ID1206 Operating Systems

3 tasks of an operating system:

Abstraction: Create a layer of abstraction between the hardware and the application.

Virtualization: Create image for each process, so that it thinks it has the whole cpu & memory while in reality it shares those with other processes.

Resourcemanagement: How do we share limited recourses in a fair way.

Kernel: The program that interracts with the hardwore.

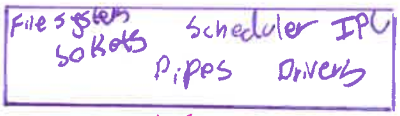
POSIX

Portable OS Interface.

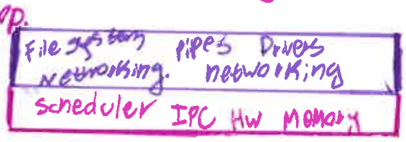
Includes:
- API: fork, exec, wait....
- Process communication: pipes..
- Threads: pthread_creates...
- File system
- Networking: sockets.

Monolithic VS Microkernel

A monolithic kenel is a very lage system of of all parts of the os with an equal hierachy. Of all processes.



A micro kenel is the minimal tools needed and everything else is built ontop.



Microkernels are more robust and stable since errors can be handled by the kenel. But they are far slower since everything needs to go through the kenel.

Standard C (ISO C18)

Memory allocation: Malloc, free...
Signal handeling: Signal, raise, kill...
file operations: fopen, fclose, fread....

Commandline Interpreter

Shell: text based
Scripting langeuges:..

C process

a c process has the following:
- a program
- a stack
- a heap
- program counter
- open file descrip

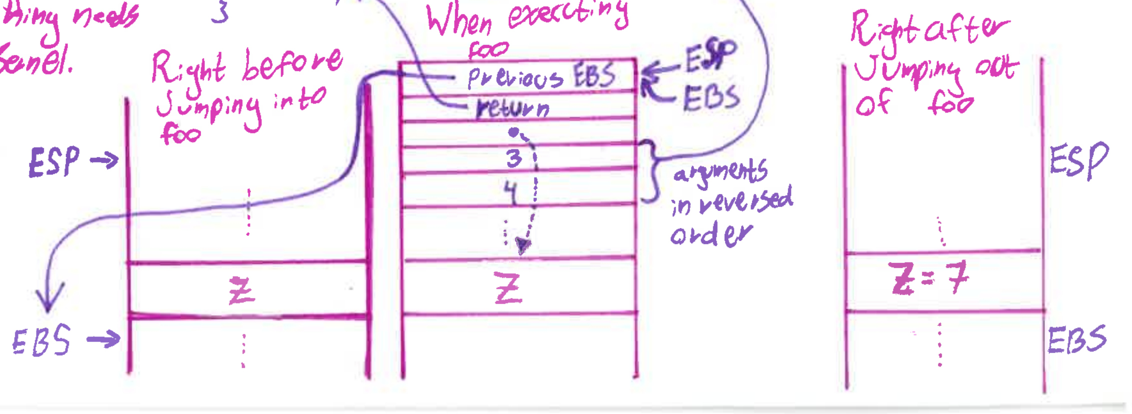
The Stack

A stack frame contains, arguments, local variables, info to be able to exit the frame

Pointers in CPU:
- instruction pointer (EIP) next instruction to execute
- Stack pointer (ESP) top of the stack
- base pointer (EBS) current stack frame.

```
int bar(){
  int z;
  z= foo(3,4);
  return z;
}

int foo(x, y){
  return x + y;
}
```



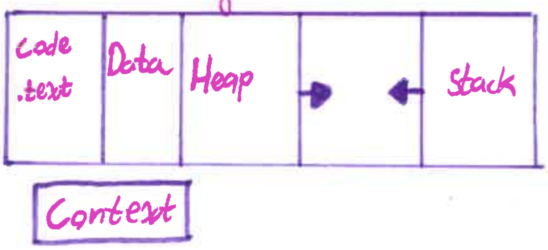
Returning Datastructures

```
int* foo(int x){
    int a[5]={1,2,3,4,5}
    return a;
}
```

Calling this function will return a segmentation fault. This is because a is allocated within foo's stackframe and thus its location is cleared after foo is exited. instead the heap should be used.

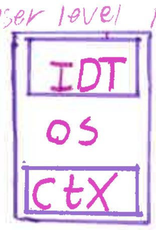
Program to Prossess

1) find program in presistent storage.
2) Allocate to hold the code Statics, heap & stack.
3) Process context to hold neccessary values.



