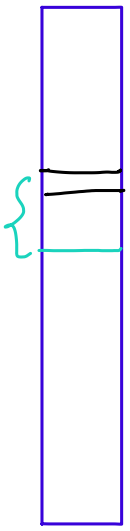
OS Structure



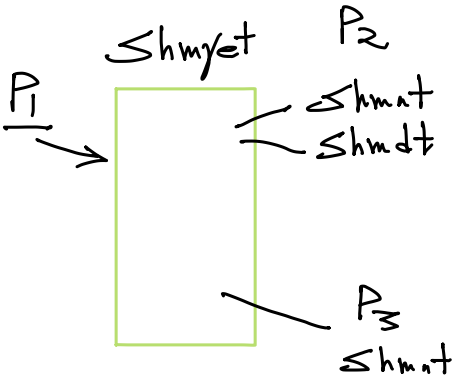
5 bytes

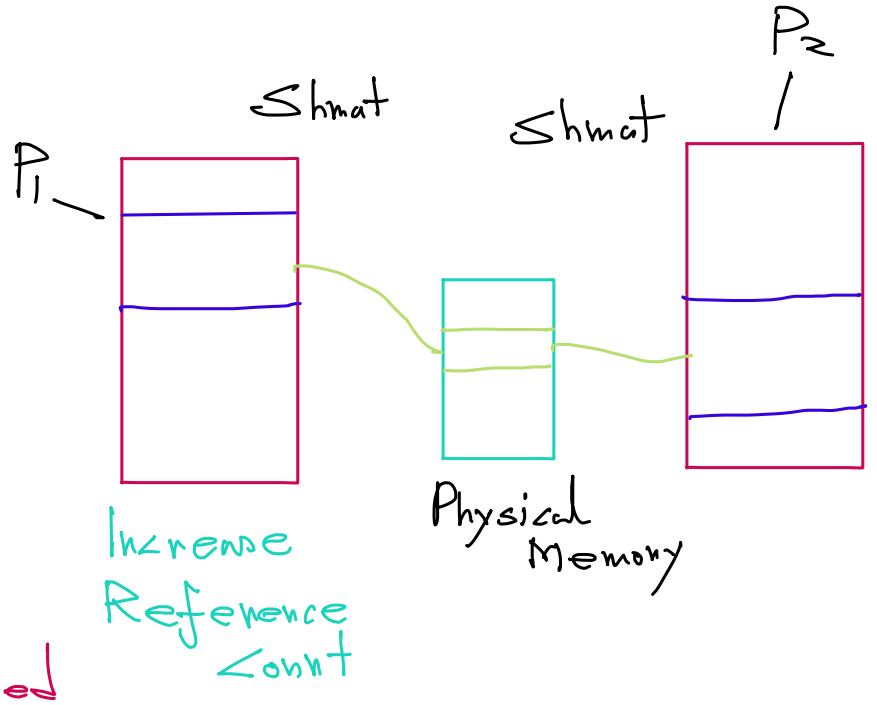
Virtual Address Space

Memory Leak

Process exits ⇒ Free'd

Thread exits ⇒ Not Free'd



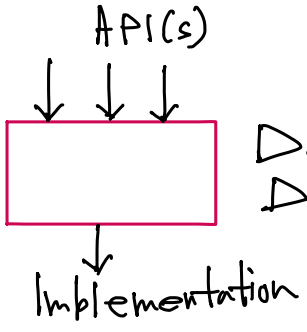


Q) Can we delete shared memory?

⟹ Depends on policy

Q) Too many shared memory create, it will fail, why?

