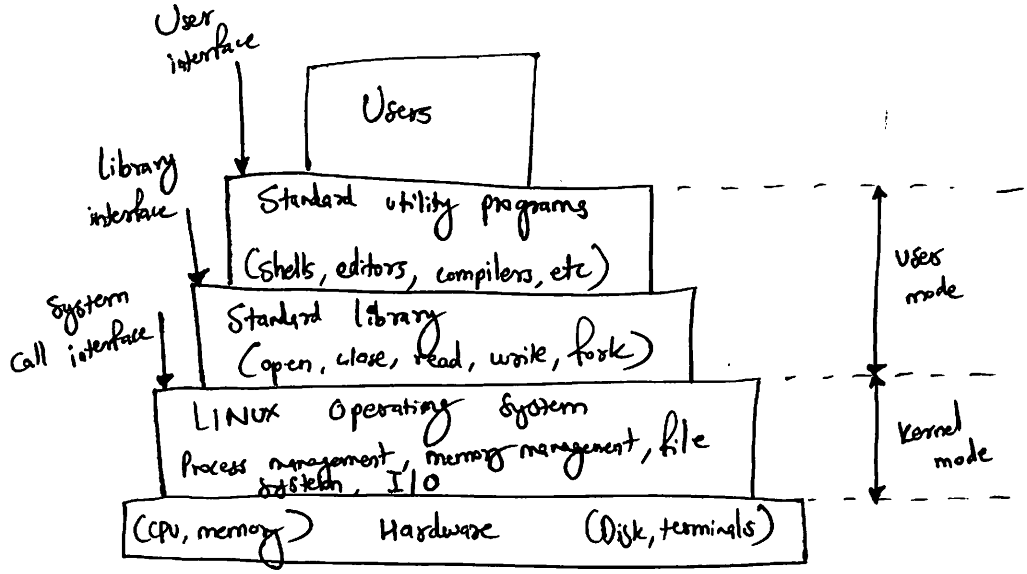
Linux — • principle of least surprise (eg.— ls A* rm A*)

• power — basic elements — ∞ variety of combinations

Layers in linux system


User interface
Library interface
System call interface
Users
Standard utility programs
(shells, editors, compilers, etc)
Standard library
(open, close, read, write, fork)
LINUX Operating System
Process management, memory management, file system, I/O
(CPU, memory) Hardware (Disk, terminals)
User mode
Kernel mode

Hardware — • control hardware

• system call interface to all programs

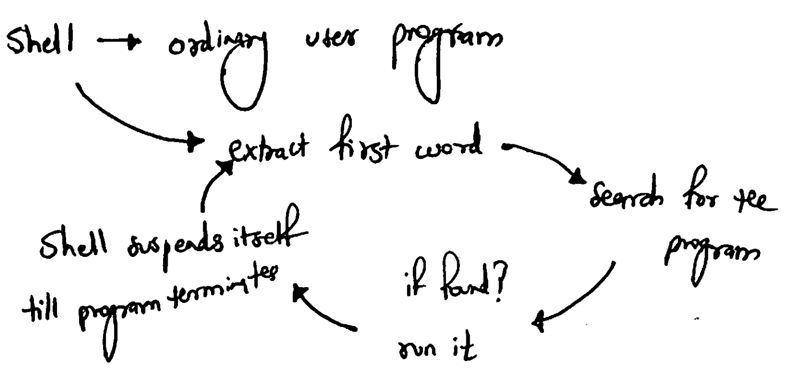
How system calls are made() —

1. Putting the arguments in registers (or sometimes stack)

2. Issuing trap instructions to switch from user mode to kernel mode

3. To issue trap instr. → library is provided → 1 ~~produce~~ procedure / system call

Synchronous interrupt as they are produced by the CPU while executing instructions (done after terminating the execution of the instruction)


Shell → ordinary user program
extract first word
Search for the program
if found? run it
Shell suspends itself till program terminates

Shell- when it starts up - acces to

	File descriptors
standard input	0
standard output	1
standard error	2

redirection:- stdin <
stdout >

(run process in background - &)

Everything in LINUX is a file, including directory

Linux kernel

→ System calls

→
- I/O component
  - ↳ VFS
    - Terminals
    - Socket
    - Network protocols
    - Network device drivers
    - File systems
    - Generic block layer
    - I/O schedulers
    - Block device drivers
- Memory management
  - Virtual Memory
  - Paging page replacement
  - Page cache
- Process management
  - Signal handling
  - Process/thread creation and termination
  - CPU scheduling