Interrupt descriptor table

Used by Kenel to execute Kenel level instruction invoked by user level processes.



Protected area of the memory.

These calls are expensive since everything needs to be saved temporarely.

Fork

Is used to clone a prossess with a deep copy of the memory (they do not share variables)

```
int pid=fork();
```

It returns 0 once for the child and a new process id for the parent process.

Heap API

void *malloc (size_t size): Allocate size bytes of data on the heap, returns a pointer to the structure.

free(void *pointer): Deallocates the structure at ptr

void *calloc(size_t, nmeb, size_t size) Same as malloc but fills the area with nmeb.

void *realloc(void* ptr, size_t size) Changes size of exesting malloc.

Indirect Execution

The OS decodes the instructions and runs them like a virtual machine. Java, Python, Erlang.

Direct Execution

The code is directly executed on the cpu OS gives up control. Dangerous!!!

Limited Direct Exection

with the use of hardware support code is directly executed with some limitations.

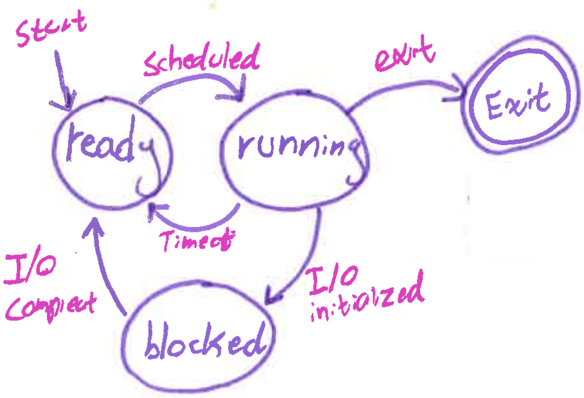
- Will not be able to execute all types of instructions.
- Can only access its own memory
- has limited cpu time.

Interrupts

These tell the Kenel to do something based on a trigger, eg divide by 0, keystrokes, exceptions.

They can also be "Asynchronus, raised by hardware (not cpu) or Ascynchronus, raised by cpu

Process State



Many processes can be blocked and ready at the same time but running is limited by the cpu.

Fork commands

exit(); terminates process

wait(); wait for other process to terminate.

exec(); load different program and start exec.

getpid(); gets pid.

kill(); send signal to process

raise(); raise exeption

sigaction(); sets signal handeler.

Preformance Metrics

$T_{turnaround} = T_{Completion} - T_{arrival}$

$T_{response} = T_{First\ Scheduled} - T_{arrival}$

MLFQ

round robin with priority
Priority changes dynamically

Rules:

1) if priority(A) > priority(B) Pick A
2) if priority(A)=priority(B) round robin
3) New Jobs have higest priority.
4) if Job does NOT use an I/O operation priority decreases by 1
5) After some time of low priority the process is set to the higest priority again.

Stride Scheduling

Jobs are given a value inverse to their priority. Every time the Job is executed this value (called stride) is added to a value associated with the process. The process with the lowest value goes first.

Real Life

uses Stride Scheduler and keep Jobs on the same core to prevent cache misses.

Scheduling Strategies

Shortest Job First (SJF)
Shortest time to compleation (STCF) * optimal turnaround.
Round Robbin (RR)
Multi-level Feedback Queve (MLFQ)
Lottery
Stand in line
Stride Scheduling used in linux

Realtime Scheduling

Adds a new requirement that Jobs should be compleated before a deadline.

Hard: All deadlines MUST be met, missing a deadline is a failure.

Soft: Dedlines can be missed but the application must be notified

Tasks are described by a triple (e, d, p)

e: worst case exec time
d: deadline of task
p: how often task should be scheduled.

d<p: Constrained
d=p: default
d>p: Several out-standing

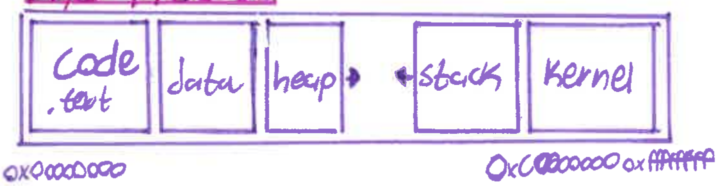
Rate Monotonic Scheduling (RMS)

assumes default (d=p)
Rule: Schedule task with shortest period.
works always when utilization ($\sum^{n} \frac{d}{p}$) is less than 69%
but in general when utilization is less than $n(2^{1/n}-1)$ n = number of Jobs

Earliest Deadline First (EDF)

Based dynamically on deadline
works always if utilization is less than 100%

The Process



32 bit Linux

However the process does not actually operate on address 0x00000000-0xFFFFFFFF in memory. instead a so called memory management unit (mmu) translates between physical and virtual memory addresses.