⟹ Run out of Virtual table descriptor space



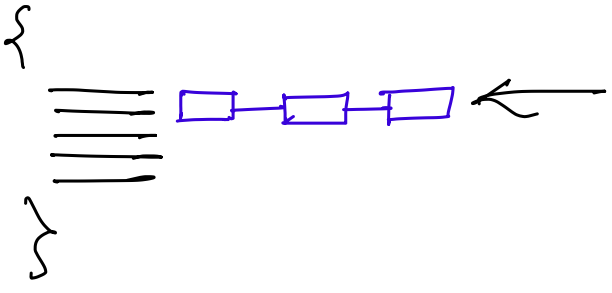
FAT FAT32 NTFS

open() → How we know which file system

VFS → Virtual file system

vfs_*

Process :-



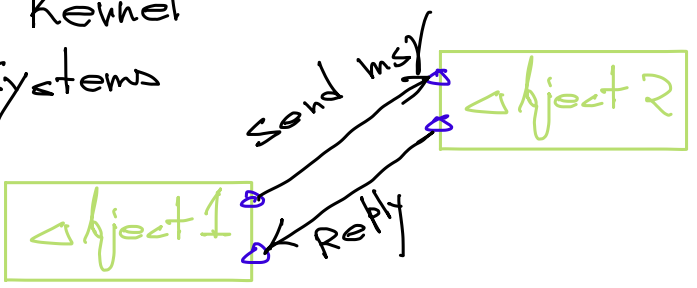
All maybe in different pages

Microkernel Based OS

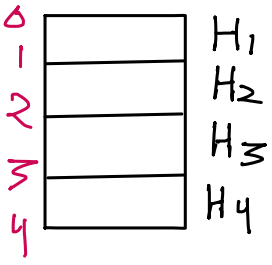
→ Only minimal work in the Kernel

Mach:- Small Distributed Systems

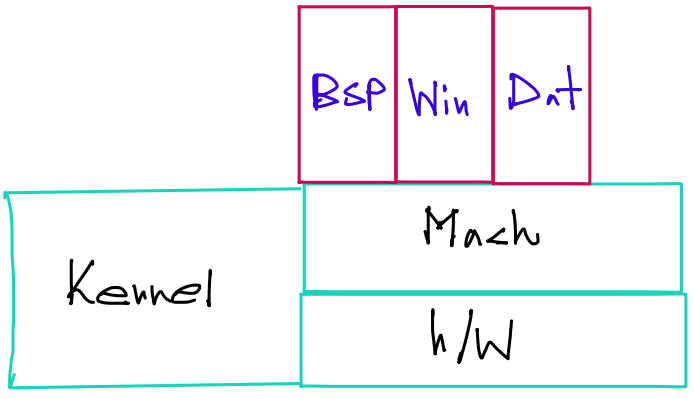
Basic Approach:-



User 4MB
0-4MB



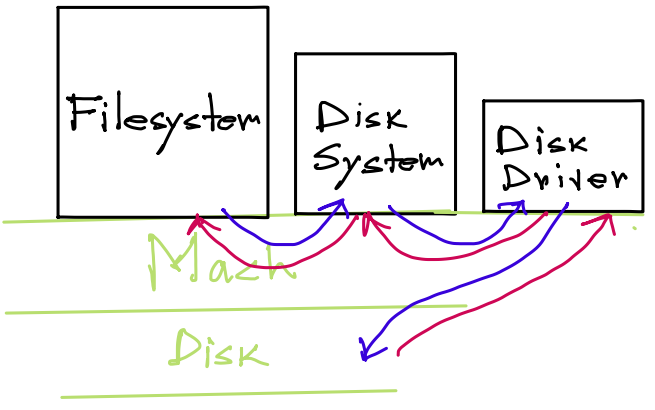
H1	H2	H3	H4
1 MB	1 MB	1 MB	1 MB
0-1M	0-1M	0-1M	0-1M



Modules interact through Mach

Read File

File System Code
↓ Where in Disk
Disk Subsystem Code
↓
Disk Driver + Code



Exokernel:-

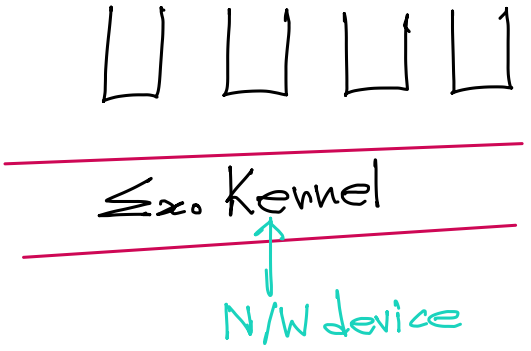
Secure binding example:- Library OS requests X bytes of physical memory. Exokernel allocates block to A.

