size_t>
  - offset += nbytes
- lseek(int fildes, off_t offset, int whence)
  - move offset of file, return new offset
  1. SEEK_SET: offset set to passed in offset
  2. SEEK_CUR: offset set to offset + curr_offset
  3. SEEK_END: offset set to size of file + offset

open flags

O_RDONLY	R
O_WRONLY	W
O_RDWR	R/W

- dup(int fd): copies fd using lowest unused fd

  >>> dup(1)

  0 → stdin
  1 → stdout
  2 → stderr
  3

- dup2(int oldfd, int newfd) uses newfd instead of lowest unused fd

  >>> dup2(3, 1)

  0 → stdin
  1 → stdout
  2 → stderr
  3 → a.txt

File: abstraction of fd, higher level, fopen ... clib funcs, not sys calls

Threads: smallest unit of sequential instructions that can be scheduled by OS
- int pthread_create(pthread_t *thread, <attributes>, void * (start_routine), void *arg)
  - return 0 if successful
  - create and starts a child thread
- Int pthread_join(pthread_t thread, void ** retval)
  - wait for thread to terminate
  - return 0 on success
- Int pthread_yield(void): go back to ready queue let another thread run share address space
  - no guarantee that a new thread will run
  - return 0 on success
- pthread_exit() → kill thread
  - If parent thread exits, child thread also exits
  - new thread → allocate stack everything else same
  - kernel schedule threads

Synchronization

Thread safe? $x=1$ yes int $y=x$ yes x++ no
↳ atomic or not

Sema
- v() up "release"
- p() down "acquire"

Lock
- 0 means free
- 1 means taken

• Order you acquire lock can cause deadlock
↳ acquire in fixed order

wait() - release lock, block self, reacquire when signaled
- mesa → while( ) wait
- hoare → If ( ) wait

- signal() - wake up one waiter
- broadcast - wake up all waiters

- pthread_mutex_lock(&mutex) (&f→lock)
- pthread_mutex_unlock (&mutex)
- pthread_yield()

Mode Transfer

1. syscall - program request system service (exit)
   - like func call but outside process
2. Interrupt - external async event triggers context switch (Timer, I/O)
3. Trap or exception - Internal async event triggers context switch (seg fault, divide by 0)

Process Control Block

- status (Ready, Running, Blocked) (waiting)
- registers, sp, pc
- pid, user, executable, priority
- execution time
- mem space, translation tables
- kernel scheduler maintains PCBs

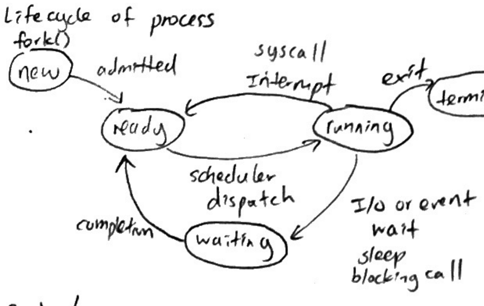
Switch from one virtual CPU to other (context switch)
save pc, sp, reg in current PCB and load PC, SP, reg from new PCB

User mode

↑ exec syscall ↓ ↑ rtn Interrupt ↓ exception ↓ exit ↓

kernel mode - Interrupt handler
- scheduling threads
- create threads

Life cycle of process



Banker's

Max - current = Needed
Available = Total - sum of current

Thread State

shared between threads
- memory (global var, heap)
- I/O state (fd, network conn)

Private between threads
- TCB
  - CPU registers (PC)
  - executable stack
    - params, temp var
    - Return PCs

Sockets

- abstraction of network I/O queue
- read/write against descriptors to transfer data
- over any kind of network (local, internet)

Server

1. create socket <int fd = socket (server → family, server → socktype, server → protocol)
2. bind socket <bind(fd, &addr, sizeof(addr))>
3. listen (fd, 0) tell socket to accept incoming requests
4. accept requests <int c = accept(fd)>
   return new socket for new connection

Client

1. create socket
2. bind socket
3. connect() socket to make connections

Scheduling

FIFO - gets scheduled as they come (good throughput)
- convoy effect: short processes stuck behind long ones
- has best case but also worst case
  ↳ come in increasing order

SRTF - preemptively schedule process w/ shortest remaining time first
- optimal but impossible (good I/O throughput, avg response time)

Strict priority - always run highest priority
- starve lower priority threads, deadlock

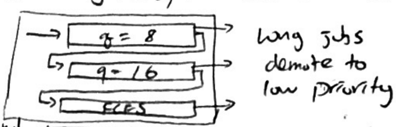
Round Robin - run each process for fixed unit of CPU Time (quanta) (Fair)
- no process wait more than $(n-1)q$ time units
- quanta ↑ → FIFO, q ↓ → hyperthread, q >> context switch or overhead too large
- b/n wait and best case FIFO