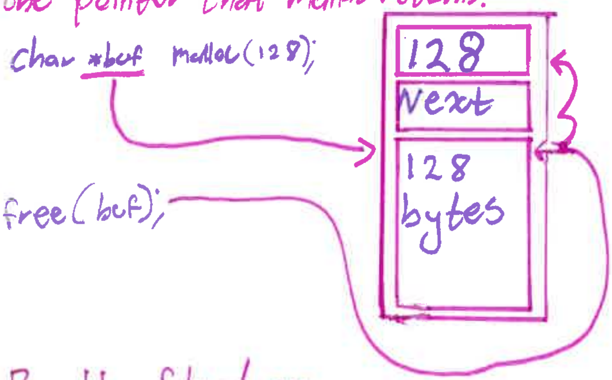
Brk() & Sbrk()

Change the end of a process heap segment. brk() sets the value to the arg and sbrk() increases current size by the argument. sbrk returns the new address.

these are syscalls while malloc and free are library routines.

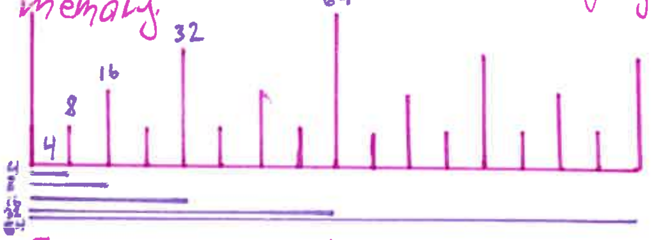
Malloc & Free

In order to know how big an area is this is stored two addresses above the pointer that malloc returns.



Buddy Strategy

Used in linux for allocation of physical memory.



Each memory segment can recursively be split into buddies, each of ½ the size. they are split until a region fits a node as closely as possible

very efficient

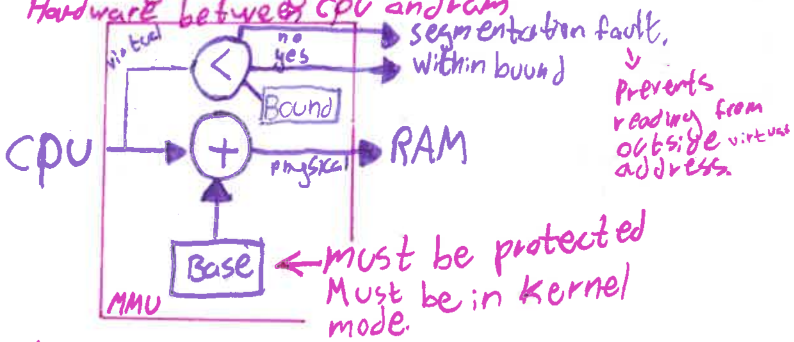
coalescing eficient (lg(n))

handles fragmentation very well.

internal fragmentation. if 65 bytes is needed 128 is given.

Memory Management Unit

Hardware between cpu and ram



the con is that some memory like libraries must be copied to each process and shared memory is not usable. Problems with fragmentation.

Processes are always assigned 4GB even if only a few bytes are needed.

Transparent: Processes are unaware of virtual addresses.

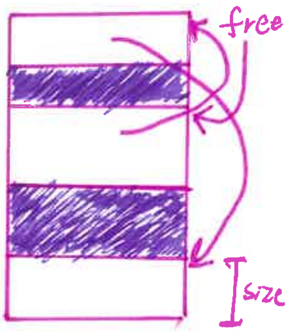
Protection: Processes cannot interfer with each other

Efficiency: Should be as close to real execution as possible.

Free list

used to find free space in memory

```
typedef struct __node_t {
    int size;
    struct __node_t *next
}
```



When blocks are freed they are added to the start of the list.

when consecutive blocks are freed they must merge (coalesce)

Free list strategies

If i need to malloc 40 bytes how do i know how big or which block i malloc?