A implements VM (its own way)

A writes something to TLB VM → PM

Check if phy mem address is within range allocated

Packet Filter:-



Linux Process Architecture

www.kernel.org

Kernel memory cache

- Cache of objects
- On a new object creation, return an already "semi-initialized" object from the cache
- Sequence of continous pages = "slab"

task_struct

exit_state

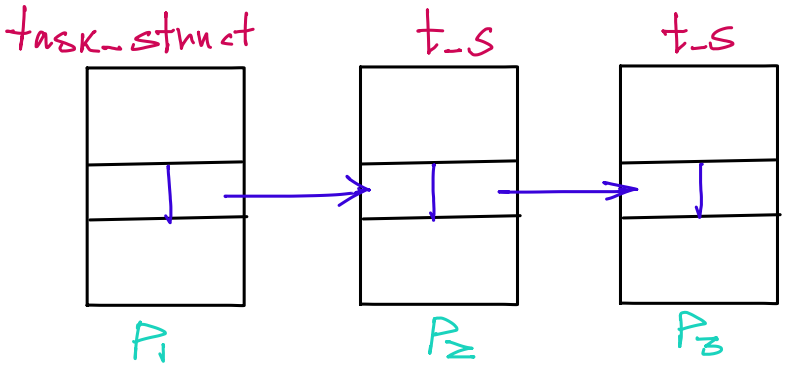
state = TASK_DEAD
exit_state = EXIT_ZOMBIE

Waiting in ready queue is also TASK_RUNNING

TASK_RUNNING = Actually running or in ready queue

TASK_INTERUPTABLE = blocked on event

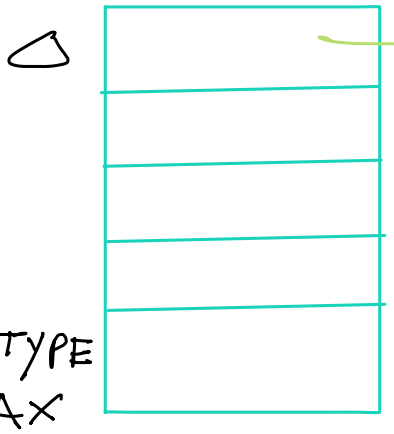
_UN "



So many lists
⟹ Simple macro for traversing the list

struct pid

Three types of PIDs
→ process id
→ process group id
→ session id



→ task struct structure of the process

PIDTYPE_MAX − 1

Each Process
1. PID
2. Task_struct
3. struct pid

⟹ Many other processes can have reference to the process.

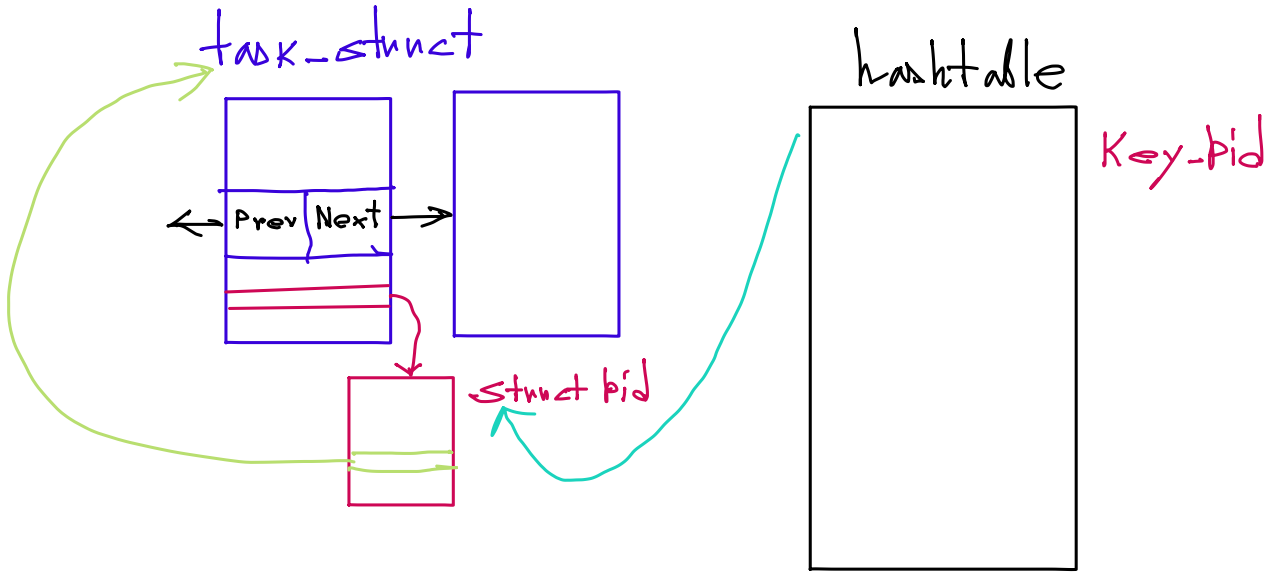
Q) Why not direct map from pid to task_struct?
⟹ Wrong Reference by others