→ Interrupts

→ Dispatcher - module that gives control of the CPU to the process selected by the short-term scheduler → dispatch latency

(It takes care of context switches, switching to user mode)

Block device drivers — seeks and random accesses are allowed

Processes

initially — 1 thread of control — 1 program counter (keeps track of next instruction)

↓ after started can have

additional threads

daemons — background processes

↳ eg:- cron daemon — starts up like once a minute

↓

sees if something needs to be done

↓

goes to sleep

(disk backup, routine signals, etc.)

fork() — memory images, variables, registers — identical to parent and child

pid — different — 0 for child ; non-zero for parent

↳ this is child's PID (getpid())

Process communication can be done using (I.) Pipes

sort | head

sort's stdout passed to head's stdin

(II) Also using signals

process decides what will happen when signal arrives

(terminate, catch, ignore)

↳ if catch → signal handling procedure

Process can only send signals to its process groups
↳ Parents, siblings, Children

ctrl^C
↓
SIGINT (terminate the application)

- SIGALRM
  |
  the alarm clock has gone off

ctrl^D
register a EOF on standard input

Signals —
- SIGCHLD — child terminates process
- SIGKILL — send to kill a process (Cannot be ignored or killed)
- SIGPIPE — process has written to a pipe which has no readers
- SIGSEGV — process has referenced an invalid memory address
- SIGTERM — Used to request a process terminate gracefully
- SIGUSR, SIGUSR$_2$ — available for application-defined purposes

waitpid() — waitpid(pid, &statloc, opts)

← like waitid() → returns a siginfo_t

pid → allows caller to wait for specific child (if -1 → any old child)

& → it is ampersand (&)

statloc → address of variable that will be set to child's exit status ↓ to know the return status

opts → Caller blocks or returns if no child is already terminated (usually 0)

execve(name, argv, envp)

name → name of the file to be executed

argv → pointer to the argument array

envp → pointer to the environment array (used to pass stuff like
- terminal type
- home directory)

eg:- execve("/usr/bin/ls", ["ls", "-l"], 0x7fffa3746e78) = 0
↳ strace of 'ls'

pause() — suspend the process till the next signal arrives

↳ when a program has nothing to do but stay IDLE and wait for a signal to start action

linux kernel — represents tasks — task_struct

task_struct →
- scheduling (priority, CPU time, time spent sleeping) — which to run next
- Memory image
- Machine registers
- File descriptor table ← ls
- Signals (which are ignored, caught, temporarily blocked)
- system call state
- kernel stack

When fork() — child has to copy memory of parent

BUT — copying is expensive, so — Copy on Write

↓

if multiple callers ask for resource which is initially indistinguishable → give them pointers of the same resource (read-only) → if a caller tries to modify the copy of its resource

↓

create a true private copy for that caller, (this is transparent to others)

→ others still point to original

⑥ clone() → ~~Histor~~ Historically (before linux), file descriptors (open files), signal handlers, alarms, etc. were per process, not per thread

↓

with clone, it can be either per process or per thread

pid = clone(function, stack_ptr, sharing-flags, args)

Linux scheduling — threads not processes

real time threads { 0 — highest priority
⋮
99 — lowest priority

non real time threads { 100
⋮
139

Time (in linux) = no. of clock ticks

(older — clock ran 1000 Hz — each 1 ms)

nice — each ~~P~~ thread — default — 0

range — -20 to +19

↓ high priority

↓

Static priority of each thread

Use system call: nice(value) to change it

How are PIDs allocated

↳ hash table entry for PIDs – if collision – there is chaining

Counter from x to y — increment each time; ~~[illegible]~~ if pid is assigned, it is skipped

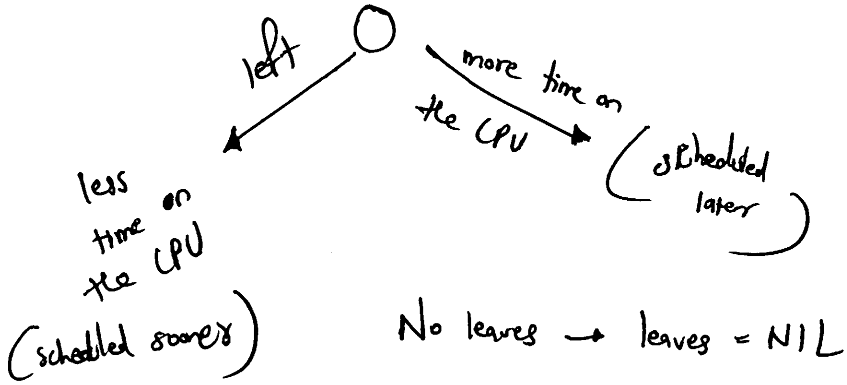
Completely fair scheduler

red black trees

(tasks are ordered → amount of time they spent running on CPU)