- Best fit: smallest possible block
- Worst fit: biggest block
- First fit: first possible block

Segregated lists

have separate list of blocks of predefined sizes. (8,16,32,64...) very fast at finding blocks of aproximate size.

mmap

Default use in OSX instead of sbrk and brk

```
Void *mmap(
    Void * addr,
    Size_t len,
    int    prot
    int    flags
    int    fd
    Off_t  offset
);
```

Creates virtual mapping

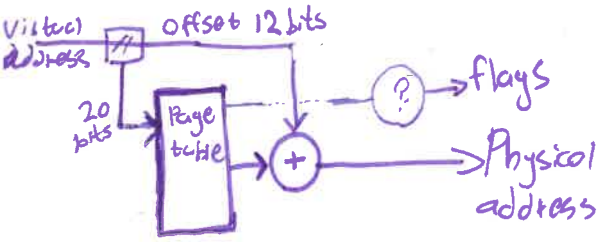
addr=NULL ⇒ dont care

prot= protection (allows shared memory)

flags, fd, offset allows mapping to a file in memory.

Paging MMU

Memory is split into pages of equal size. Usually pages are 12 bits large of address space.



overflow can never happen since a page is exactly 2^{12} bits large.

Segmentation and paging is often combined.

Lookup procedure:

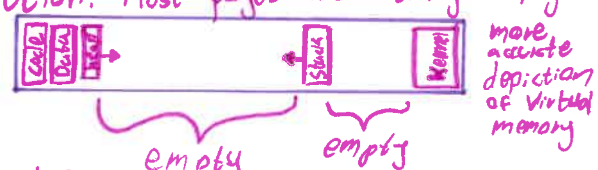
Movl 0x11111222, %eax
virtual address

1) Page table base register PTBR
2) virtual page number VPN 0x11111 (20 bits)
3) read from page table PTBR+(VPN<<3)
4) extract Page frame number from table
5) add offset to page frame number.
6) read memory at this location (PFN<<12)+0x222

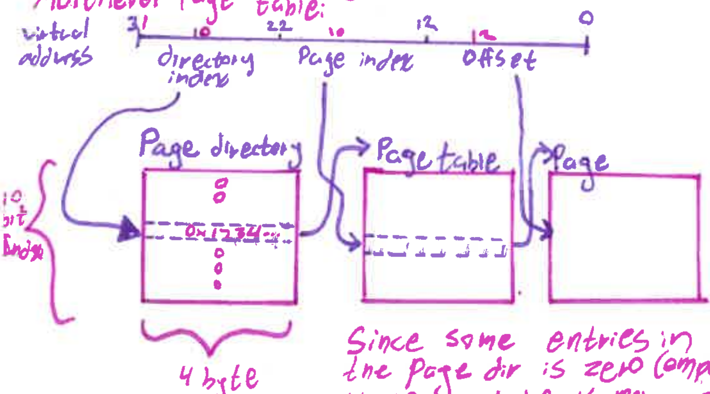
This is too slow since each lookup requires two memory accesses.

The drawback is that each entry is 4 bytes
Each page table is 4 Mibyte
Each process has its own page table
thus for 100 proceses 400 Mibytes of tables is needed (too much).

Solution: Most pages are actually empty.



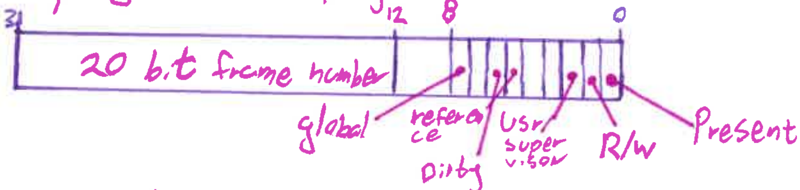
Multilevel Page table:



Since some entries in the page dir is zero (empty) there wont be as many page-tables needed assuming there are gaps.

Page Table

Each entry is a map between a virtual and physical address. Or rather a page in the physical address table.



Physical address extention

in 1995 the x86 extended the frame numbers to 24 bits. thus 64 GB of Physical address could be addressed. Virtual process still dont know abot the extention and still thinks its limited to 4GB.