⟹ Each time a process refers to P, the refcount in the struct_pid field is increased by 1. (get_pid)
⟹ Each such process is supposed to call put_pid to decrement ref_count

attach_pid

detach_pid

V.5.10.188



Kernel_clone
copy_process
dup_task_struct
alloc_pid

⟹ Somebody will have to do the put_pid

⟹ First get_pid is done by parent

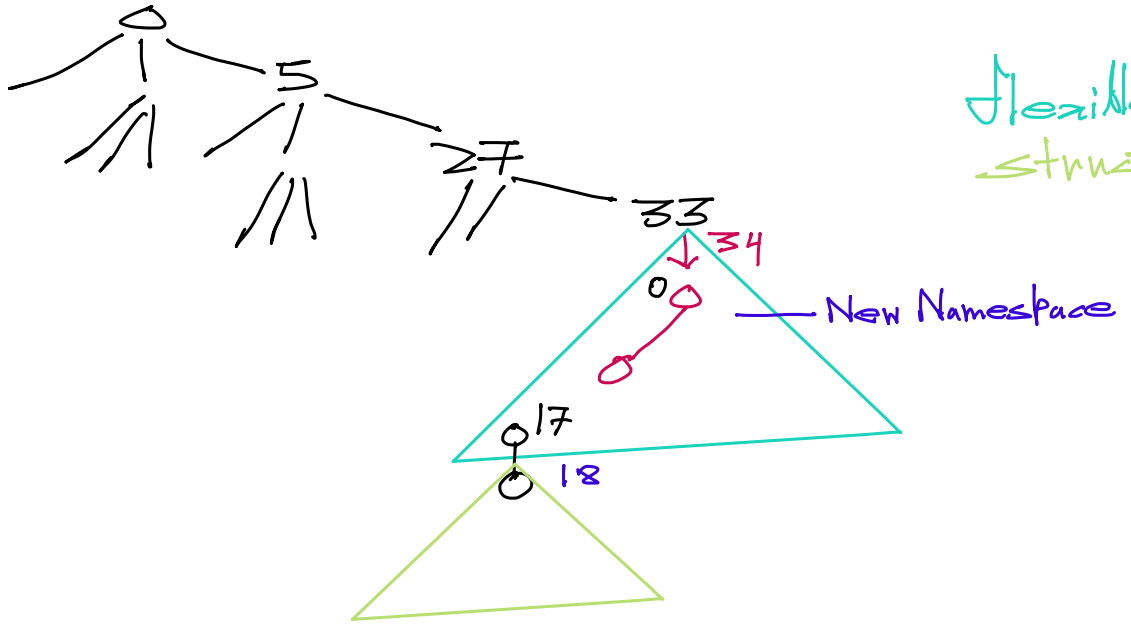
⟹ Where is the exit code of task set?

⟹ struct task_struct

⟹ Check_preempt_curr() → Don't stop the current process put it in ready queue TIF_NEED_RESCHED

level → What level in the pid tree

PID Namespace



flexible arrays
struct upid numbers[1]

P

C

P /usr/bin/ls → /usr/local/bin

< /usr/bin/ls → /linux/bin

init_main.c

start_Kernel

⋮

rest_init

⋮

run_init_process

⋮

Kernel_process

fork()

"User-Space"

system_call Wrapper

System Call

INT 80 Syscall

"Kernel"

Clock → Interupt → Increase Counter

→ The code itself should not take too much time

⇒ Store the parameters in specified registers

⇒ Store a "System Call no" in specific reg

⇒ Trap

Scheduling

Schedule:-

① Which task to schedule

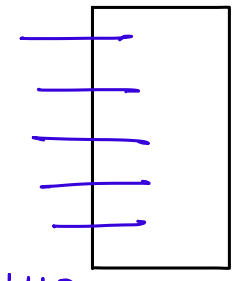
② How much time slice to give to a good task

Task

Priority 1 — 130 x ≥ 0

80 — 60 x ≥ 0

} Reverse it



140

Active Dynamic Priority → How much sleep?