"v runtime" → internally called

Internal node → task



No leaves → leaves = NIL

Algorithm → • Schedule the leftmost node in the tree [O(1)]

- Increment that vruntime
- Compare it with the leftmost node in the tree
- If less, continue to run
- If more, go to the appropriate place in tree [O(n log n)]

data structure name

- runqueue — contains the runnable tasks (ready queue)
- waitqueue — associated with events that tasks may weight on
  - spinlock → for preventing concurrent modifications ^ to datastructures like waitqueues

Memory Mangement — Linux

Address space of a linux process
- text — machine instructions that form the program's executable code
- data
  - initialized data — need initial value when program starts
  - Uninitialized data (BSS- block started by symbol)
    - '0' initialized (global variable)
- stack

brk(addr)

for dynamic allocation — linux allows — data segment memory to grow and shrink

→ System call brk() is available to allow a program to set the size of the data segment.

↓

malloc() makes heavy use of it

To allocate more memory, program can increase size of brk()

stack — top of the virtual address space and goes downwards

- Linux has shared text segments for two programs running the same code
- Data and stack segments are never shared except after a fork

Memory mapped file — if a file is in disk

↳ we use the file descriptor in the 'open' sys call to open the file and then:—

read
write
seek —— move the pointer to and fro in the file
close

Operations are done

However, you can use mmap(), to map the physical file to the address space —— entire copying is not done initially

↳ On Demand copying mechanism using the page table

↓

To write back changes

↳ It will be upto the kernel to decide when to schedule a write back to disk

↳ the page table entry has a modified bit which tells if an entry is dirty or not file descriptor

mmap(addr, len, prot, flags, fd, offset)

address at which file is mapped (memory address)

how many bytes to map

protection for mapped file

private or sharable

where in the file to begin the mapping

unmap(addr, len)

└──→ removes a mapped file

(256 TB)

Linux process — 64 bit x86 — 48 bits for addressing

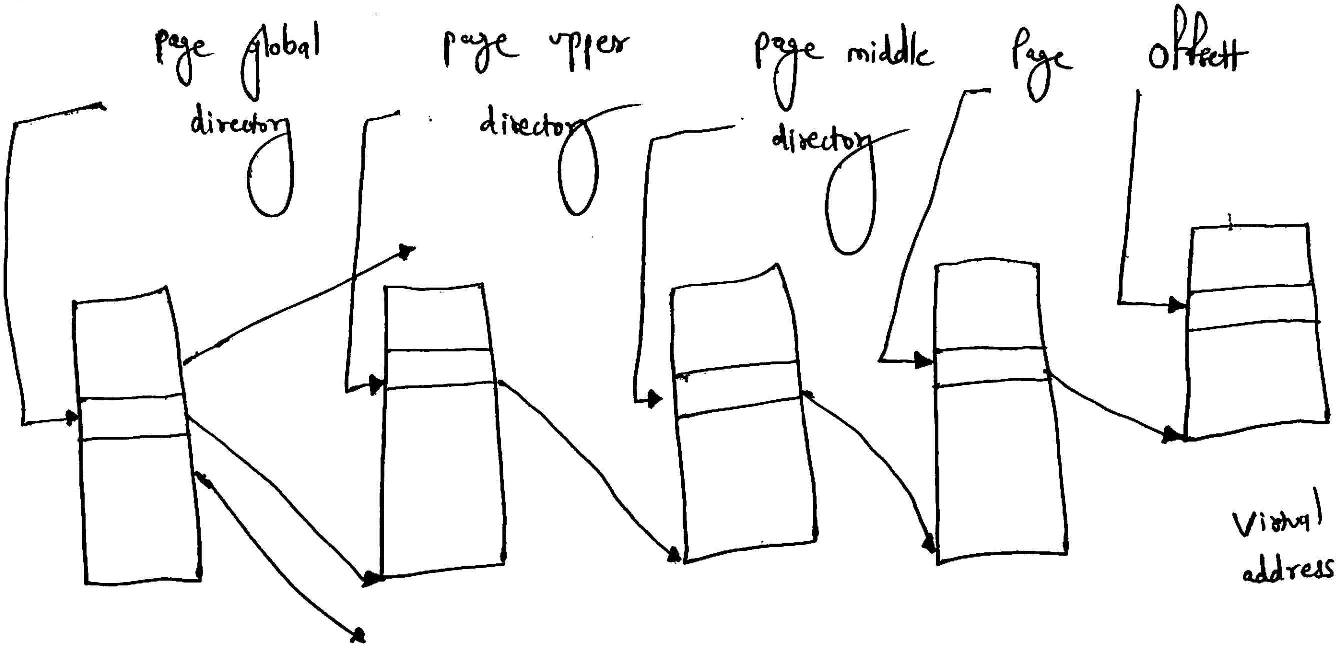
Kernel space
128 TB max
per process
Virtual address
space