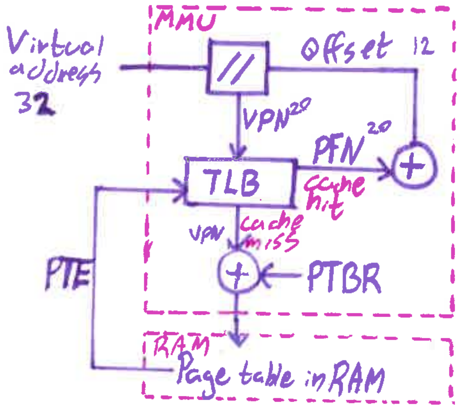
x86-64 has 48 bit virtual address 0x7ffff...

Speedup using hardware

A copy of each lookup is stored in the cpu cache so sequential lookups are giga fast.

This is called TLB (translation lookaside buffer)

Non-global entries need to be flushed between usages when switching active processes.



Inverted Page lookup:

If we have 8 Gibyte RAM then it'll fit 2Mi frames of 4Kibyte then a table with 2Mi entries can map a process to its frames

index	page num	Process id
0	2	1
1	.	2
2	.	3
.	.	4
.	8	4
.	.	8
.	.	.

When you find your entry the index of the table is the actual page number in physical Ram.

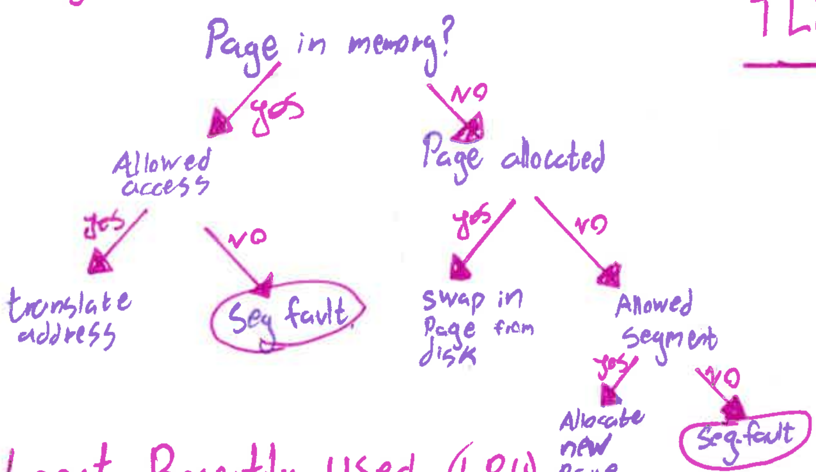
Swapping

Benefits:

1) Give illusion of private memory
2) give illusion of much larger address space.

When the main memory is filled up the unused processes have their pages moved out to external memory i.e. hardrive. later when paged the OS will fetch the page from the harddisc

Page lookup process



Least Recently Used (LRU)

used to decide what page to swap when RAM is full. The idea is that a page that was not used recently is less likely to be used again.
It is expensive to keep track of which pages have been used when there are 1000+ pages.

FIFO

First in first out, approximation of LRU but the que is only updated on misses. much cheaper algorithm since the que is only updated when the processor is already taking time to find page in presistent memory.

Clock

Uses hardware and is another approximation of LRU. A buffer of all pages is created. When a page is accessed and it is in memory it's usage bit is set to 1. If there is a miss a pointer goes through the buffer if the pointer points at a 1 it is set to 0 and pointer is incremented, if it points to a zero, the current page is swapped and pointer is incremented.

Page Table Entry Pt. 2

As shown on the previous page there is a flag called "present" that indicates if the MMU has an entry for that page. But if it is 0 there will be 31 remaining bits that can be used to find the page on the HDD.

Cost of page faults

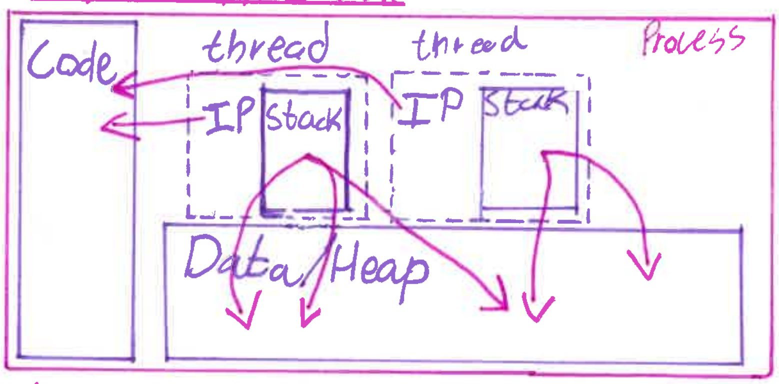
TLB hit — address in cache: 1 ns; mem: 10 ns

TLB miss — page in mem: 100 ns; Disk: 10 ms = 10^7 ns

Concurrency & Parallelism

The illusion of things happening at the same time is concurrency.
Parallelism is actually doing things at the same time.