⇒ Not go to expired queue until time slice over

O(1) Why Problem?

Suppose you have only ≥ priority 139 processes

⟹ Frequent context switches

Completely Fair Scheduler:-

Goals of FCS:- ① Each process should get a "fair" share of CPU Based on its priority

② Interactive tasks should run "earlier"

As of V.6.6 (Oct'23) EEDVS scheduler is default

Problem:-

If too many processes, many context switches

Scheduler Basic

A "scheduler tick" (timer interrupt) every x (fixed, but configurable ms)

At every tick, recompute vruntime of current task

Check the new min vruntime task

Recompute the slice

Schedule the task.

⟹ A process may run for time > timeslice

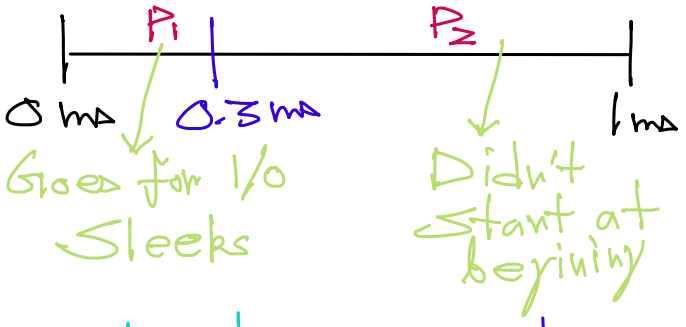
(Till next schedule tick)

Imp sched_entity — class specific

sched_class — one_structure for all classes (fair, rt, deadline, ...)

sum_exec_runtime → Total CPU time

exec_start → Starting time on last tick period



Note that at the end of Scheduler tick the current process may not be the one at the start of the tick.

struct rq
struct cfs_rq

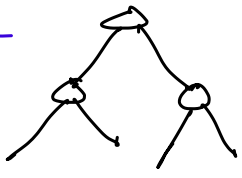
curr → normal_prio
↑
Parent

Data structure related to Scheduling

struct sched_entity — per process
struct sched_class — One per scheduling class
struct rq — One per CPU (for all processes)
cfs_rq, rt_rq } — One per CPU per class

updating min_vruntime

leftmost



cfs_rq
{
min_vruntime = 200
}

What if curr's vruntime = 300
left's " = 100

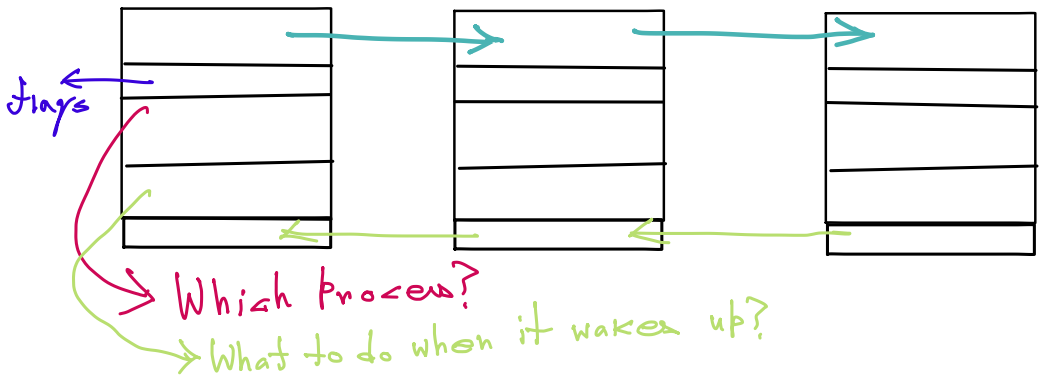
Vruntime set to 200

When a new process comes, it's initial vruntime is set to cfs_rq min_vruntime

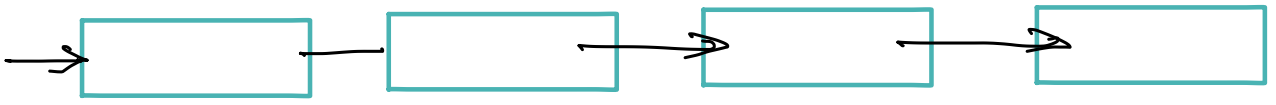
How is this possible
1) 300 was earlier 50
or
2) 100 was not even there