User space
128 TB max
per process
virtual address
space

Main memory on linux — 3 parts
- kernel
- memory map
  } pinned in memory (never paged out)
- page frames
  - text, ~~page~~ data, or stack page
  - page table page
  - free list

Linux supports kernel modules (saves space) — pieces of tee kernel that reside on disk till needed
- live in "/lib/modules"
- as kernel needs functionality — loaded from disk and integrated to kernel
  - not used — unloaded from memory ("autocleaning")

Memory map
- zones — memory zones
  - DMA (used for direct memory access)
  - Normal (regular mapped pages)
  - Highmem (high memory addresses, not permanently mapped)
- free page frames



4-level page table Linux

New page frames of physical memory – page allocator – ~~By~~ buddy algorithm

→ divide in power of 2 till you found right size, when freed, combine the buddy of same power ($2^3+2^3 \rightarrow 2^4$)

→ this can cause internal fragmentation

to alleviate this → slab allocator (carves slabs from these chunks and manages them separately)

Virtual machine address space representation → organized as a red-black tree

↳ vm_area_struct

- protection mode
- pinned in memory (non pageable)
- direction it grows in (up for data segments, down for stacks.)
- backing storage on disk
- also records whether area is private to the process OR shared with one or more other processes

during fork(), parent and child point to the same page table → copy on write functionality enabled

mm_struct —stores info on→

- all VM areas belonging to the address space
- different segments (text, data, stack)

In linux:-

Main Memory Management unit = Page (granular)

WHY? → process need not be entirely in memory in order to run

What is required for a process to run? (Memory Wise)

user structure

page table

↓ if not there, Swapping needs to be done.

Paging → implemented

partly kernel

partly page daemon (process 2)

↓ if it sees no. of pages on free memory pages is low

↓ free up more pages

Swap area — fixed length page files

→ each has a priority

When a page has to be thrown out

→ the highest priority page partition or file (still having memory) is selected

Page Frame Reclaiming Algorithm

Pages in Linux

- unreclaimable (like kernel stacks, memory map)
- swappable (must be written back to swap area)
- syncable (if dirty, need to be written back)
- discardable (reclaimed immediately)

The algorithm tries to reclaim easy pages, then proceeds with harder ones.

Backing store: — part of hard disk that is used by paging or swapping system to store information not currently in main memory.

$$\text{Swappiness} = \frac{\text{pages with backing store}}{\text{pages which need to be swapped out selected by PFRA}}$$

→ loop — scans through — Zone's active and inactive list,

reclaim different list with different urgencies

active list

→ used by some process in the system

inactive list

→ kernel thinks might not be in use

(16)

File — sequence of zero or more bytes

- Filename — 255 characters — ASCII (except NULL)
- Directories are stored as files and treated like files