The Thread



Threads have a shared memory including heap & stack, GCC however may optimize the code, so use the label volatile on shared variables.

Threads can be scheduled by either the prosess or by the OS in GNU Linux everything works in kernel space.

`int pthread_create(...);` Create thread

`__thread int local = 4;` Create local variable to a thread.

Mutual Exclusion

First try using __sync_val_compare_and_swap();

```
While(try(lock) !=0){
    ;
}
```

Loops forever until lock is free. Scheduler does not know the process is just wasting time.

Second try using yield();

```
While(try(lock) !=0) {
    sched_yield();
}
```

loops once every time it is scheduled. Still bad but much better.

third try using futex/park;

```
While (try(lock) !=0) {
    futex_wait(lock, 1);
}
Void unlock(volatile int *lock){
    *lock=0;
    futex_wake(lock);
}
```

Kinda cool but takes use of syscalls and thus takes some time.

Best try, use pthread_mutex

pthread_mutex_t
pthread_mutex_init
——||——_destroy
——||——_lock
——||——_unlock

Possible Problems

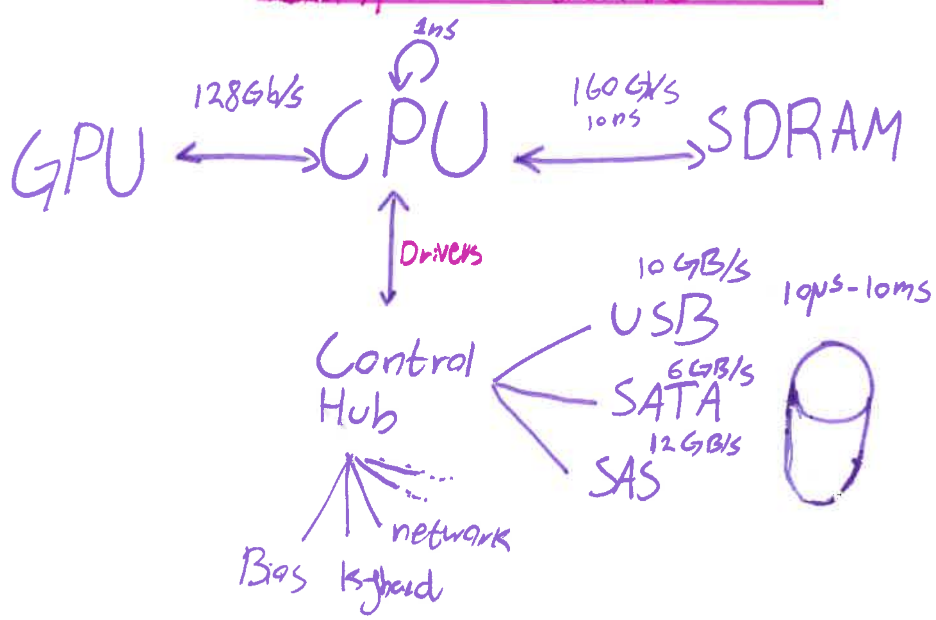
Deadlock. Threads are circularly dependent on each other. Stuck with no progress.

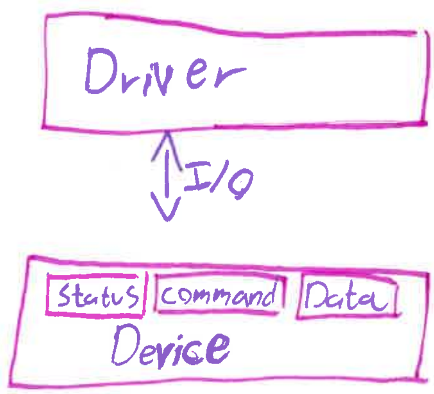
Livelock. Threads thinks they are doing stuff but is actually stuck in a loop.

Starvation some threads are making progress but one is neglected

Unfairness some threads gets more resources unfairly.

