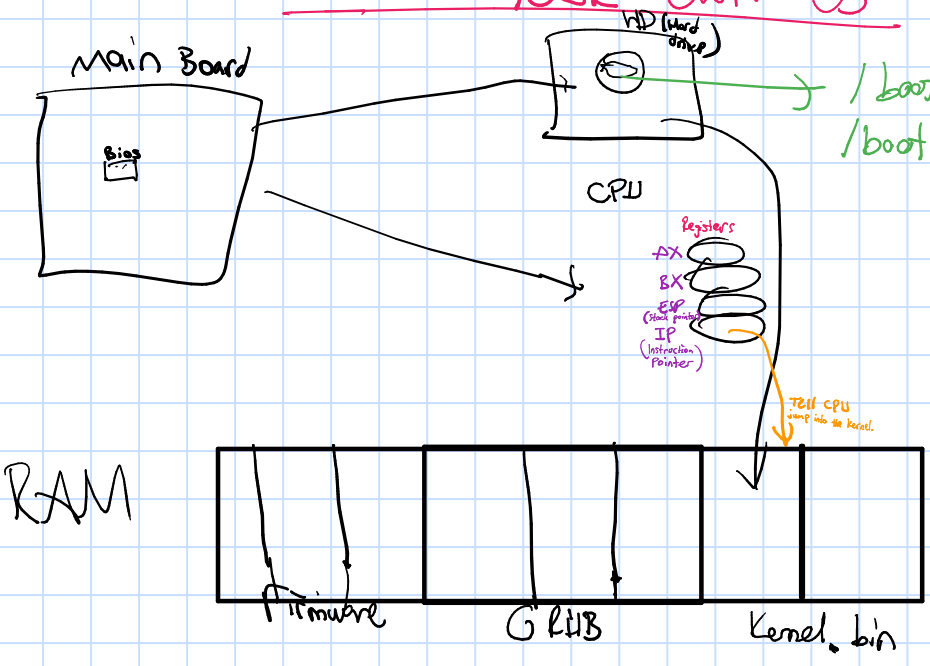
WRITE YOUR OWN OS



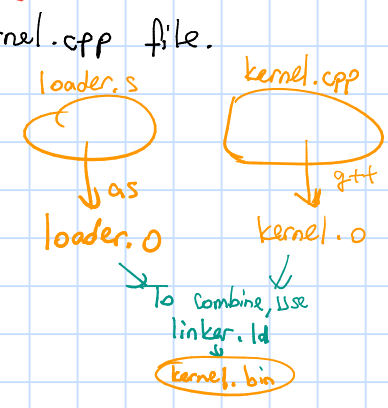
Notes:

When start computer, main board will take the data from the BIOS, it copies contents from BIOS to the RAM.

Then it tells to processor put instruction pointer into the RAM position. It means, CPU will read the instruction there and execute it. (Execute the firmware)

Hard drive content will be loaded into the RAM as bootloader.

loader.s ⇒ Set the stack pointer and jump into kernel.cpp file.



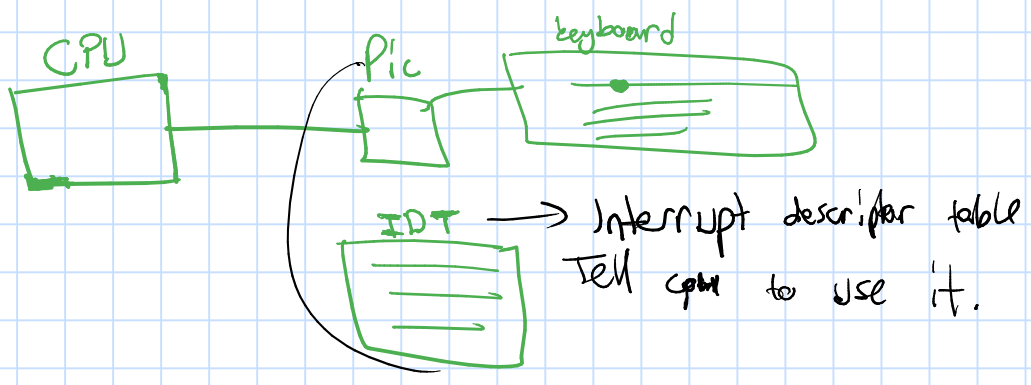
Foo bar(42)

```
void xyz()
{
    Foo(42)
}
```

↳ Constant Not given in xyz's parameter.

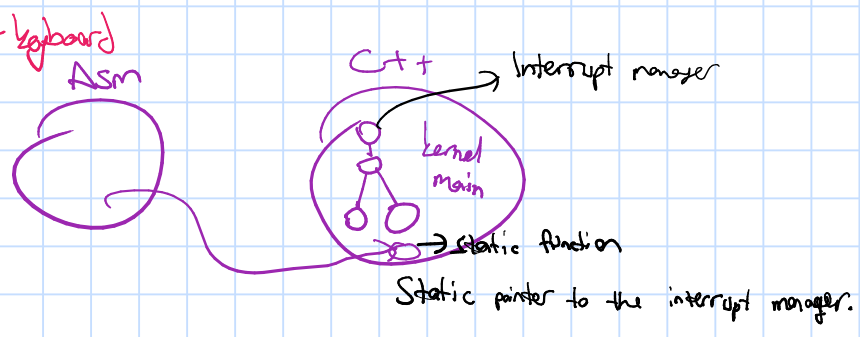
For ex: Assembly ⇒ int 4 bytes
Kernel CPP ⇒ int 8 bytes (unsigned)

Pic: When you click the keybard, you get programable interrupt controller.(p.i)



ESP ⇒ current stack pointer

Pic will sent interrupts to CPU.

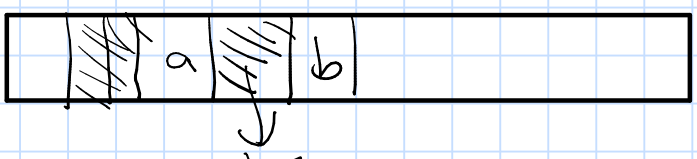


Makefile

```
GPPPARAMS = -m32
ASPARAMS = --32
objects = loader.o kernel.o
LD-PARAMS = -melf_i386 # Executable and linkable format
%.o: %.cpp
    g++ $(GPPPARAMS) -o $@ -c $
%.o: %.s
    as $(GPPPARAMS) -o $@ -c $<
mykernel.bin: linker.ld $(objects)
    ld $(LDPARAMS) -T $< -o $@
install: mykernel.bin
    sudo cp $< /boot/mykernel.bin
```

To make bootloader believe it's the real kernel.bin file, it will look for magic number. So we need to put the magic number in there.

To accept multiboot structure, you can use void*.



Don't override by mistake, then you are safe.

Linker.ld: Selects sections from different .o files and combine them to new sections.

```
SECTIONS
{
  . = 0x0100000;
  .text :
  {
    *(.multiboot)
    *(.text)
    *(.rodata)
  }
  .data
  {
    start_ctors = .;   /*Constructors for global objects. */
    KEEP(*(.init_array));
    KEEP(*(SORT_BY_WIT_PRIORITY(.init_array.*)));
    end_ctors = .;
    *(.data)  /* Collect data sections from .o files */
  }
```

Bss section: Section of memory in an executable file or program that contains uninitialized global and static variables. These variables are initialized to zero by the OS when program is loaded into memory.

Race condition

Producer produces and consumer consumes in synchronize way.

Semaphores

Down(S), Up(V)   No sync in single thread app.

Only down blocks.

```
Semaphore mex = 1;
down(&mex);
/* critical region */
up(&mex);
```

→ Implementing is done in kernel
→ Test and lock

Mutex

Implementation

```
mutex_lock:
    TSL REGISTER, MUTEX
    CMP REGISTER, #0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
    RET
ok:
```

Shared Memory Between 2 Processes

System call ⇒ Context Switch so slow

Sometimes before say mutex available; maybe busy waiting is better.

⇒ Futex ⇒ like mutex

trylock ⇒ Tells if get mutex or not. Doesn't block process.
↳ for ex: you can use it 100 times and then use.

Condition Variables

Under what condition you will block

→ Producer waits if buffer is not 0. That means there is something to consume.

pthread_cond_wait(&condp, &the_mutex);
↳ Temporary release the lock. ↳ makes it 1. (Releasing the lock)

Monitors

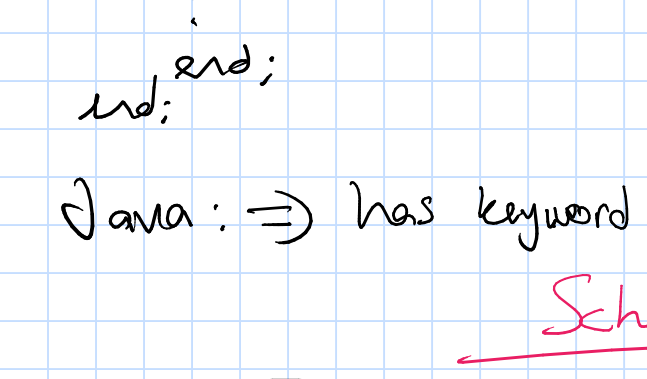
Similar to class, directly gives mutual exclusion

Monitor example → contains cond variables wait, signal.

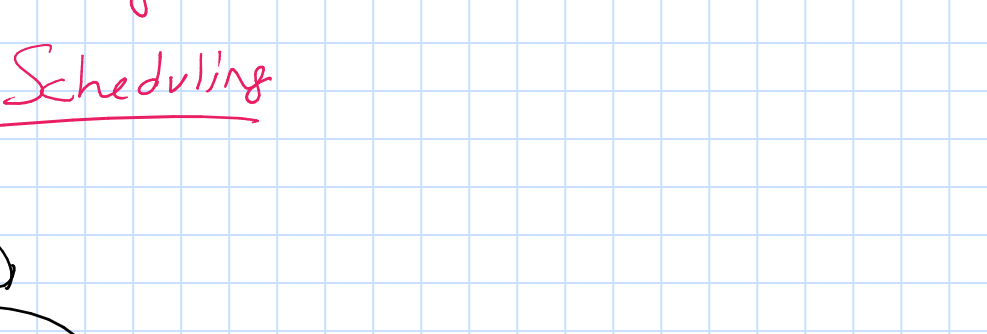
```
int i;
cond c;
procedure producer();
  ⋮
end;
procedure consumer();
  ⋮
end;
end;
```

Java: ⇒ has keyword synchronized

Scheduling



CPU Bound:
I/O-bound



DMA?