special → /bin — binary (executable) programs

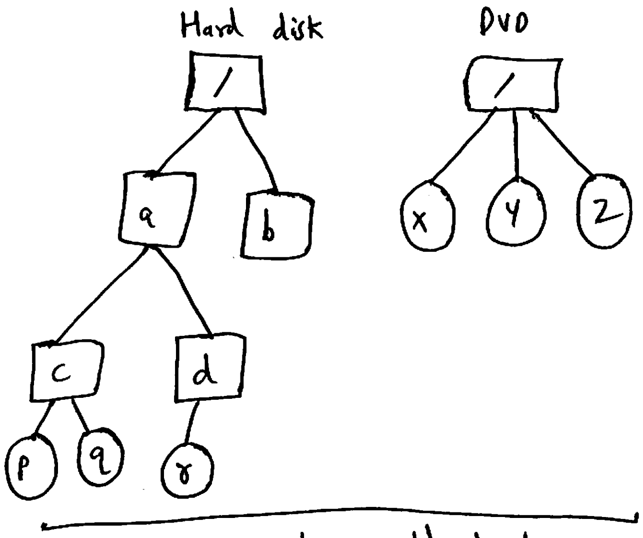
/dev — special files for I/O devices

- /dev/tty — reads from printers
- /dev/lp — if write, writes to printers

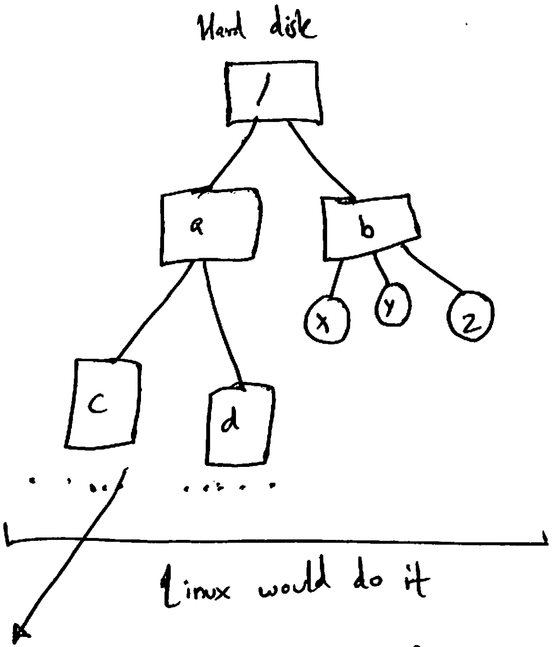
/etc — miscellaneous system files

/lib — libraries

/usr — user directories



how windows would do it



Linux would do it

no longer has to be aware of "which file on which device"

Locks — shared — second attempt to place shared lock will work but exclusive lock won't

Exclusive — all attempts to lock any part of that portion will fail until the lock has been released creat (file_name, mode) — not only ~~opens~~ creates but also opens for writing

mode → protection

returns fd (lowest numbered not currently in use)

file system calls

fd — 0 (stdin)

fd — 1 (stdout)

fd — 2 (stderr)

} already open → shell arranges this

read (file, buffer, nbytes)

write (fd, buffer, nbytes)

nbytes → no of bytes to transfer

fd → file descriptor

buffer → where to put the data or get the data from

lseek (fd, offset, whence) — move the file pointer

lseek → returns the absolute position after the file had been changed

whence → relative to what? beginning? current position? or the end of the file?

file status information [stat (file_name, &buf)

fstat (fd, &buf)

&buf → pointer to a structure where the information requested is to be put

→ device file is on

→ I-node number (which file on device)

→ file mode

→ no of links

→ identity of file's owner

→ group file belongs to

→ file size

→ creation time

→ time of last access and modification

mkdir (path, mode) — create new directory

rmdir (path) — remove a directory

link (oldpath, newpath) — link to existing file

unlink (path)

chdir (path) — change the working directory } ls

readdir (dir) — read one directory entry

↙ no lseek inside

VFS (virtual file systems)

superblock — layout of file system

i-nodes (index nodes) — each describe exactly one file

dentry — directory entry → directory entries are cached in
↓
dentry_cache

file — normal files to be opened with sys calls like open(), close(), creat()

ext2

└→ first block — ① superblock

- layout of file system
- number of i-nodes
- number of disk blocks
- start of the list of free disk blocks

② group descriptor — ext2 wants to "spread directories evenly on disk"

- location of bitmaps
- no. of free blocks and i-nodes
- → no. of directories in the group

③ i-node — • accounting info
• enough info to locate ALL diskblocks that hold file's data

④ data blocks — files and directories are stored here

⑤ Bitmaps — used to make fast decisions
↳ where to allocate the data

Directories are searched linearly (also cached) → in dentry cache

how to traverse a file system in ext2
- ~~file~~ inode is fetched (root has i-node → 2)
  - ↳ all disk blocks are fetched
    - ↳ ~~file~~ directory is read
      - ↳ next i-node is fetched
        - ↳ process repeats itself

file descriptor table (task_struct stores this) ↔ open file description (pointer to i-node) → i-node → disk blocks

(20) load — CPU bounded load
— out-of memory issues
— I/O bound load

Out of memory — System has run out of available RAM

↓

started to go into swap

↓

each process gets slowed down as disk gets used

→ TRAP: easy to rule it out as I/O bound load (as disk is being used as RAM)

↓ so

Check RAM next

- Don't just look at the free column (see — free + cache)

WHY? → you need file → kernel loads it to RAM

↓ after done

If enough RAM available, kernel leaves it → It tries to cache as many files as it can in RAM

↓ so

This free RAM can get smaller →so free + cache

This free RAM can get smaller → so in some cases, we might not need a RAM disk