Computer Architecture

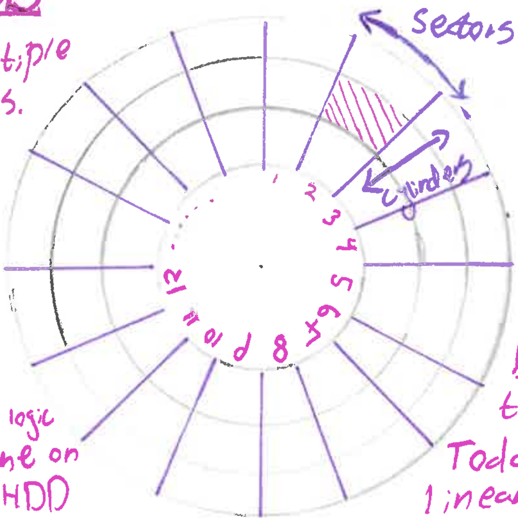




Process does I/O request, Suspends and blocks itself. When I/O is done it wakes process via interrupts.

HDD

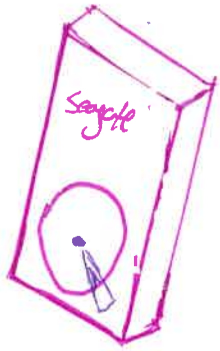
Multiple Disks. each with two sides.



Smallest unit: 4K

In the Past sectors were addressed by disk, cylinders, sector but this limited storage to 500MB. Today it is all done with linear addresses.

Most logic is done on the HDD



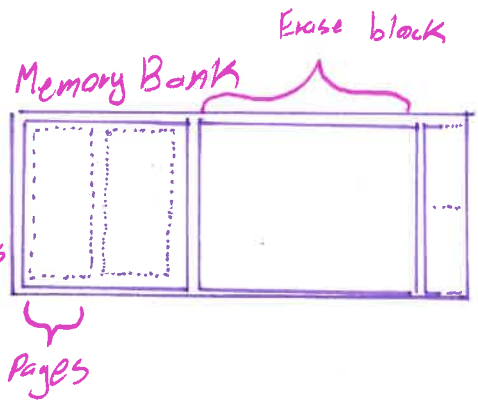
2 TiB
3.5"
7200 RPM
SATA III
Cache 64 MB
read 156 MB/s

0.44 kr/GB

SSD

500 GiB
2.5"
rA 10μs
read 560MB/s
SATA III

2 kr/GB



Only a whole erase block can be erased. Thus changes cannot be made to individual Pages. Thus to edit the Hardware moves a page instead of editing.

SSD are cool and can be but on every bus.

BUS limitation

SATA III 6 Gb/s
SAS-3 12 Gb/s
USB3.1 10 Gb/s
PCI Ex. 3.0x16 128 Gb/s

RAID

Redundant array of independent Disks. Multiple disks seen as one, also has error corrections.

Raid 0:
Stripe file across multiple disks.

Raid 1:
Keep a mirror copy of each file.

Raid 2-6:
Spread a file + parity info across several Drives.

Files

-Presistent, shared, path.

A file is:
- sequence of bytes.
- metadata
  - ↳ size & type etc.
  - ↳ owner & premissions.
  - ↳ author
  - ↳ created, changed
  - ↳ (icons, fonts)

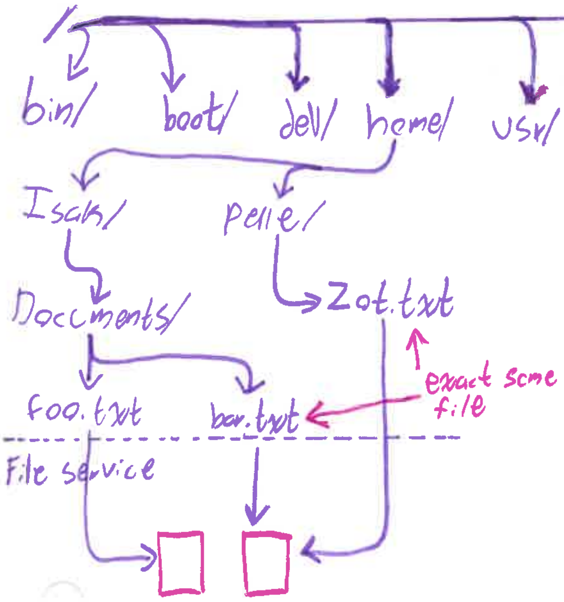
functions:
- create & delete
- finding
- read write
- control authorization.