How does OS knows when I/O completes (if any system calls didn't happen?)

After some time (20 ms) timer interrupt occur.

In non-preemptive OS scheduler: Process Run as long as it wants, only switch when it blocks

In preemptive: Switches after a time quantum. (20 ms)

Scheduling Algo Categories

Batch — Expensive purpose %100 CPU util

Interactive

Real-time — Often non-preemptive

Maximize CPU utilization
Maximize throughput
min turn around time
↓
First come, first served or Shortest first

Goals

Fairness (Give each process its share of the CPU)
Policy and enforcement: Prevent subversion of OS scheduling
Balance: keep all parts of the system busy

Preemptive (Efficiency time) makes us lose time.
↳ Makes us context switching, expensive.
If one block, then turn around.
Don't preempt based on timer.

Shortest Remaining Time First

A B C D E
2 4 1 1 1

A(2) | B(1) | C(1) | D(1) | E(1) | B(3)

Invalidation of cache creates problems.

Priority Inversion

p(H) > p(M) > p(L)
↳ If L holds lock of H, it may create a deadlock.

Aging: Changing priority number, how much do you use.

P_1: 2 ms
P_2: 10 ms → Changing quantum time of P_2 lower, so that interactivity increases.

Keeping Priorities Organized

Using Priority Classes
Run round-robin schedulers

In batch systems, we don't have priorities. (Usually all the jobs are equally important)

Check linux scheduler.pdf in columbia.edu (In chat)

Quantum time too small makes so much switching.

If shortest process next vs guaranteed scheduling,
If interactivity is better for one of them,
or user experience (mouse, keyboard)

Lottery Scheduling

	P_1	P_2	P_3	P_4
Pri:	10	20	20	40
	1-40	41-65	66-90	91-100

No one starves.

Fair Share Scheduling

U_1: P_1 P_2 — 1/6, 1/6
U_2: P_3 — 1/3
U_3: P_4 P_5 P_6 — 1/9, 1/9, 1/9

Realtime System

Meeting the Deadlines — Avoid losing data
↳ No such things as waiting
Predictability. (key press - 20 ms)

Windows, Unix ⇒ Interactive systems

A	B	C	IDLE	A	B	C
20	20	20	20	20		

Guarantees that A will run in every 80 ms.

Control A doesn't exceed 20 ms
↳ They are designed with the system itself.
↳ You can't install your own programs.
↳ No preemption in RTOS.

Soft Real time: Rarely occasional deadline is tolerable.
↳ Video player: 50 ms - 1 ms (20 frames/s)

Hard Real time: No toleration

Usually we have slots and run 1-by-1.

Scheduling Periodic Event

	A	B	C
Period	100	200	500
CPU usage	50	30	100

$\frac{50}{100} + \frac{30}{200} + \frac{100}{500}$

$0.50 + 0.15 + 0.20 = 0.85$ → Schedulable

↳ Very special programs in the systems.

Mechanical hard drives are not allowed because they are not always predictable.

Aperiodic Systems

It uses different address spaces. A1 - B1 (Different threads) (If needed they can do this)
cheaper ← A1 - A2
↳ Because we cannot make cache content same, we need to invalidate it.

Dispatcher thread is important, many workers but 1 dispatcher.
↳ Scheduling this thread is more important.

Philosopher
↳ After thinking put mutex.
Down (mutex)

Memory

IBM 7094 (32k words) ⇒ Now 150k more

Main memory is a resource
OS is managing the resources

Early MSDOS
Relocation problem

Static Relocation

Actual offset given by compiler and linker.
Problem: No protection

[A] Base and limit registers are done by the hardware.

Jmp 38 ⇒ Add base address + 38
when context switch happens addresses change

Protection solved.

Problem: Still have to load all program to memory

Address space: Each process have it's own address space.

Binary addition might be expensive since carries (Sequential so expensive)

Solution: Swapping and virtual memory

Compaction ⇒ You have to move processes a lot.
Even I have enough memory, I need to move a lot.
Disk access is slower than memory access.

Swapping only some part of program is enough (Overlay, Partial programs)

Memory Management with Bitmaps (Old) ✗

MMU gets virtual address and converts to physical address

$2^{16} = 64$ KB

4 bits	12 bits

4 bits → Page number

Virtual mem ⇒ 64 kib → If exceeds put a part in disk, and write to physical memory.
Physical mem ⇒ 32 kib

Good thing: Only necessary things in memory, others in disk. (Paging)

Principle of Locality: Helps; no compaction, not many swapping, assoc memory

Each 4 KB of chunk is called Page frame
↳ If you use 2 KB other 2 is wasted. (Internal fragmentation) It is not avoidable

Small is good to make wasted less.
But if you make every chunk like 128 B, it's too small.
So page table size will be so much.
4 KB of page sizes are good enough.

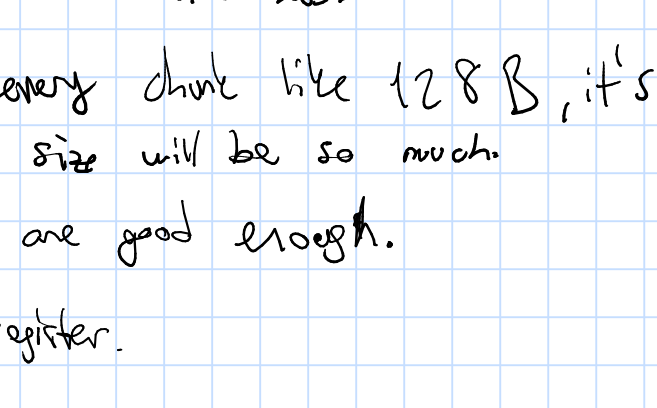
→ Faster than base register.

→ Mapping is done by pages. Each 4k bytes

Done by hardware ← <M[V], O> Concatenate

Page table jobs → Done by OS

→ If virtual address space == physical address space, all can be 1.
Usually slots are empty
→ Read only pages are useful.

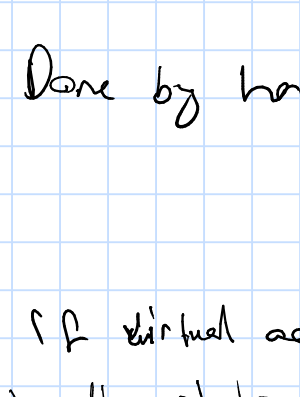


Share memory and makes ipc is easier.
MMU have the information.

Dirty bit ⇒ Modified bit   Table elements, ask gpt?
↳ If 1, I have to write to secondary storage.

Caching disabled ⇒ Page shouldn't be kept in cache.

If I have 1GB of memory



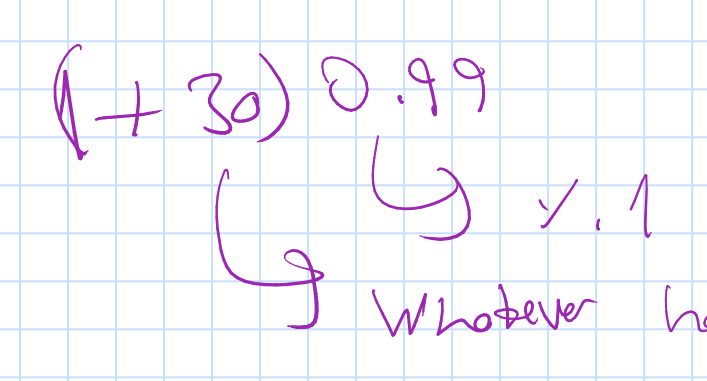
If process have 4 GB, and each page has 4 kB
2^{32} virtual addresses
$\frac{2^{32}}{2^{10}} = 2^{20}$ for each process table.
$2^{20} \cdot 2^{2} = 4$ MB
↳ is not realistic because I can't fit

If page tables don't fit in memory. management unit registers.
Put page table in RAM. (It's small)
Solution: TLB (Translation Lookaside Buffer)
↳ Associative memory
↳ It's like cache
↳ Speeds up paging

Because of principal of locality, most memory references will be on same page.

→ On many systems TLB is managed in hardware,
RISC systems, implemented in OS. (Use hardware for more registers.)
↳ windows and unix

→ Many virtual pages but 1 TLB; so running process is using it. (In context switch invalidate it.)



→ If this is 0, this is junk.

$(1 + 30)\ 0.99$
↳ %1 miss rate
↳ Whatever happens, we will go to real memory to get address.

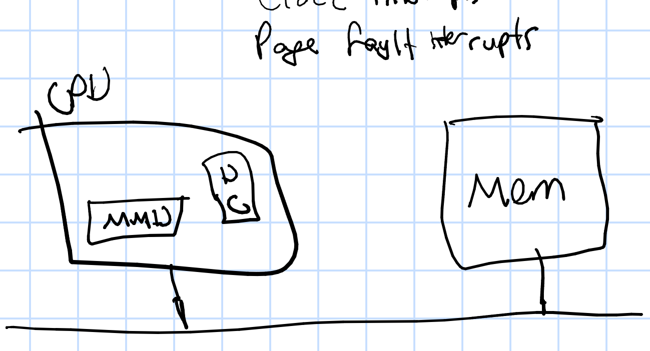
Minor Page Fault: It's not in TLB, but it's in memory, Another process brought it.

→ Paging the page table (Multi level page tables READ)
Most of page table is on the disk. Not in memory.

18 April Week

OS tools ⇒ Referenced bit (page has been used recently)
Modified bit (Discarding this page will be more expensive)
Clock interrupts
Page fault interrupts

Maximum Virtual memory Space
On 32 bit → 4 GB
On 64 bit → 256 TB ~ 16 EB
(4 level page table structure)

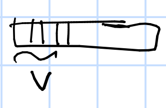


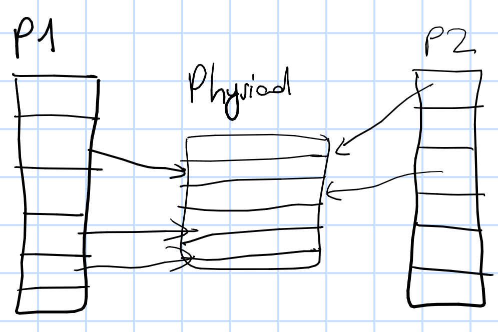
Inverted Page Table

→ 1 entry / page frame in real memory
NOT 1 entry / page of virtual address space

→ Save lot of space but virtual-to-physical address translation is harder.
→ Using TLB and hash table solves problems of finding physical page by using p as index.