dl_sched_class
rt_sched_class
fair_sched_class
idle_sched_class

Sleep and Wakeup



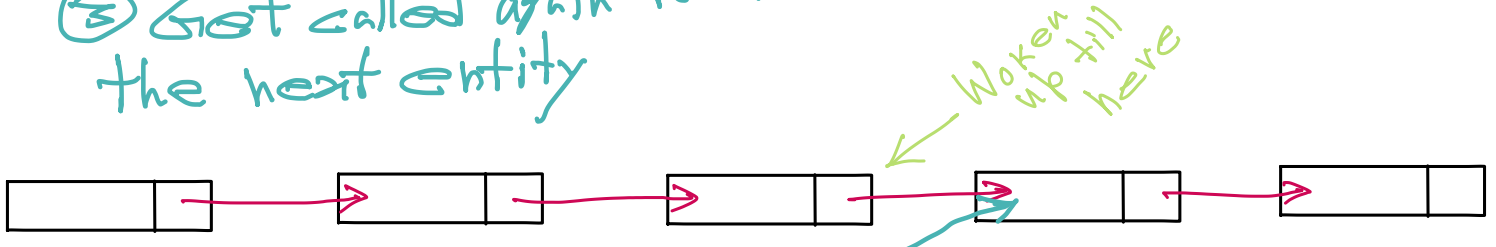
EXCLUSIVE broadly
→ for events that when woken up, only one process is selected to run
(eg., a binary semaphore)



1. You always wake up non exclusive process
2. You wake up x no of exclusive process
(x is a parameter, "nr_exclusive)

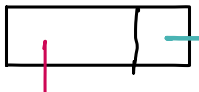
What if there is a "large" no of non-exclusive processes the wait queue? ⟹ clogging the queue, New ones will not be able to enter

⟹ So, what it will do is
① Wake up upto a max count
② "Bookmark" upto what is done in the queue
③ Get called again to do the next starting from the next entity



Book Mark entry

Pointer to it →



Set this before return in last call

Set the flag to WQ_BOOKMARK (indicate already done)

Example:- 1) lock_sock 2) release_sock

Deadline Class

SCHED_DEADLINE — Scheduling — Modified version of EDF

Each process has:- 1) Execution time
2) Deadline
3) Priority

sched_setattr
sched_getattr
↓
Issue:- Everyone would want to be SCHED_RT

Embedded OS

Real Time Tasks

→ Static Scheduling

Static/Fixed Priority

Sufficency Test
✓
Necessity Test

$\sum \frac{C_i}{P_i} > 1$
⇓
No feasible Schedule

RM

1) only for periodic tasks
2) Only deadline = period
3) less computation
4) less preemption
5) lower Schedulability Guarantee Range

