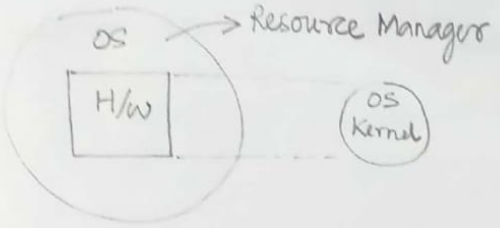
1-08-2017
Tuesday

Operating Systems

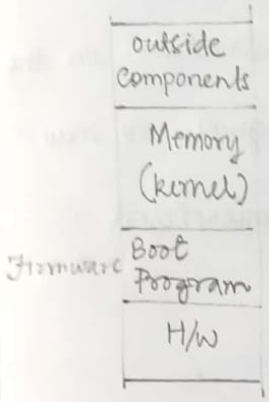


memory requirements

Datacenter OS: TB

Desktop OS: 4GB

Tiny OS: 4KB



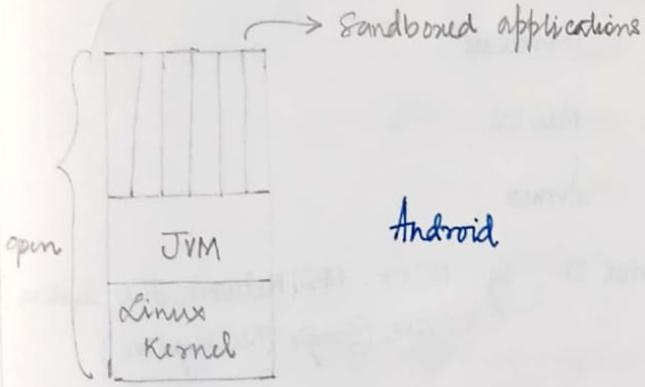
BIOS · Basic I/O

When computer boots up, the boot program executes and loads up the kernel into the memory resident part. That upon execution transfers control to user.

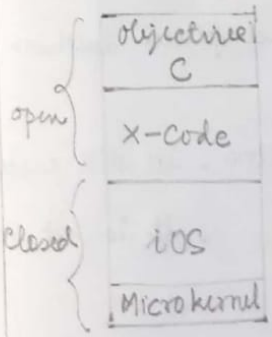
DOS: Disk Operating System

Initial floppy drives : 320 kb.

Mobile OS



Android



iOS.

2-08-2017
Wednesday

Notion of separate compilation

f1.c f2.c f3.c

gcc

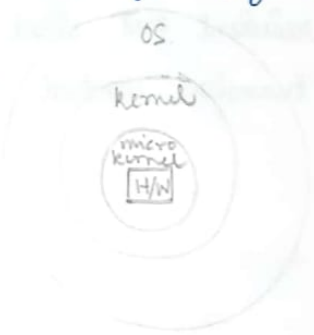
f1.o f2.o f3.o

linker

.o file : no unresolved symbols.

objdump -D f1.o ⟶ C file with all symbols resolved

A microkernel is the first layer of abstraction on the architecture. (i) It makes it easier to migrate to new architecture by simply modifying the microkernel.



micro-controller : Tiny OS.

Mobiles → iOS, Symbian, Android

Desktop Systems → Windows, Mac OS, Linux

Datacenters: Distributed OS. Eg: DC++, NFS (Network File System), GFS (Google File System)

3-08-2017
Thursday

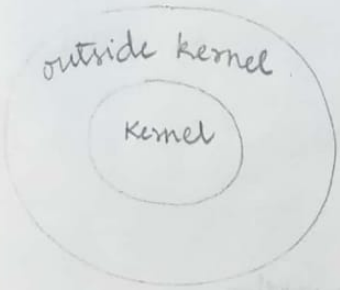
RTOS : Real time OS - Tasks must be completed within deadline.

Soft Real Time :- If task misses the deadline, it still executes.

Hard Real Time :- , it is not executed.
Priority is reduced to 0.

(ii) • 2nd reason of using micro-kernel is that it can be used to verify the kernel. Eg seL4 is a verified kernel.

OS-Design



20% kernel

80% device drivers.

OS: Resource Manager

↳ CPU, memory, disk, devices, ..

- Kernel core has 27 '.c' files - sched.c, fork.c, mm.c, panic.c, timer.c, ...

After all these .c files are compiled, bzImage is formed which is then booted up. It also contains compiled driver files.

Kernel is closed.

(Object oriented)

- Newer kernels can accomodate modules (drivers) on the fly. This is done by exporting an open interface
- Kernel can crash due to bugs in the driver

- Driver Interface :-

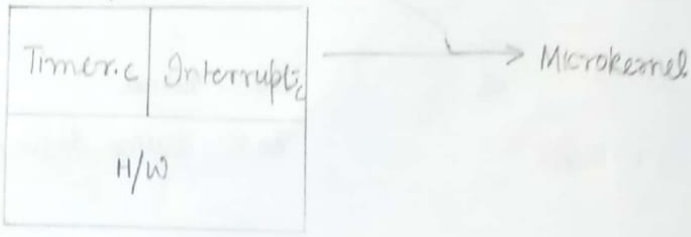
f1(....)
f2(....)
f3(....)