$H(v) = v \% 4$
↳ IPN





Page Replacement Algorithms

Algo	Comment
Optimal	Not implementable
NRU (Not recently used)	Very crude approximation of LRU
FIFO	Might throw out important pages
Second Chance FIFO	Big improvement over FIFO
Clock	Realistic
LRU (Least recently used)	Excellent but difficult implement exactly
NFU (Not frequently used)	Fairly crude approximation of LRU
Aging	Efficient approximate LRU well.
Working Set	Expensive
WSClock	Good efficient

- NRU and FIFO is not used.
- WS Clock → we cannot use in all cases
- NFU with Aging is good
- Second Chance FIFO

Memory Requirement Change

Processes and working sets grows and shrink.
Monitor the page fault frequency (PFF) for each process
PFF ∝ allocation size

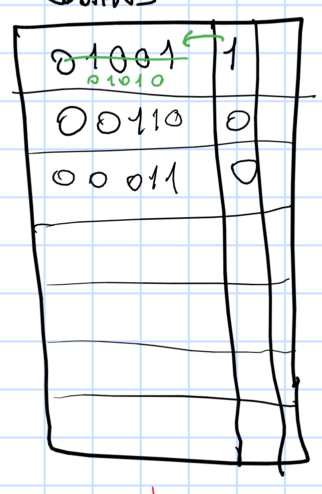
Least recently used

LRU ⇒ Hardware help is needed.
Using 64 bit instruction counter.
if least recently used instruction counter is small,

Software Simulated LRU

→ Have an array of counters, one per page
→ At each clock tick, add value of R to the counter

→ Problem: Never forgets
(Counter always increases)
Counter doesn't tell how recently that thing is used, it just tells how frequently that page is used.



Actually it measures not frequently page used. (NFU)

NFU with Aging

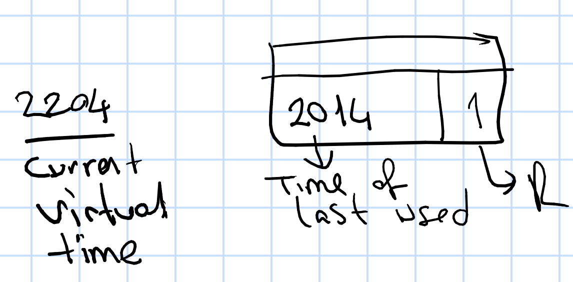
- More recently used have more weight.
- (Shift each counter right - divide by 2 - before each addition.
- Every clock tick the R bit is updated.

Thrashing (Bilgisayar kasması)

- Where a process repeatedly page faulting.
- If thrashing happens we should not continue.
  he process doesn't have enough frames to support the set of pages needed or fully executing an instruction.
  disk traffic and it's expensive.
- Solution 1: More memory → Expensive
- Solution 2: Don't run so many programs at the same time.

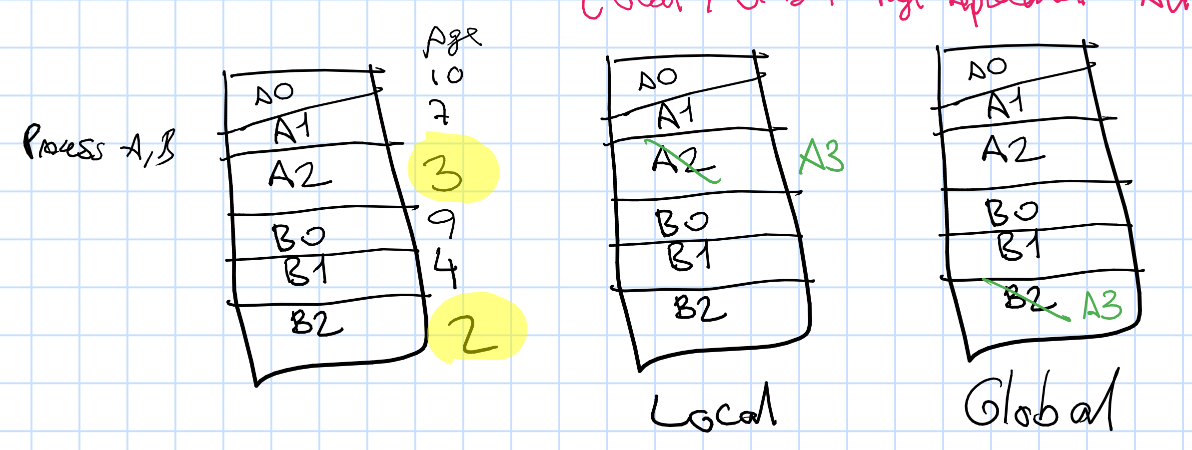
To prevent thrashing WSClock

Tries to find the pages that are not in your working set.
If you cannot fit your working set in memory, thrashing happens.



Q: Only exception doesn't clear out the R bits at every clock interrupt?
A: FIFO/LRU: Doesn't use R bit. LRU do more complicated thing.

Local / Global Page Replacement Allocation

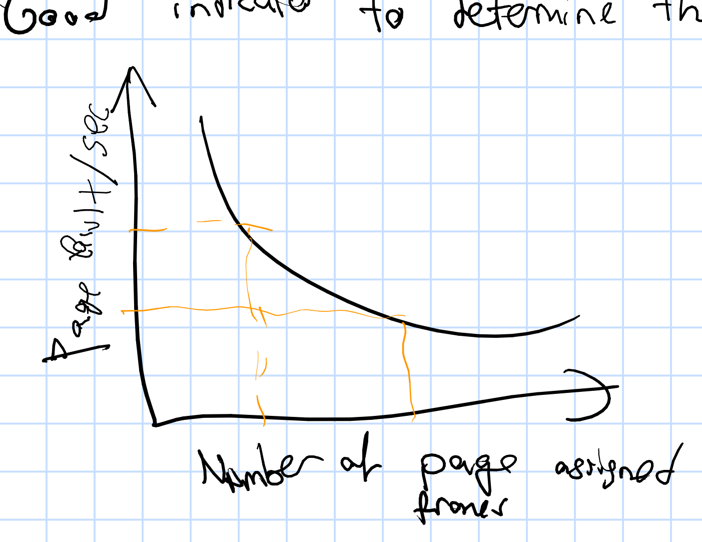


Adding A3
Local vs (Global) → Better

If you use Local, one process might be thrashing while others still efficient.

Understanding Memory Requirement Changes

Monitoring page fault frequency (PFF) for each process
If process has high PFF, it gets larger allocation.
Good indicator to determine the size of page allocated.



Working set clock is nice but it can be used as local allocation replacement algorithm.

If memory is not enough there will be thrashing. We can use swapping.

Page Size	+ sides	− sides
Larger	Smaller page table	Wasted memory
Smaller		

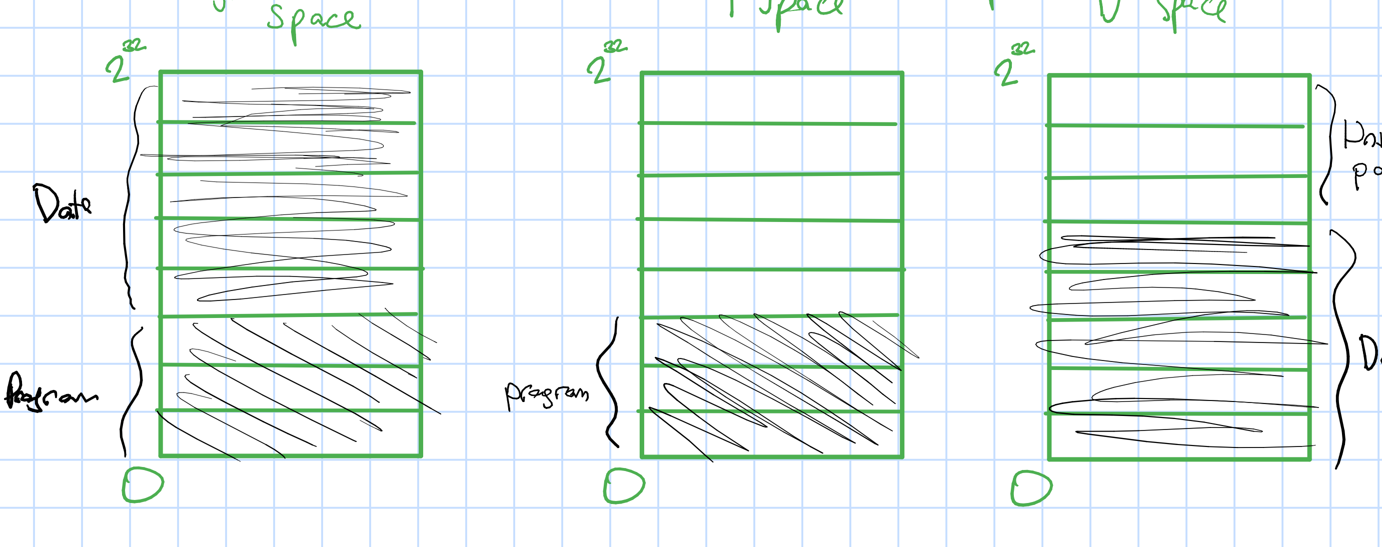
$$O = \frac{s}{p} e + \frac{p}{2} \qquad p = \sqrt{2se}$$

s → process size, e → table entry size

Seperate I and D Spaces

- Most computers have single address space that holds both programs and data.
- If address space is small, things get difficult
- Seperate I and D spaces (Instruction and Data)
- Now they are used to divide the L1 cache.

Question: GOOD EXAM QUESTION
How can you take advantage of different instruction and data spaces when you design a page replacement algorithm?



Answer: For I space, modified bit cannot be 1. Always has to be 0.

25/04/2023

WS (working Set) needs to be local
When menary is not enough thrashing occur.

Swap: Write program into disk when blocked

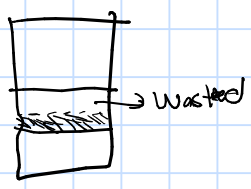
for example swapping out largest process ⇒ 2 level scheduler.

2^{N} N⇒10..12 Translation by MMU

To choose N:

If N is small:
- if N is 0, each page is 1 byte.
- No fragmentation
- We need to access so many pages.
- The other will be 0 bits, so all will be virtual address.
- Page table size will be as long as program size (for ex 1 mega entries)
- Each entry will hold may things.

If N is large:
N=30 Each page will be 1GB long.
Too many page faults.
60 processes, it will cause thrashing.
2 pages for each program, wasted physical menary.