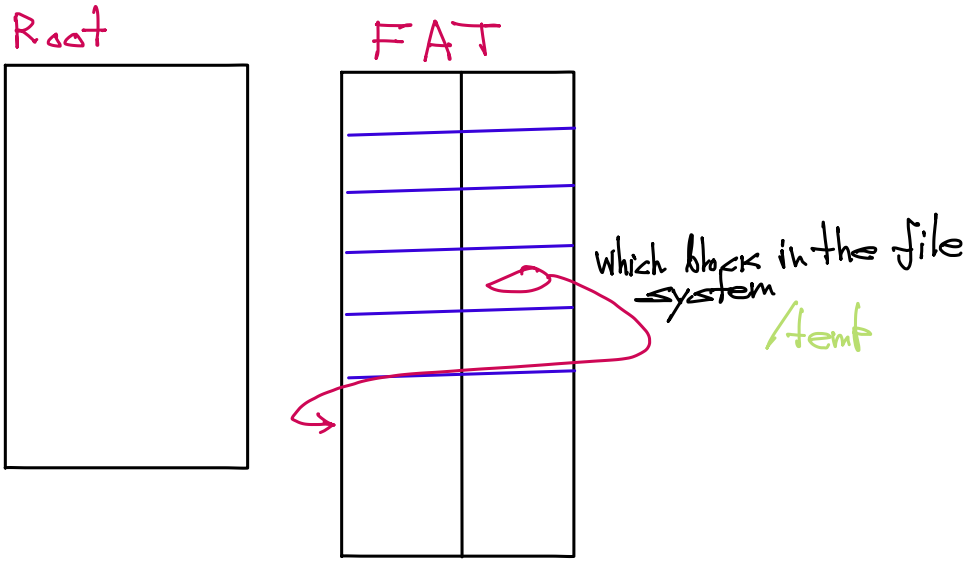
EDF

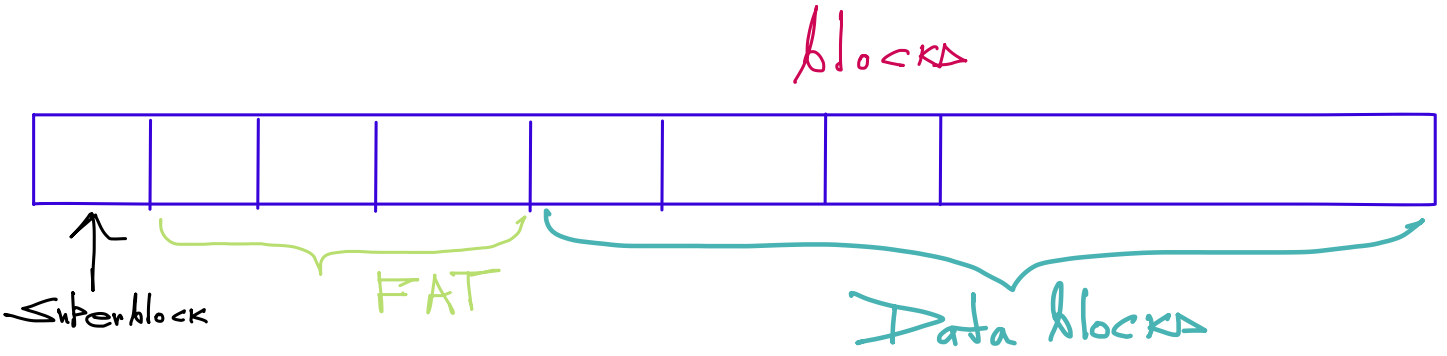
1) Periodic & aperiodic tasks
2) Deadline ≤ Period
3) More computation
4) More preemption
5) Higher
6) Minimizes maximum lateness/Tardiness