→ Only signatures are known to the compiler. Then at runtime, the compiled modules are inserted → INSMOD

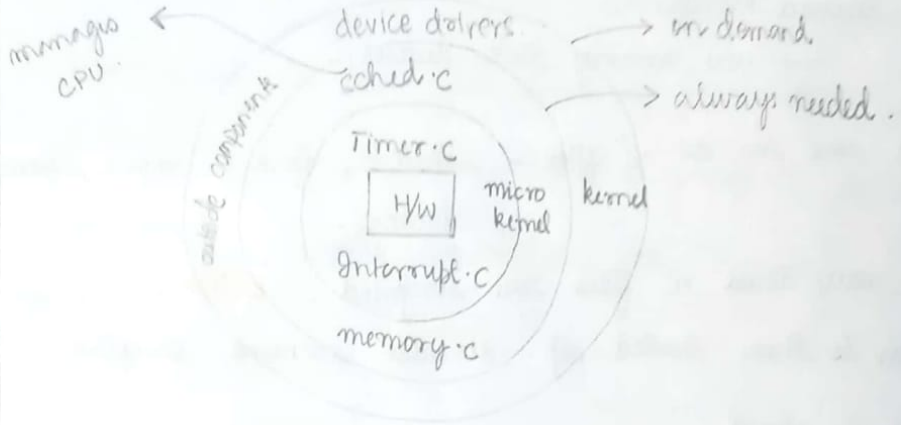
'.ko' file is the compiled module which is inserted into kde.

- When OS is booting up, malloc & calloc are required during compiling user program. Hence must be handled differently

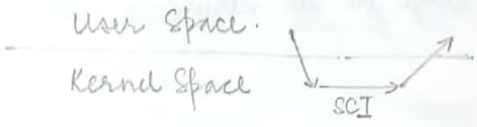
8-8-17
Tuesday



Important part of timer :- synchronization



Kernel provides services to outside components - 64 Calls
System Call Interface



Security issue :- User acquires kernel priviledge for the duration it makes a system call

9-8-17
Wednesday

Process : Program in execution.
↳ Set of instructions

Image : Binary file - Machine Level Instructions

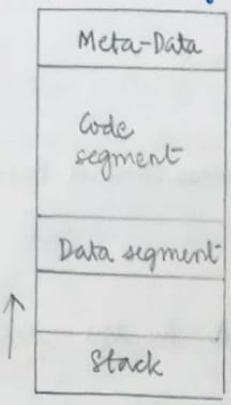
Code segment :- Instructions

Metadata on the image.

Data Segment :- Initialized & un-initialized

Stack Segment :- Used for calls & returning back.

~~stack segment~~ Program Image.

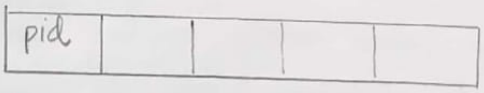


A process owns / shares resources with other processes.
↳ CPU, Memory, Devices, etc.

OS: A set of processes which owns resources & shares it with others.

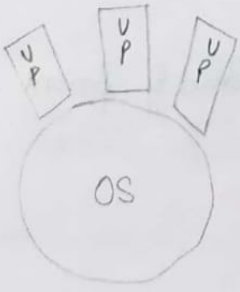
- When a program is compiled & a.out is created, an image is basically created. The OS creates a process out of it by giving it a context.

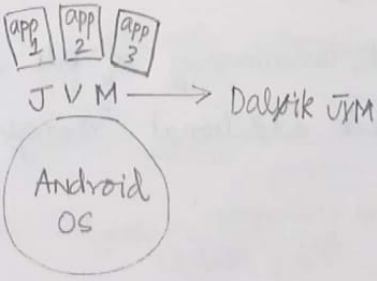
When process is created, ~~a~~ it is added to the process-table. The context is stored there ~~there~~ containing a process-id (pid).



context:
process table
Registers
open file descriptors
priviledge (root / user).
⋮

In Android OS, different applications are sandboxed so as to be treated as separate users. Each application has a pid.





Process in OS

Task - ADA

Process owns resources:

- → Open file descriptors
- → Registers. Eg PC
- → Stack

} Process Control Block PCB

Context of calling program is stored in the activation record

Metadata :- contains format of executable image (.o, .elf)

Re-entrant Code :- Same program shared by multiple users. Same code segment, separate data segment.

- Stack grows bottom-up.
- 64 bit machine : 2^{64} address space.
  32 : 2^{32}

Fork : System call to create a new process.

fork() == 0 if child process is spawned.

Eg.
```
if (fork() == 0) {
        // child process code.
    }
else {
    // parent process code
    }
```

fork() == 0 replicates current process with a separate pid.

fork() is a OS call in Linux for parallelization

Lab 1

Test concurrency of file pointers in child & parent using fork. Check additional things also.

Thread

A light weight process.

Linux: combination of process & thread : called Task

Thread has :-

(i) Stack

(ii) Registers (PC)

(iii) Instruction stream.