$\frac{S}{p} \cdot e + \frac{p}{2} = 0$ (process size)

$p = \sqrt{2se}$

page size ↓ entry size ↓ overhead (ek yük)

large is good

Windows ⇒ 4 KB page size

Some systems have instruction-data section seperately

After fork, data space is different but instruction space is same.

Its okay to share many pages between processes.

Copy on Write: Context switch overhead reduced by page sharing.

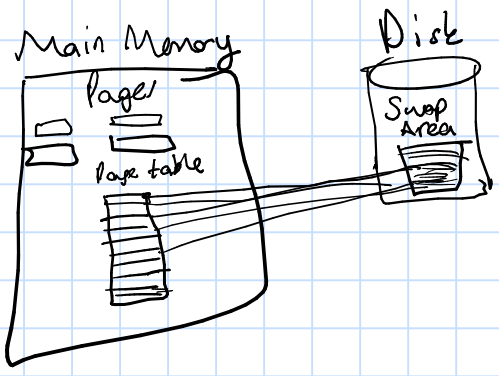
Modified pages are not much.

Most of the time, we only read so we can share between processes.

Shared Libraries (DLL files)
↳ for ex: GUI functions

Soft Miss: Use only link to menary
Usually caused by libraries

BACKING STORE



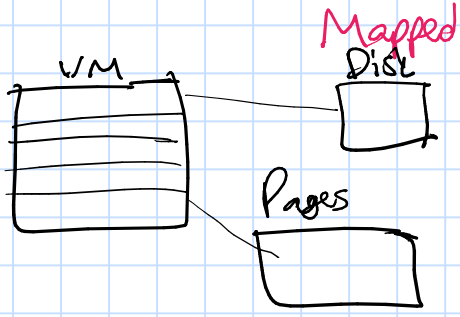
Waste of disk space (because exact copy)

- If each process keeps 4GB of menary, may disk space will be wasted.
- Menory is expensive.

Split Policy From Mechanism

Page fault Handling, it does handling but doesn't know algorithm, fault and MMU handler in kernel space.
Policy ⇒ Machine independent
For different CPU, MMU is only thing we need to change.

Mapped files



Mapped files provide an alternative model for I/O.

Paging system have paging daemon, sleeps most of the time.

OS is only responsible for setting up page table on start/fault.

Writing to disk takes a lot of time. If a frame is dirty (it will be written to disk, we can schedule to write.)

- Page fault handling is similar to interrupt handling.

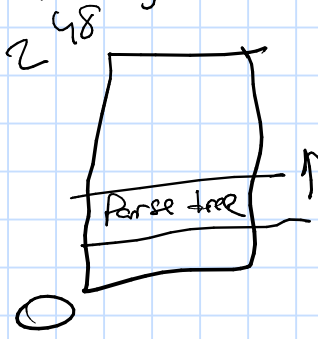
Locking Pages In Menary

If IO starts, then blocks, page that hold the buffer can be replaced.
Some pages might be locked.

⇒ Programs will try to fill the menory available to hold them

OS abstracting away details of hardware

Segmentation is not used anymore because paging is better.
↳ Gives different menory spaces, each one will start from 0.
→ Segment is a logical entity program knows.



Paging	Segmentation
→ 1 linear address space	Many
→ No need to know technique	Programmer needs to know it is used.

Pure segmentation is not implemented because of external fragmentation.
With segmentation you don't think size getting longer or smaller.

2.05.2025

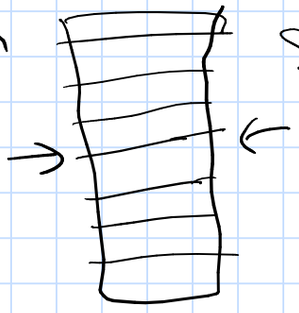
FILE SYSTEMS

Using secondary storage (better than disk used)

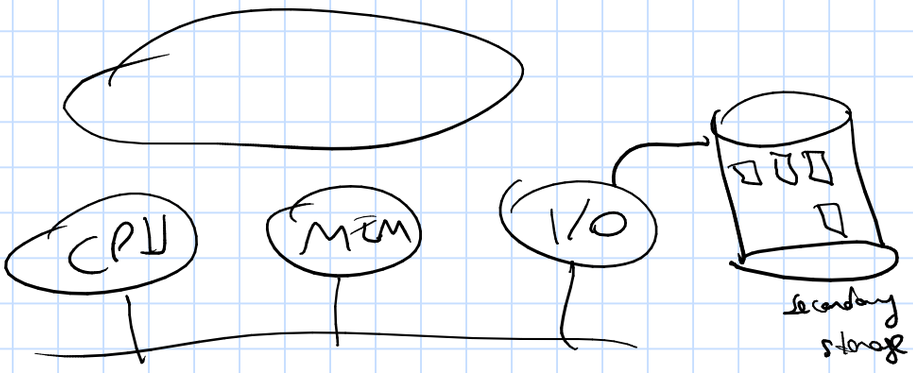
Process ⇒ Abstraction of processor

Address space ⇒ Abstraction of physical memory

File system



Similar to virtual memory (Access with block number)



In file systems, there will be access control bytes, file names, fault recovery

FAT 16 → MSDOS

FAT 32 → Win 98

NTFS → Windows NT

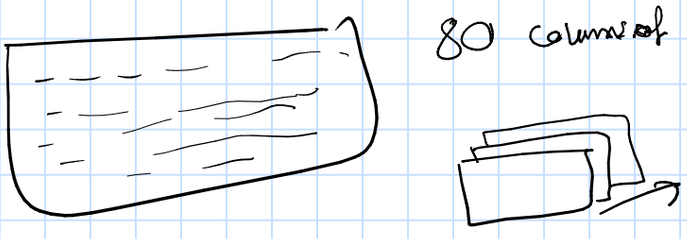
Win 8 ReFS (Resilient file system) (Robust against crashes)

EXFAT for flash drives and large file systems

ext4 ⇒ Linux advanced file system
↳ Journaling file system

Punch cards are actually file systems.

80 columns of text data.



Unix and Windows ⇒ Randomly accessible byte sequences file structure

↳ Forex record structure, byte structure, B tree structure

↳ Directory ⇒ Special file has other files

↳ Hard drive disks are still used largely in servers.

Address spaces are important.

8/05/2023

File is like memory address space, there are many of them.

Unix and Windows ⇒ Randomly accessible byte sequences
(Not page oriented)

Advantage of paging: multiple processes sharing memory
↳ Solves this problem ✗

No need paging in secondary storage. Where do i put out of page?

File types
↳ Regular files
Directories
Character special: Terminal, printer
Block files: Disks

Magic number:
In unix there is no extension.
So it cannot understand if it's executable or not.
If first 4 bytes of file is special (like 3F 37) it means it's executable. (Only in UNIX)

File Attributes (Metadata: Data about data)

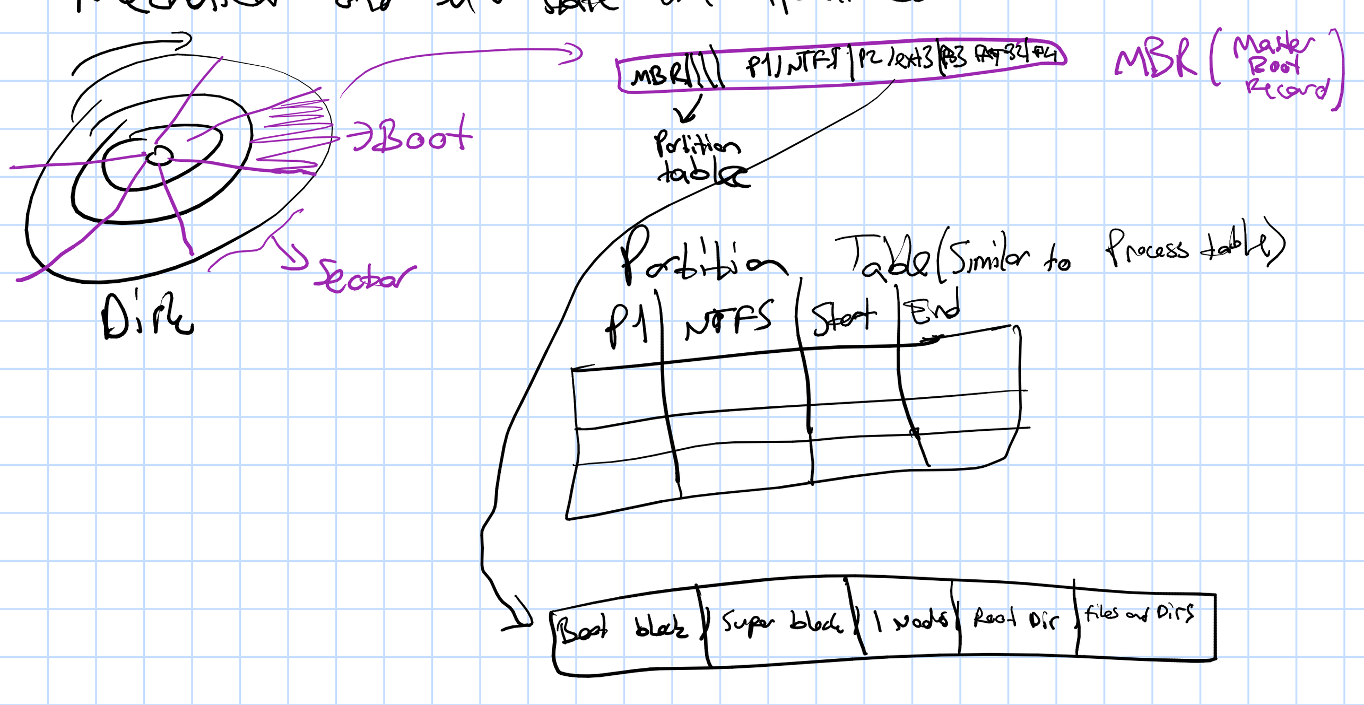
When some run file crash, they need to be cleaned up.

File Size / File Size on Disk
↳ Sector (Record) size different,

When you open file; it will load data in memory, when you read it, it will start from where you left off.

→ Hard link is not copy.

Mechanical and solid state are different solutions.



Protecting boot block and MBR is important.
MBR is responsible for scan MBR and finding active partition from table.
Superblock contains all the key parameters. OS reads it to start file system.
Contiguous allocation is perfect for CD-ROM file system. (ISO 9660)
↳ Because you can't delete.