→ Partitioned, Global

File Systems

File Allocation Table:-



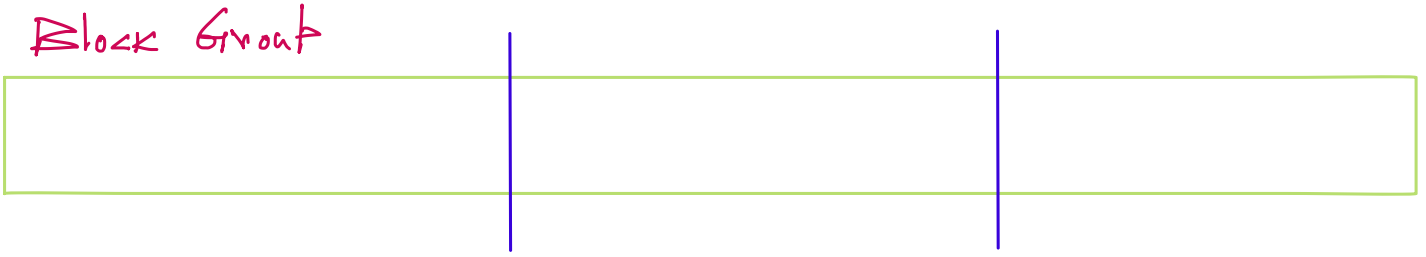


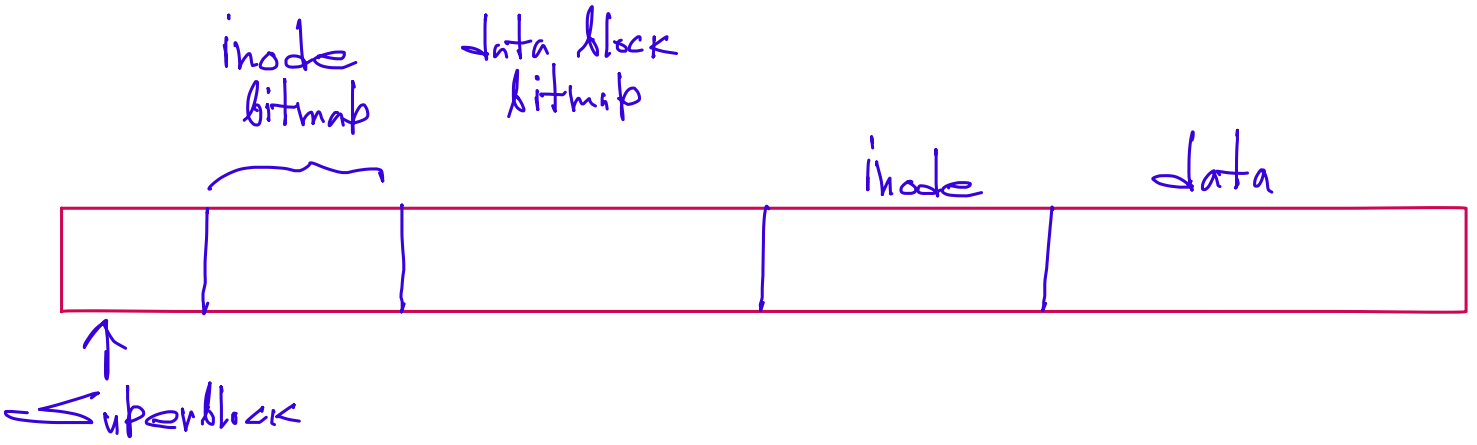
i-node:-

find /usr/tmp/ay.txt

1) Get the inode of /
2) inode gives a data block
3) From data block get /tmp
4) Go there and get location of ay.txt
5) Find ay.txt
6) Go there

6 block accesses





Operation:-

Write to a file (New)

- Allocate it an inode (writing into bitmap)
- Allocate a data block (Writing data bitmap)
- Write info into the inode
- Write to the datablock

The ordering can be of different types

Crashes

inode waste

datablock waste

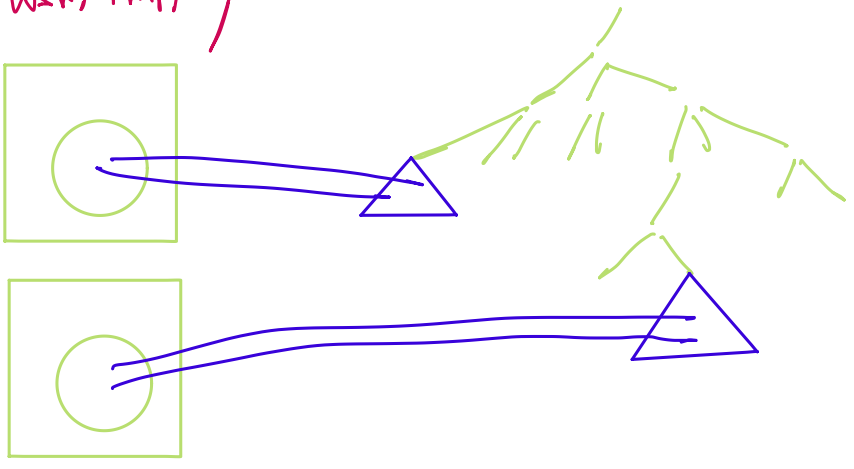
Point to garbage

Fsck:-

Distributed File Systems

Naming:-

a.b.com//var/tmp/ay.txt ← This is not good.



Need to maintain Global namespace

Security

Reasons:-

Don't allow recursive mounting

(Design Choice)

Server crashes:- Every info about the open fd is lost

RPC:- Remote procedure calls

naming scheme — Non-uniform namespace with mounting

/usr/tmp/bin/ls

lookup( )

— Mounted from other server

file handle ⇒ No need to tell whole string of filename

stateless

Caching:- 1) Not strictly part of NFS

2) But always there in implementation

Directory

File attribute refreshed every x sec ($x \approx 30$)

Data block refreshed every 3 sec

Sharing semantics:-

— No clear sharing semantics defn

NFS was designed for sharing files, mostly read-only, between "small" set of users over LAN ⇒ "fast"

Typical Use:- large part of shared read-only data mounted

≈ all your bins

User-specific data mounted in home directory (writable, but mostly only user writes)

VH → Not in exam