Lottery - give each job some # of tickets, pick random ticket to run
- assign ticket based on how long job should take

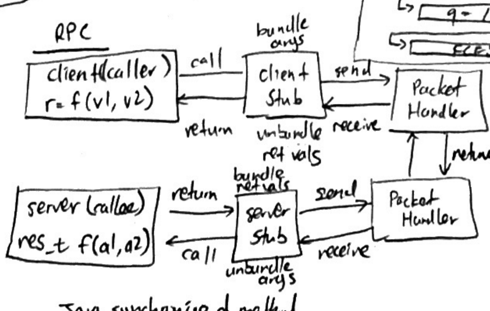
Linux CFS - each process gets an equal share of CPU

EDF - run process w/ earliest deadline first
- won't work if too many tasks, schedule exists if $\sum_{i=1}^{n} \frac{C_i}{D_i} \le 1$ (run time, deadline)

Multi-Level Feedback



RPC



uint32_t htonl (uint32_t hostlong)
- converts from host endianess to network endianess

uint32_t ntohl (uint32_t netlong)
- convert from network endianess to host endianess (used by server)

Deadlock Req

1. Mutual Exclusion
2. Hold and wait
3. No preemption
4. Circular wait

C++ - Lock guards

- mutex released when 'lock' out of scope

Python - with keyword

with lock # Auto call acquire()
release called however we leave block

Java synchronized method

- lock acquired on entry and released on exit
- properly released when exception occurs
  - wait, wait(long timeout)
  - notify(), notifyAll()

Golang

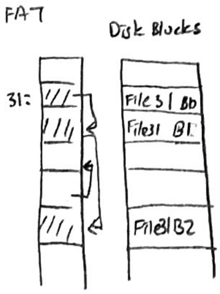
- defer keyword
- go routines - lightweight, user-level threads
  - channels - bounded buffers like pipes but in userspace

FAT (File Allocation Table)

- File is a list of disk blocks
- File # is index of file's block list root
- Follow block list to seek in file
- Grow file by appending to list
- Good Seq, Bad Random b/c LL traversal
- no external fragmentation, small file → internal fragmentation
- Big file (FAT32 max 4GB) 2^{32}
- bad locality for files and metadata

Directory – file w/ dir entry
entry: name – attributes –
index of 1st block – size

Pro: no external frag, can grow file size, hierarchy of dir
cons: no preallocation, not contiguous allocation (slow, read/write)



RAM Disk

1. disk field boot sector
   - info to boot computer
2. Super block – fixed size, metadata & filesystem
3. FAT – each entry has datablock index
4. Data section – small 4KiB blocks

N>0, N is index of next block
N= max array size, end of file
N= -1 free block

ACID

Atomicity: happen or doesn't
Consistency: valid state
Isolation: every transaction separate
Durability: Tx effects persist despite crashes

- data left in intermediate state
- data left in invalid state
  - concurrent TX interface
  - crashes destroy committed TXs

RAID 1: duplicate data
RAID 5: use data and parity bits
- data stripe across disks

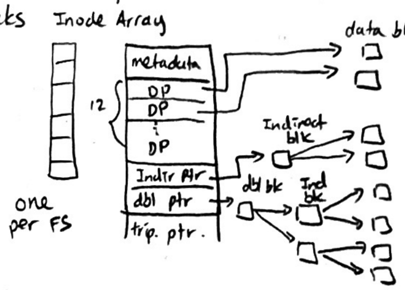
D1	D2	D3	P1
D4	D5	P2	D6
D7	P3	D8	D9
P4	D10	D11	D12

P1 = D1 ⊕ D2 ⊕ D3

Journal: keep track of Tx,
if committed → has to be done

FFS, Unix FS

- most files small, disk usage mostly used up by few large file
- bitmap allocation   Inode: DS that has metadata of file/dir
  metadata: owner, filesize, mod timeline, file mode, ref cnt
- 12 direct block ptr, 1 indirect, 1 doubly indirect, 1 triply indirect
- direct pointer to one data block on disk.