Linked List Allocation is good for appending to file system.
Problem: If I need to reach a file, it will be slow compared to memory access. It will move head of mechanic disk. (Both sequential and random is slow)

Linked List - Table In Memory
Keep File Allocation Table (FAT) in memory.

Problem: Memory is precious, I cannot keep very big table in memory.

Ex: 200 GB disk with 1 kB blocks. Memory required?

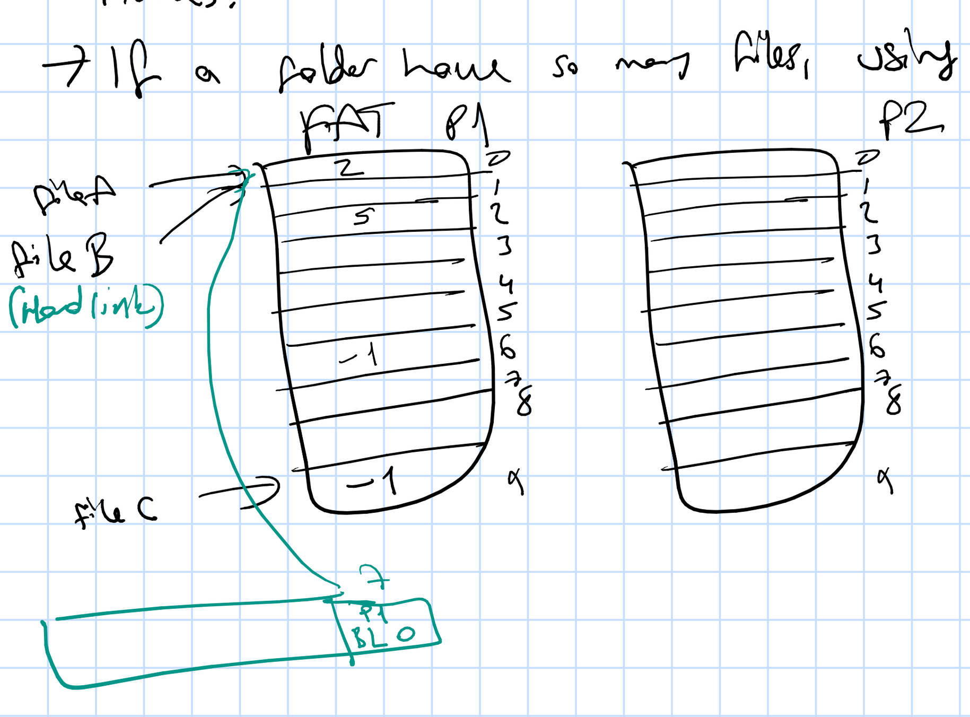
$$\frac{200 \cdot 2^{30}}{2^{10}} = 200 \cdot 2^{20} = 200 \text{ MB only for FAT to keep in memory.}$$

FAT 16 ⇒ I have 16 bits for table.
$2^{16} \cdot 2^{12} = 2^{28}$ ($\frac{1}{4}$ GB fat table)

FAT 32 ⇒ $2^{32} \cdot 2^{12} = 2^{44}$ (Good for keeping TB of data)

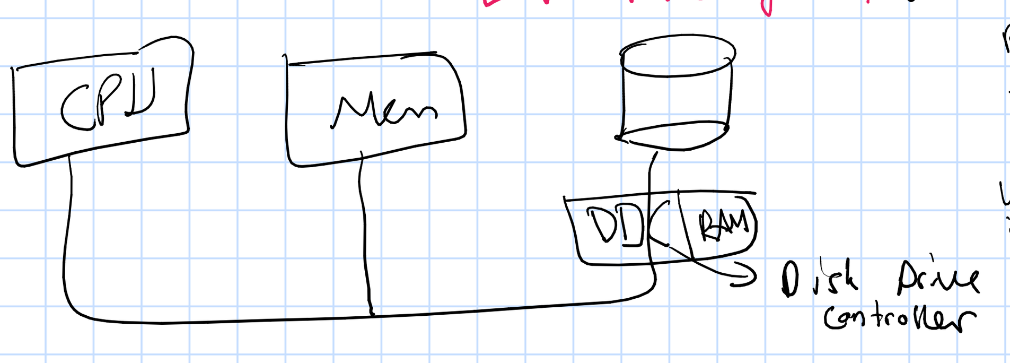
Indexed Allocation
UNIX said why keep all FAT in memory; instead only hold open file's table in memory. (Using I-Node)
When you load I-node in memory, and write it, you have to write back. (In linux you may lose data if you don't write I-Nodes)

→ Early versions of MSDOS have small limit of character in file and extension names.
→ If a folder have so many files, using hash table instead of linear search.



I Node → pointers are good for sharing stuff.

Log File Systems (LFS)



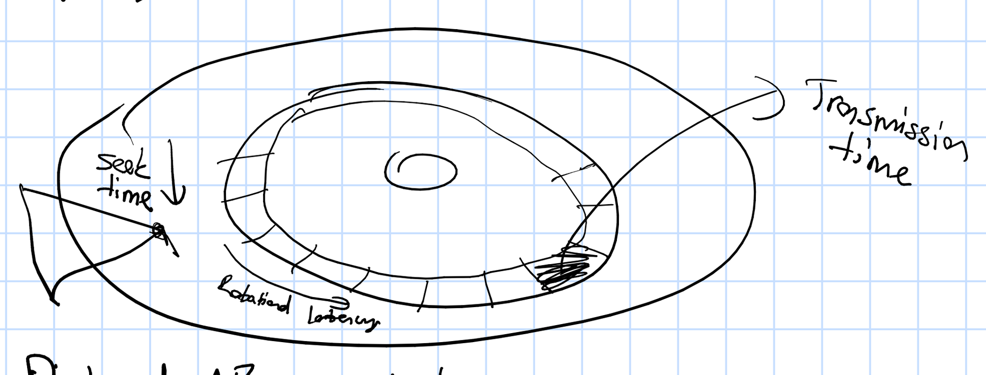
READ!
They are for efficiency, not for robustness.
Write all of modified stuff in disk as 1 segment.

Journaling File Systems (Good for recovery)
In case of crash check log and knows what to do.
MS NTFS and Linux ext3 are all journaling.

Remove Step In Linux
1- Remove file from its directory
2- Release I-Node to the pool of free i-nodes,
3- Return all disk blocks to pool of free disk blocks
↳ If system crashes, either file removed or not at all.

Virtual File System
POSIX compatible. It has 2 interfaces, POSIX and lower interface.

Disk Space Management
Block Size
If Allocation unit is too big we waste space - internal fragmentation.
Blocks are virtual. Physically there are sectors.
Seeking: Moving disk in and out.
Rotating: is not same with seeking.



Disk 1 MB per track

4.165 ⇒ Rotational time (Average)
5 ms ⇒ Average seek time
8.33 ⇒ Rotational time

$\underbrace{5 + 4.165}_{\text{Ignore for big data}} + k \cdot 2^{20} \times 8.33$

I nodes ⇒ for ex file A.

Keeping track of free blocks

Linked List Storing

Bitmap
Random access
Knowing if block is occupied or not
But when many of them are 1, losing disk space.

Fat has its own table and loads in memory. Indexes in memory.

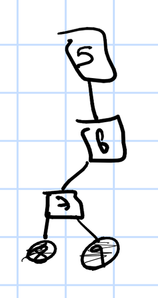
Disk Quotas
Ex: Quota is 200 MB. When file is opened, OS keeps structure of file. There is quota pointer and shows quota table.

Backup files
→ No need backup for tmp, /dev files
→ Compress during backup
→ Security problems

Physical Dump
Write all disk blocks onto the output disk in order.

Incremental Dump Backup: It stores last updated changes,

Logical Dump:
For each modified file, its inode is marked in the bitmap. (UNIX)
- If you modify content of file, directory isn't changed.
- If you add remove file, directory is changed.



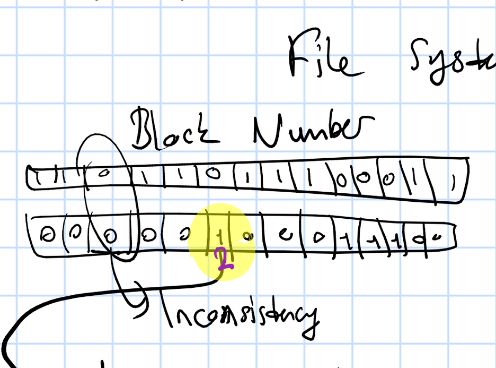
Crash ⇒ Execute bad instruction.
Physical error in disk hardware.

Typical file system crash scenario?

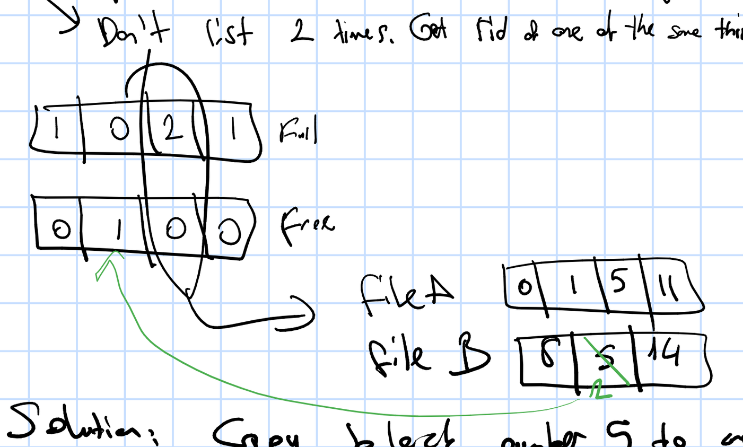
Lack file system layout?
↳ Free space mgmt, INodes should be written to disk.
Without ejecting USB, you can lose data.

→ Windows always keep everything in disk.
Unix uses disk cache.
↳ fsck, sfc → Check consistency.

File System Consistency

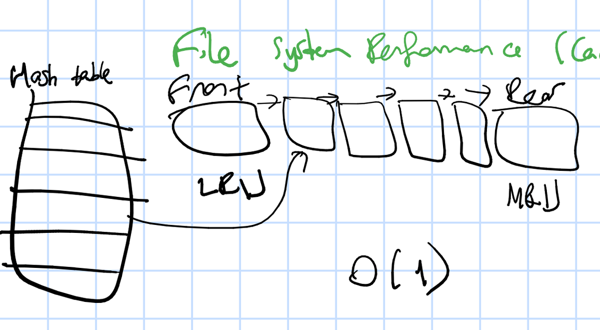


1 way: Just free it and make it 1.
→ If this one was 2, it's linked list inconsistency, not bitmap.
→ Don't list 2 times. Get rid of one of the same things.