Directory } mapping

File module
Access control } file structure
File operations

Block operation } Drivers
Device operation.

Tree File Structure



OS Functions

Creat - create file

Unlink - remove link, when last link is gone remove file

link - Hard link

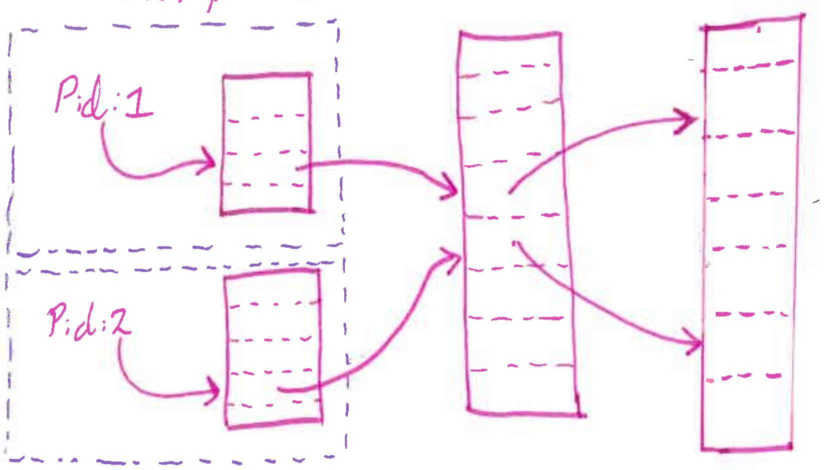
Symlink - soft link

Stat - read metadata

The kernel keeps a record of each process accessing a file. By default a process has 3 open files, standard input, standard output, standard error.

File Tables

One table per process, copied when process is forked.



File Descriptor table. list of files used by process.

Open file Table keeps track of open(); calls. has a counter for when 2+ processes access a file

Inode Table One Inode entry for each file.

File Operators

Read - reades bytes into buffer

Write - writes bytes from buffer

Seek - reposition write head.

The Inode

Includes:

- Mode (access rights)
- # links
- User Id (owner)
- group Id
- Size
- blocks
- identifiers
- generation.
- access time
- modify time
- change time.

Total 15 total Pointers.

first 13 Pointers are normal block pointers

Pointer 14 is a to a block of pointers.

Pointer 15 is a Pointer to a block of blocks of Pointers.

The File system

The first few blocks of a HDD is metadata that tell the OS about the HDD.

- Describes file system
  - How many Inodes and where
  - How many data blocks and where
  - where bitmaps
- bitmap for which inode is taken
- bitmap for which datablocks are needed.

(remaining blocks)

Directories are Inodes with a single block.

File System errors

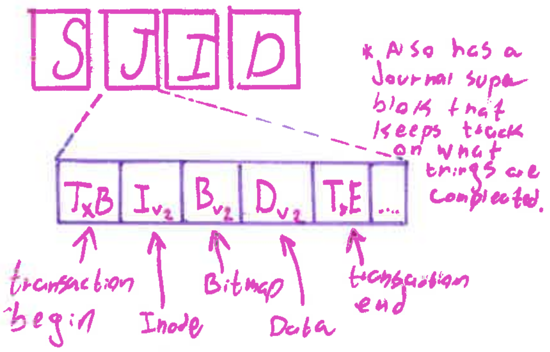
When a file is changed 3 changes must be made.

- update bitmap
- update Inode
- Update Block

Crashing before doing 3/3 tasks is very bad.

Journal

Keep track of what we are going to do. Add a new Journal Block.



Either All are compleate or none are compleate
These can be queued before many are done at once.

Approaches:

file system check: flag that tells file system that it did not shut down properly prompting a compleate file check.

Journal: write down what you want to do before you do it.

log: log all changes.

copy on write: creat a perfect copy and last thing you do is flip a pointer.

Log Structures

Write sequentially since reads are cached in memory write speed is the most important

Everything is written left to Right. Inodes point to data blocks before them. Map Point to which Inodes are most up to date. Check region points at the latest Map.
This way the System can fail before CR is updated and it will simply be reverted to a previous state. When full the first blocks in the list will be copied to the end, clustering them and frees consecutive space.

Virtualization

The OS is run with a so called Hypervisor between the os and the Hardware. It mimics the interrupt table and acts as a man in the middle. This is done in order to emulate different hardware eg. Arm on a x86 machine, or to have different versions on an os before everything is migrated/updated.