External frag: free gaps between allocated chunks
Internal frag: don't need all mem w/n allocated chunk

block
Tag | Index | Offset
↑ identify actual block
lookup candidates in cache

Queuing Theory

HDD: platter, disk, sector
related to bandwidth

Latency = $\underbrace{\text{Queuing}}_{Tq}$ + controller + seek + rotational + transfer (T_{ser})
SSD: no moving parts, really hard to erase block
read latency: Queuing + controller + transfer
write latency: Queuing + controller (find free block) + transfer
Stable system: Avg arr. rate = avg dep rate

arr. rate → Queue → ser rate → server
$\lambda = \frac{1}{T_A}$ (Req/sec) $\mu = \frac{1}{T_S}$ (ops/sec)

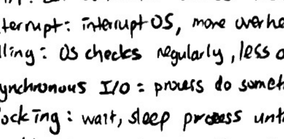
Throughput: rate at which ops are performed over time

Eff BW = $\frac{n}{s + \frac{n}{B}} = \frac{B}{1 + \frac{sB}{n}}$
ops, setup time, time/op

Parameters

λ: arr rate (job/sec)
Tser: time to service job
C: squared coeff of var
μ: service rate $\mu = \frac{1}{T_{ser}}$ (job/sec)
U: $\frac{\lambda}{\mu} = \lambda \cdot T_{ser}$ [0,1] >1 not stable

computed params
$L_q = \lambda \cdot T_q$ len(queue)
T_q = Time spent in queue
memoryless dist (C=1) M/M/1
$T_q = T_{ser} \cdot \frac{u}{(1-u)}$
General dist (no restrictions) M/G/1
$T_q = T_{ser} \cdot \frac{1}{2}(1+C) \cdot \frac{u}{(1-u)}$

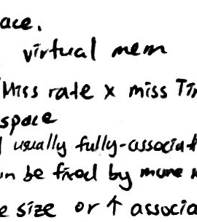


Resp time = Queue + I/O
device service time

Layering

1. Bottom, Physical Layer
2. Datalink: send to MAC Addr LAN
3. Network: global connect, unique network addr, WAN } everywhere
4. Transport: TCP Transfer, maintaining reliability } only on hosts
5. Application

Quorum Concensus: R+W > N+1
General's Paradox: no way to guarantee 2 entities do something simultaneously over an unreliable network
Flow control: fast sender doesn't overwhelm slow receiver
Byzantine's General's Problem: must have N >= 3F + 1 to solve
TCP: Guarantee in order delivery, start w/ 3-way handshake
Receiver: last byte rec – last byte read ≤ max receiver buffer
Adv window = max receiver buffer – (last receive – last read)
Sender: Last sent – last Acked ≤ Advanced window

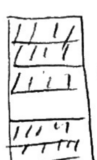


I/O: OS receive and send data to device
Controller: perform actual I/O operation, communicate w/ HW
Programmed I/O: transfer data via processor load/store
DMA: let controller access mem bus w/o CPU
Interrupt: interrupt OS, more overhead, good for infreq events
polling: OS checks regularly, less overhead, good for freq events
Asynchronous I/O: process do something else, get notified when it's done
Blocking: wait, sleep process until completely done.
non-blocking: return quickly w/ # bytes transferred, could be 0

Inverted Page Table: fixed size hash table where entry is physical page of ram and value is VPN, pid, and metadata. Either scan table or hashing VPN to reduce search space.
Paged segmentation: Table size ~ # of pages in virtual mem
Average Access Time = (Hit rate × Hit time) + (Miss rate × miss Time)
working sets: varying sized subsets of the addr space
TLB: record recent VPN ⇒ PPN translation (small usually fully-associative)
cache misses: compulsory (first time), Capacity (can be fixed by more mem), conflict (mapped to the same loc) either ↑ cache size or ↑ associativity
coherence (invalidation): other process updates mem
Programs spend 90% of time in 10% of their code
MIN Page Repl: replace page that won't be used for the longest time, optimal impossible
Belady's anomaly: adding mem → more page faults; same pattern

Simple File sys

- Disk is big array
- TOC → data
- Files stored contiguously (Fast read/write)
- Base and bounds



pro: fast R/W
con: external frag
- can't easily extend file
- no hierarchy of dir

soft link: dir entry contains the path and name of the file
hard link: sets another dir entry to contain the file number for the file

1 bit 0/1	0	1	2	3	4	5	6	7	8	9	10	11	12
1 byte = 8 bits	1	2	4	8	16	32	64	128	256	512	1024	2048	4096

1 word = 32 bits

1 KB = 2^{10} byte (10^3) 1 MB = 2^{20} (10^6) 1 GB = 2^{30} (10^9) TB, PB (10^{12}, 10^{15})

Dec	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

100 ms = 0.1 s

Synchronization

Locks - one thread hold at a time, only holder can release
p_thread_mutex_lock(&lock)
pthread_mutex_unlock(&lock)

Dead lock : 2 threads wait on each other
- avoid by acquiring locks in consistent order
- mutual exclusion: one thread at a time
- hold & wait : thread holding at least one resource is waiting to acquire resources held by other threads
- No preemption
- circular wait