Solution: Copy block number 5 to other available place. Don't free, it will create problems.

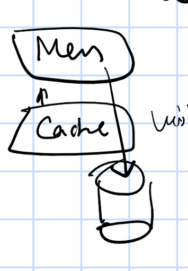
File System Performance (Cache)



O(1)

Good question:
How come LRU is not a feasible page replacement algo but it's okay for disk caches?
LRU cache to save time, extensive db search
it will store the latest info in db.
LRU remove track of page hasn't been utilized,

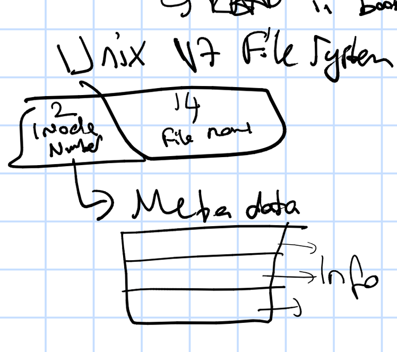
Write through cache



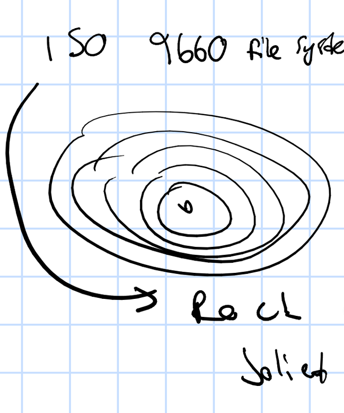
Write to cache and disk,
Reading cache is efficient.
All writes will be slow but backed in disk.

Read ahead
↳ READ in book SS

Unix V7 File System
1970s



ISO 9660 file system



CD-Rom
Single continuous spiral containing bits in linear sequence.
1 sec = 75 blocks

→ Rock ridge extensions for UNIX compatible.
Joliet for Windows NT.

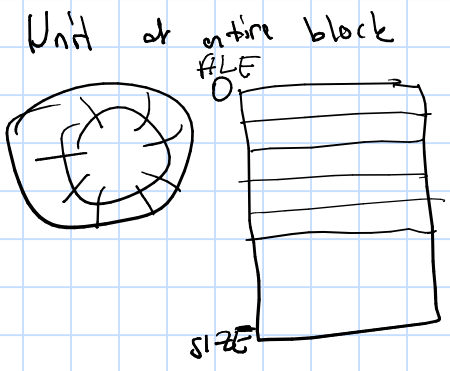
Unix and Windows ⇒ Randomly accessible byte sequences

Input / Output

- Provide abstraction such as processes, address spaces and files
- Devices are complicated. Interface should be same for all devices.
- I/O code represents significant fraction of OS.
  - For ex: GPU

Block Devices

Hard disk
SSD, blu-ray, USB stick
Store info fixed-size block
Unit of entire block



Character Devices

Deliver stream of characters
Not addressable, no seek operation
For ex: clocks doesn't fit in this schema
(nor block or character devices)

MMU is in CPU. (memory management unit)
Motherboard

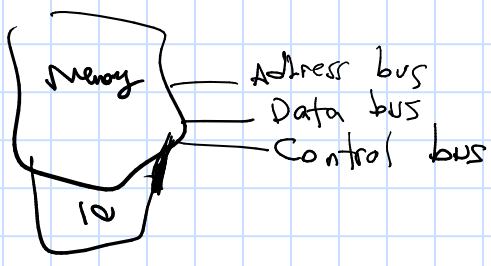
North bridge ⇒ Connect memory bus to graphic card bus
South bridge ⇒ USB — etc.

I/O Port Space

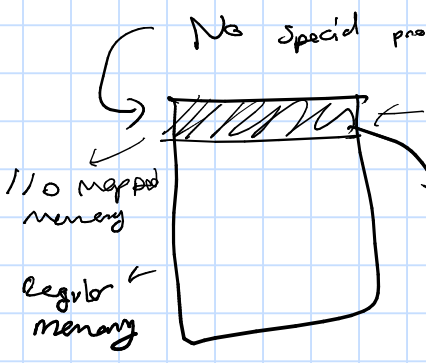
IN REG, PORT ⇒ Special instructions
IN R1, 10 Ex: MOV 3F, R1

Memory Mapped I/O ⇒ Each control register assigned to memory.

All addresses (memory and I/O go to CPU) with bus



No special protection needed to keep user processes from performing I/O.



Quiz 3
1) FAT32 data structure size?
2) How many I/Nodes for 1 file?

In C;

volatile int a; ⇒ Always get data from memory, not register. Better

register int b; ⇒ I'll access all the time so keep in CPU.
Dangerous; if b is changes, not update

DMA (Direct Memory Access)

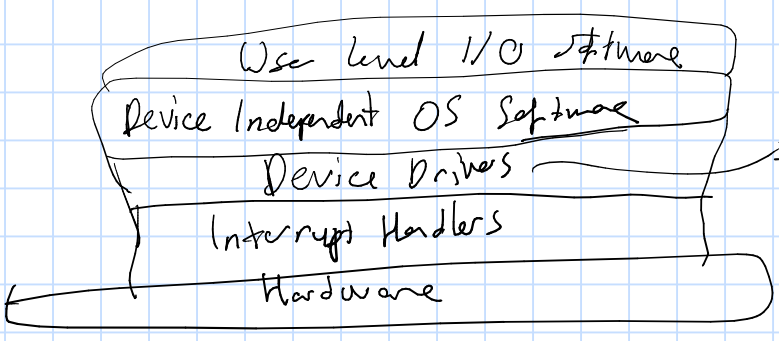
Dma controller has access to system bus independent of the CPU
has several registers

Copy data in memory address register with byte count.

Without DMA, read 1 byte, then check checksum.

DMA chip is used for any kind of I/O. Not just memory mapped I/O.
(READ).

IO Software Layers



- Interrupt handling is also expensive like context switching.
- Usually crash is because of device drivers of manufacturer.
- Prevent driver in kernel ; same OS keep outside, Ex: MINIX3 Microkernel

Keeping even device driver outside of OS.
Problem: Need to make system calls
So context switch cause slowness.

Block driver: Read as blocks; Disks, CD-ROM
Character drivers: keyboard, mouse

Double buffering: I have copies of some thing in 3 spaces. (But safer because what if new data comes.

Circular buffering:

Paging is handled by OS with help of hardware.
It is more efficient page out and keep buffer in memory space.

Study Notes

Program uses virtual address, memory uses physical address.
Translation controlled by memory management unit (MMU)

I/O Device: Device + Device Controller
Firmware; Device Driver

3 Input / Output Methods

1- When data is reding, CPU busy waiting

2- When controller finishes its job, it sends interrupt information signal, then os goes back to blocked program.

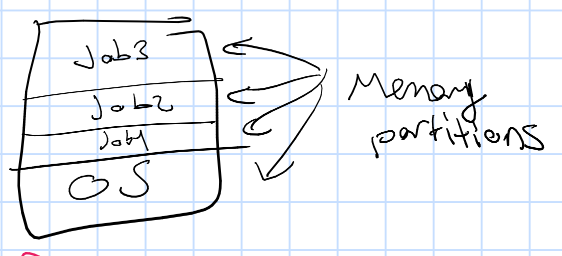
3- DMA (Direct Memory Access) chip using. This chip manages controller-memory without using CPU.
CPU only tells DMA which device, how much data. When DMA finishes its job, it creates interrupt to inform CPU.

2 Type Buses

ISA and PCI
↓ modem, sound, printer — CPU, RAM

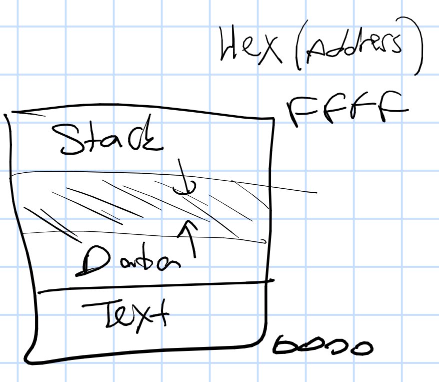
History

3rd generation OS (65-80s)

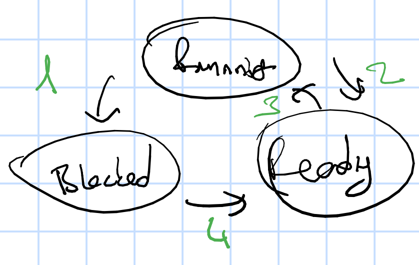


Process In Memory

Data Segment: Variables
Stack Segment: Divided by maker
Text Segment: Program code



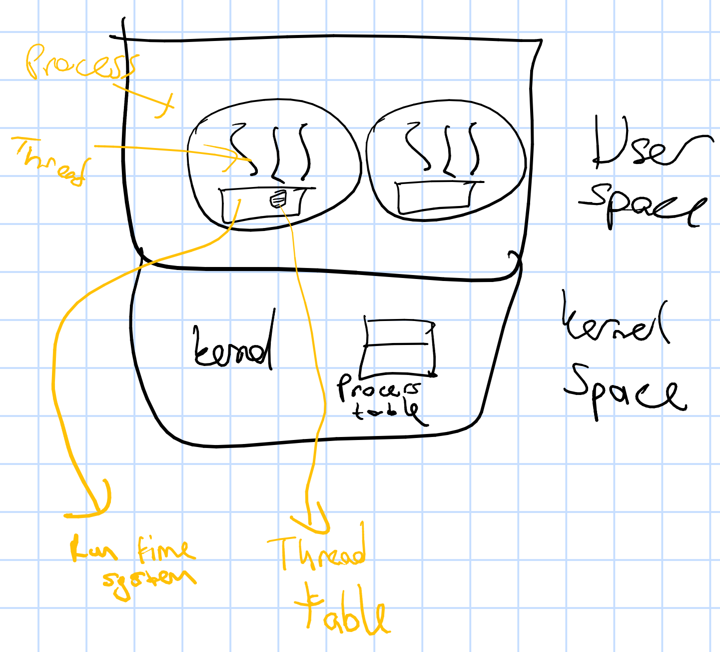
Process



1- process waits for input
2- Scheduler selects another process,
3- Timer selects this to run
4- Input is ready.

Thread

Every thread has its own stack.



Thread can be in user space or kernel space.
Kernel space threads makes scheduling of user space threads.

IPC – Interprocess Communication

When a process wants to write to a file, it writes to spooler (bekletici) info of file it wants to write.
Another process controls spooler folder. If it sees file, it prints file.
After that, it deletes file from spooler folder.

→ Race Condition Problem

Use mutual exclusion in critical section

Peterson Solution

When 2 process try to enter critical section, it uses additional variable to indicate which process was last to enter the critical section.
Preserve fairness, to prevent race condition

TSL: Test and Set lock.

Atomic and only in 1 instruction cycle.

Not: Peterson and TSL contains busy waiting.

Semaphores (Sync)

Down (P) → Enter CS
Up (V) → Exit CS

Scheduling (OS Tr Slide)

Computer has hardware clock. It sends interrupts to CPU.
2 types of scheduling algorithm with how they use clock interrupts
— Non-preemptive (kesintisiz): Bloklanıncaya ya da sırası kadar çalış.
— Preemptive (kesintili): Belli bir max süreye kadar çalışır. Hala çalışıyorsa askıya alınır (scheduler)

Batch: First Served
Shortest job first
Shortest remaining time next

Memory Management

All the time paging/swapping because we need

Multi Programming

Easiest way, partition to n parts (Fixed partitions)
But problem of empty slots not used.
To fix; put all processes to a queue.
Swapping: You can use disk as a ... (sıkıştırma)

Bitmap Memo +

2 methods for memory usage
↳ Bitmaps
↳ free list

In bitmaps; if 0 = empty
1 = full

Problem; when you put data with size k, you need to find empty place consequtively k 0's. It takes time.

Linked List memory Management

Overlay

When program doesn't fit in memory; divide program to parts called overlay.
(place shared)
0th overlay → then it calls 1st overlay ..
Overlays are in disk and swapping dynamically by OS.

Virtual Memory

Put running part of in memory, other part in disk (by OS). Swap when needed.

MMU converts virtual page to physical address.

Page in virtual mem
Page frame in physical mem } → Same size like 4KB

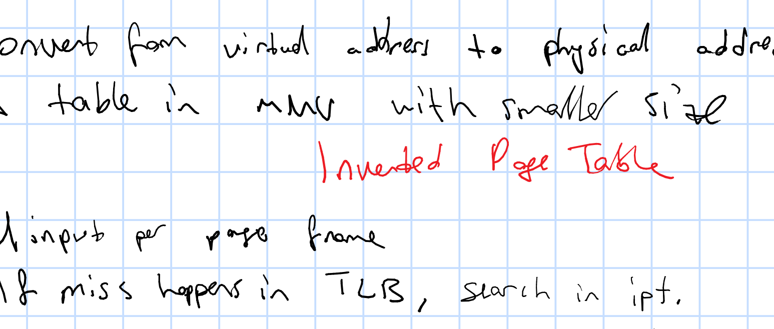
bus puts address info after conversion.

→ When MMU doesn't map, it gives warning to OS (trap) → Page fault

Multilevel Page Table ⇒ 32 bit virtual address = 10 bit PT1
+ 10 bit PT2
+ 12 bit offset

If page size 4kb ⇒ 2^{20} input in page table
PT1: ilk tablodaki 1024 inputtan her biri 4MB'lık sanal bellek
PT2: 2. seviyedeki tablonun index değeri olarak kullanılır.

Page table input



TLB

Translation lookaside buffer

Convert from virtual address to physical address without page table.
A table in MMU with smaller size

Inverted Page Table

1 input per page frame
If miss happens in TLB, search in ipt.
In ipt, to fasten search, hash table is using.

Ders 10-02 Tr READ

I/O Port Spaces or Memory Mapped IO

DMA controller has access without CPU. They share system bus.

CPU programs DMA.

Controller doesn't where data is coming from. Main memory or CPU.

Advantage of Controller

Take duty of CPU.

No need context switch.

CPU issues command one-by-one, controller do it.

- When devices issue the interrupt, we need interrupt controller:
(Because some interrupt happen at the same time, CPU can be busy)
Interrupt controller priotarizes them.
CPU programs Interrupt controller too. Connected by bus.

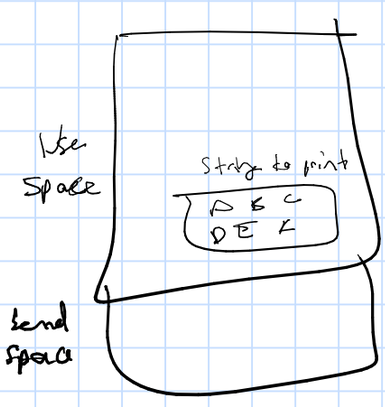
- What are the possible interrupt device program instructions on a typical CPU?
Enable / Disable, Set interrupt vector / return from, mask, unmask, query interrupt status, set priority, ack

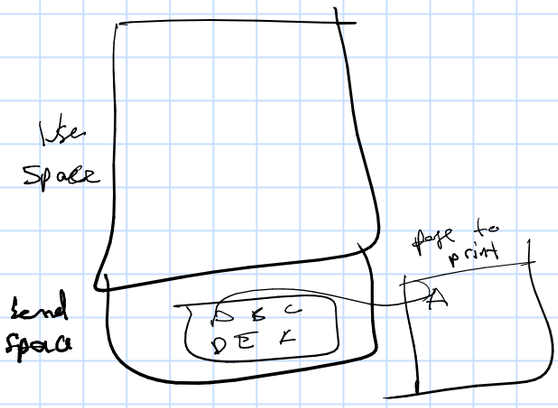
Interrupt vector: Number of the address lines is used as index to a table.
→ When the interrupt occurs, CPU jumps to spesific adress to execute handler code.

Imprecise interrupt is better than precise interrupt.

Goals of I/O Software

- Device independence: write program without access device, without know the device
- Uniform naming:
- Error handling: Handle without upper level knows error.
- Sync (blocking) and async (interrupt driven) I/O Transfers: Most physical I/O is async.
Some OS make async I/O to sync due to performance (Callback functions & non blocking calls)
- Buffering: often, data come to device cannot be stored





Polling = (Busy waiting, check condition)

Programmed I/O

Takes CPU time.

I/O Using DMA READ

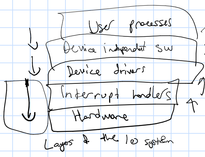
Copy-from-user

23/05/2023

Errors / Device Allocation Block Size

↳ Bad sector (I/O Errors)

We can share hard disk using time sharing or ?



Layers of the I/O system

Most of syscall are in IO

I/O sw is not always in kernel, Sometimes in user space.

Usually printf and library routine doesn't do syscalls all the time, for ex there is a loop, when buffer is full, they do system calls

Magnetic Disks

Floppy Disk

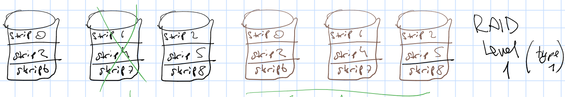
In 1 track, there are 9 sectors.
In hard disk, there are 63 sectors per track (avg)
↳ Because there is virtualization.

Floppy 360 kb — HD Disk 300 Gb → Reliable than solid state drives. Some can even stand 10 years without step. With power, can be faster.

RAID : Instead using single disk, put disk using parallelization

Redundant array of inexpensive disks

RAID level 0 type 0

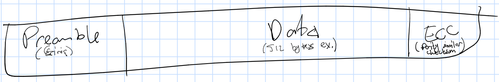


Both faster than single disk. Recover information

Sector with sync is faster than bit wise sync.

I/O 1 0 1 → ? even 1's,
1 1 1 0 → 3 1's in parity.
If anything corrupts, we can recover.

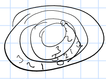
Disk Formatting



Before disk can be used, each platter must receive a low-level format done by software.

Quick format skips low-level format.

Cylinder skew is used for;
when disk rotating very fast (10000)
When we change cursor from one track to inner track it will time. (0,7 ms)



for ex; 10000 - RPM (Revolutions per Minute) drive rotates in 6 msec.

↳ Give numbers in 4 level formatting.

With low-level format; you can change formats, block size etc
↳ else ecc will reset in case of heald drives, it's good makes decision to call sector number 0,1 ...

To avoid interleaving, we need have enough buffer, controller should be able to fit all.

Most of time, seek time (move the arm to the proper cylinder) takes a lot. Usually, during low level formatting we say interleaving number.

Shortest seek first is better than first-come first-served (FCFS)

Elevator Algorithm :

You don't ignore upper or bottom level so go up, or go down.

Usually elevator algorithm is better.

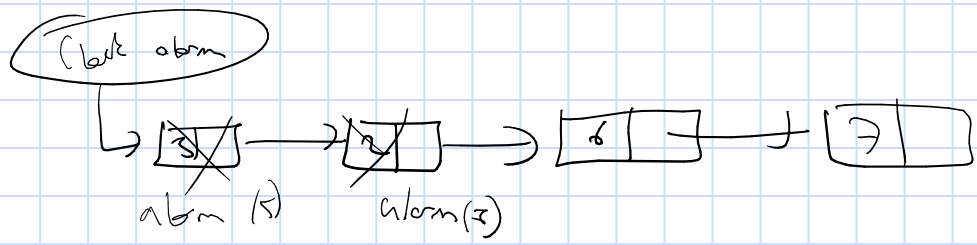
Cylinder or track is not that different.

Error Handling

Without clock we can't do scheduling in a proper way.

Clock Software

When we ~~any~~ alarm(5), we put in a linked list.



Watchdog Timers

Are frequently used (especially in embedded devices) to detect problems.
It may reset the system that stops running.

Soft Timer

Soft timers are used for the rate at which kernel entries are made for other reasons:
syscalls, TLB misses, page faults, I/O interrup, CPU going idle

Keyboard Interface

It contains an embedded microprocessor communicates through a specialized serial port.
Different give seperate interrupts.
Shift A
Shift B

Mouse Software

It generates an interrupt when clicked/released or moves.
Contains 3 items: Δx, Δy, buttons
Max of 40 times/second

Display Driver

The X Window System

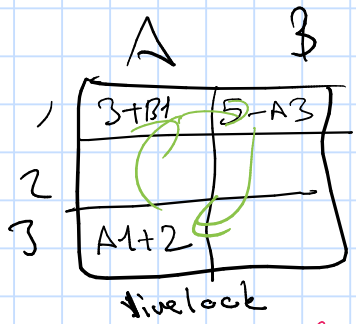
X window Server
UNIX
Hardware

Server sw

UNIX keeps GUI out of kernel,
Windows keeps inside.