Monitor - one lock and 0+ cond vars, queue of threads waiting for cond to be true
- need to hold lock to do anything

cond_wait(CV, & lock) = put thread to sleep, release lock, put thread on wait queue

cond_signal(condx): remove one thread from wait queue and put on ready state

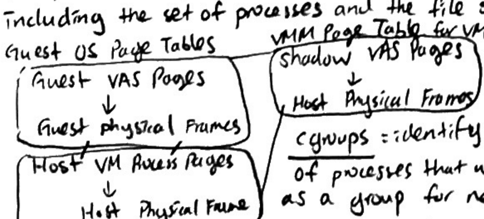
cond_broadcast( ) : remove all thread and put on ready

Hoare: wake blocked thread and it runs immediately waiter gives up lock, CPU back to the signaler when it exits critical section or if it waits again
if (...) wait

Mesa: (most Real OS)
- signaler keeps lock and CPU
- waiter placed on ready queue w/ no special priority
- practically need to check cond again after wait
- while (...) wait

VM/containers

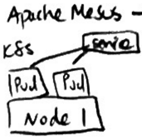
- VM provides SW the illusion of having its dedicated HW; container provides SW only its own dedicated OS, Including the set of processes and the file system.



cgroups : identify collections of processes that will be treated as a group for resource alloc

- containers define a collection of lib and exec that should be a group
- Container shares the host kernel but own sys binaries and lib

Apache Mesos → abstract data center resources to framework



Throughput = W * packet_size/RTT
window_size

Clk Alg

- approx LRU w/ better performance
- clock hand points to cache entry
- each entry has use bit
- set use_bit when accessed
- cache miss → advance clock hand, if use bit 1, set to 0 and move hand, else evict entry, bring in pg and set to 1.

Availability: chance sys can accept and process req

Durability: ability to recover data despite faults

Reliability: ability to perform its required func under some cond for some time.

Performance

Resp Time / latency : Time to perform op

Bandwidth / Throughput : rate at which ops are performed

startup / overhead = time to initiate op.

most I/O ops are roughly linear in n bytes.

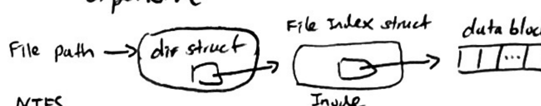
Storage devices

magnetic disks - large capacity, low cost
- slow for random accesses, better for seq
- seek time : position head over correct track
- rotational latency : wait for desired sector to rotate under head
- key for efficiency is minimizing seek and rotational delays

Flash mem - capacity and cost have been getting better

SSD : no moving parts → no seek or rotational delay
low power, lightweight
Latency : Queuing + controller + xfer
- complex write, write 10x read, erasure 10x write
- low latency, high throughput
- expensive

one blk = multiple sectors
512 sectors, 4k block



NTFS
- Master File Table, max 1KB size for each table entry containing metadata plus file's data directly, list of extents for file's data, or pointer to other MFT entries w/ more extent lists.
- Extents store the starting block and # of subsequent blocks (contiguous)
FFS : list of fixed size blocks, not necessarily contiguous

Consistent Hashing

- Each (key, value) stored at node w/ smallest ID larger than hash(key)

AFS - Full files are buffered locally upon open. Buffers are write back and only flushed on close. Last write wins.

NFS - stateless RPC protocol. Buffers are write behind every second. Few strong consistency guarantees on parallel write. NFS is eventually consistent

Ext4 - open quickly close write : slow unless buffer read : requires syscall and maybe I/O

DHT 256 B key 128 MiB val machine 8 GiB/s RTT btw dir and data 2ms
RTT btw client and dir/data is 64 ms.

1 GET Req (recursive query) 64 + 2ms + $2^{27}/2^{33}$ = 82ms
the server will do all the work

2048 GET Req (recursive) 66 ms + $(2^{11} \times 2^{27})/2^{33} = 2^5$ seconds

1 GET Req (iterative) 128 ms + $2^{27}/2^{33}$ = 143 ms

2048 GET Req (iterative) 64 ms + $(2^{11} \times 2^{8})/2^{33}$ to resolve keys.
2048 256

64 ms + $2^{27}/2^{33}$ = 0.0156 to transfer data (parallel)

acquire
while (test_and_set(&lock->state, 1) != 0)
wait (&lock -> state, 1)

release
check cond
wake (&lock->state, 1)

struct list → list_init (&list)
struct lock → lock_init (&lock)

pagedir_get_pg(t->pagedir, addr)

cond_wait in wait
cond_signal in wake.