DEADLOCKS (CHAPTER 6)

It's difficult to find deadlock.
Once a year deadlock (ignore to prevent) is easier. Since using so much CPU is also bad.



We prefer convenience than correctness.

Preemption (Ethernet, online seating)

Force release the CPU.
Risk of deadlock.

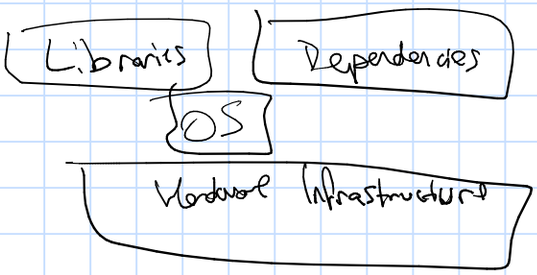
Resource Acquisition

Using 2 semaphores for 2 process, and wrong semaphore locking order.

Deadlock Condition

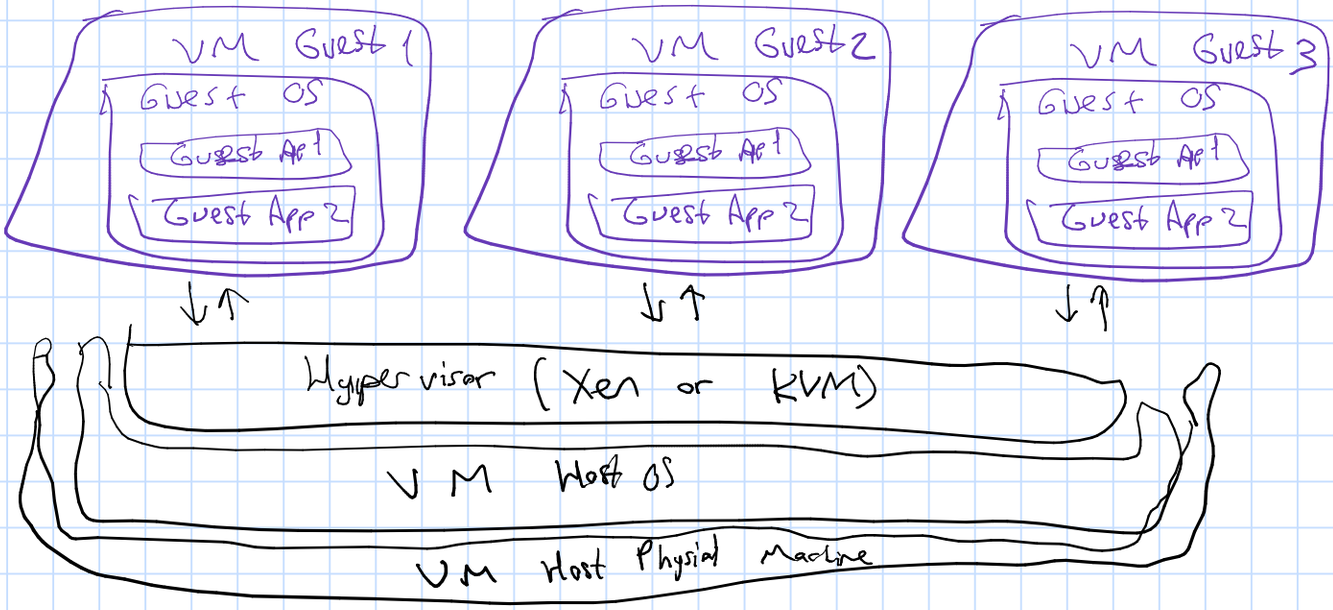
Virtualization

Bare Metal Server



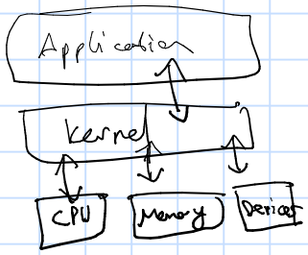
We upgrade app, but OS is not compatible so upgrade OS, but then web server is not compatible. So upgrade server.

In bare model, we have isolation problem.
In other model, there is isolation. Different services for web server, frontend, db, cache, monitoring.



Downtime => Service or app crash
Bare model => Use redundant virtual machines.

Linux Kernel



Control Groups: To control the

When there are many VM's we have hypervisor.

Docker

Open source platform to manage containers.
Docker image can contain OS, codes etc.
Docker hub contains many docker images.
Docker container is like running like a process

Process isolation / Container isolation

- Different containers can get in contact with each other.

5/06/2023

Resource

Anything must be acquired, used and released.

If we can preempt everything, deadlocks would not occur.

Some resources preemptable. (Printer...)

CPU is preemptable, it's good.

Deadlock

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Windows Linux use a lot others ostrich algo approach

Spool everything

Request all resources initially

Take resources away

Order resources numerically

If no parallelism, no deadlock.

Deadlock Avoidance

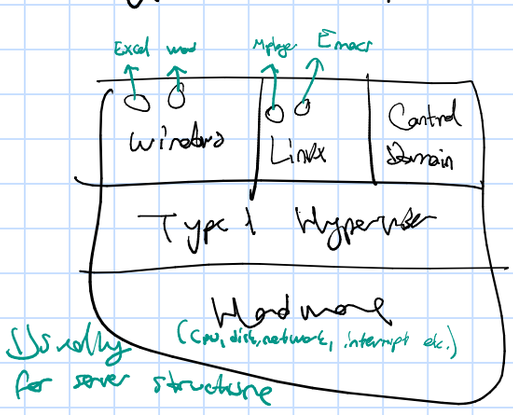
Find regions deadlock might occur. Detect resource trajectory. (Expensive)

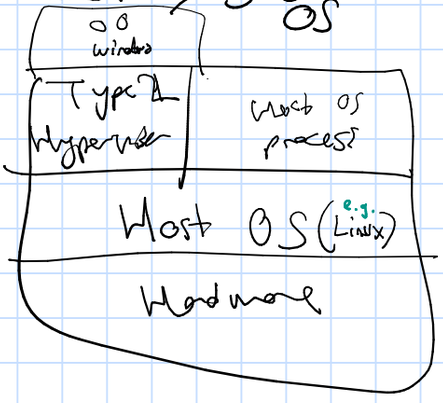
Starvation

Not Same with Deadlock

OS106
VIRTUALIZATION AND CLOUD

Hypervisor supervise the OSs. → Guest OS





Hypervisor is simpler than OS.
So chances of bugs in hypervisor is less.

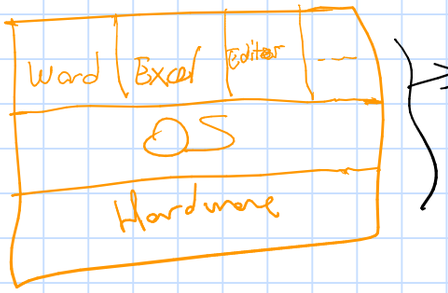
Migrating virtual machine is easier.

Requirements for Virtualization

Safety: Hypervisor should take whole control.

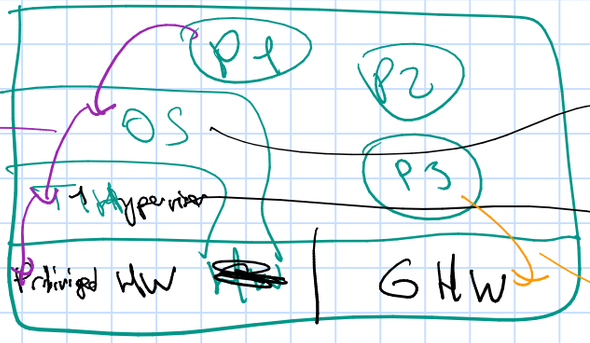
Fidelity: It should run like its original OS without virtualization.

Efficiency:



We don't like this system since OS is a privileged process

(Better one)



Expensive path. most of time, doesn't occur.

This OS cannot run well if wants. Since there is P1 hypervisor.

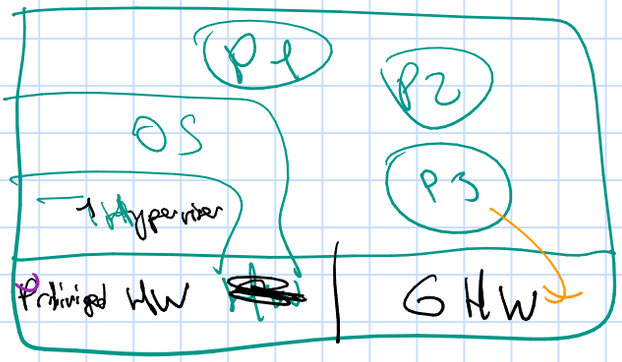
Setting up Page table, interrupt table.

Usually use this one

Switch emulator is a virtual system also.

Running some apps on virtual machine is faster than real.

VIRTUALIZE THE UNVIRTUALIZABLE



PARAVIRTUALIZATION

doesn't aim present a VM working that looks just like actual underlying hardware.

It offer set of hypercalls, which allow the guest to send explicit requests to hypervisor.

Guests use hypercalls for privilized sensitive operations like updating page table, bc they do it explicitly in cooperation with hypervisor. (So it can be faster.)

TYPE 2 HYPERVISORS

Run guest code natively and use exactly same techniques requires the type 2 hypervisor to manipulate the hardware at lowest level.

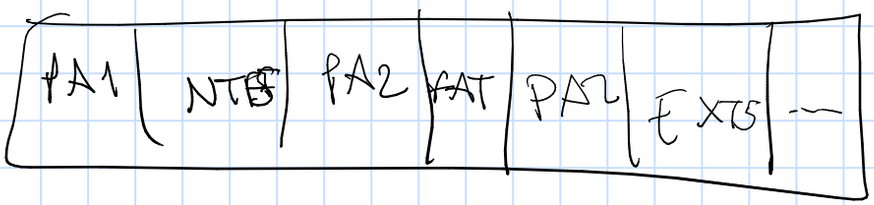
- So most modern type 2 hv have a kernel module operating in ring 0 that allows them to manipulate the hv with privileged instructions.

→ Trap is expensive since needs context switch,

Difference between Old Hardware and New Hardware

↳ when trap happens we need to invalidate a lot of data, stop populating. (On modern HW, its really expensive)

I/O Virtualization



Modern IOO support I/O remapping.