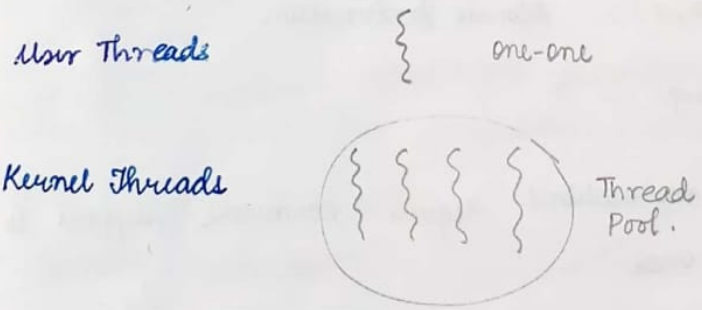
They share data segment.

Context switch of a process requires saving the context of current process & saving all registers, stack, file pointers, etc to restore control to new process. This requires a lot of overhead during scheduling.

* Threads are called light-weight because context-switch is easy, requiring only storing stack pointer & PC.

Note : Fork is an aggressive system call, copying the entire memory map. Clone is less aggressive.

- Since threads share global data, if we use fork making changes to global variables it will be reflected in both.

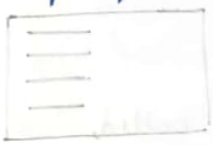


Linux / Unix supports threads by Pthreads library.

Java threads are language-level threads.

- Windows is based on interfaces, not objects :- Component Model.
  ↓ bunch of functions
  → data & fns

Registry.



Service Oriented Architecture (SOA)

Plug & Play.

Bonus Assignment (10 Marks) - Use thread pool for matrix multiplication

Processes share resources

Processes can use a resource one at a time. This leads to problem of mutual exclusion :- Only one process is allowed to be in the critical section.

The CS is protected by locks (exclusive locks)

Special variable :

```
int x;
if (x==1) then
    x=0;
    CS;
x=1;
```

This won't work because context switch may happen before x=0; is executed.
(One line of code may be many lines at CPU Level)

test & set (x, v); Atomic Instruction

compare & swap(...)

The atomic instructions require hardware support to be executed at once.

Semaphore x

P(x): Lock — Will succeed only is x is unlocked.

V(x): Unlock. — always succeed

Binary semaphores: 0, 1

0 → unlocked

1 → locked

P(x) has get test&set within it.

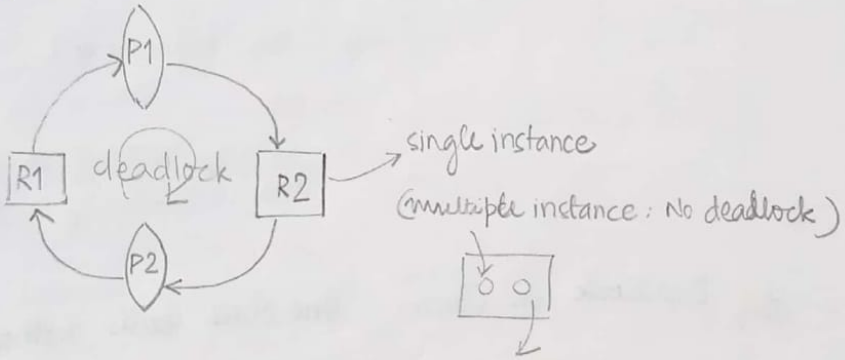
Incase we have more than 1 resources being shared, we can have a counting semaphore.

$x = n$

```
class Queue {
    public:
        add()  { L(f)  L(R) }
        delete() { L(f)  L(r) }
};
```

* Deadlock : caused due to multiple atomic instructions :-

(i) Mutual exclusion
(ii) Hold and wait
(iii) No preemption
(iv) Circular wait



Two solutions :-

(i) Deadlock detection : wait-free computation . compare&swap
(ii) Deadlock avoidance : allows some wait-free computation . No common synchronization

(a). More instances of resources.

(b). Serialize execution

Number the resources. Only increasing order of resource access.

$R1 < R2 < R3$ ....

Problem: Difficulty in identifying what resources a process needs.

29-08-16
Tuesday

Most of the OS don't worry about deadlocks. They expect user to press CTRL + ALT + DEL. But not real-time systems

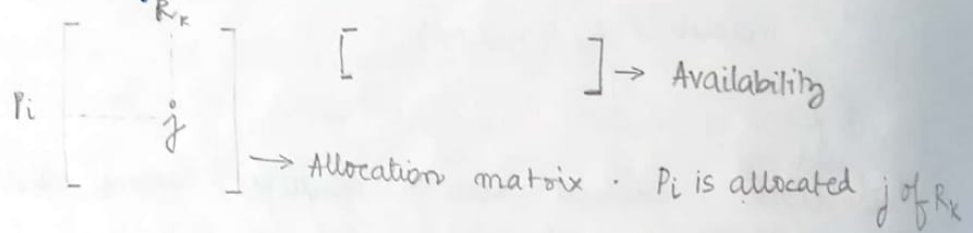
Bankers algorithm (Theoretical since we have to know the resource requirements of processes apriori).

Safe state

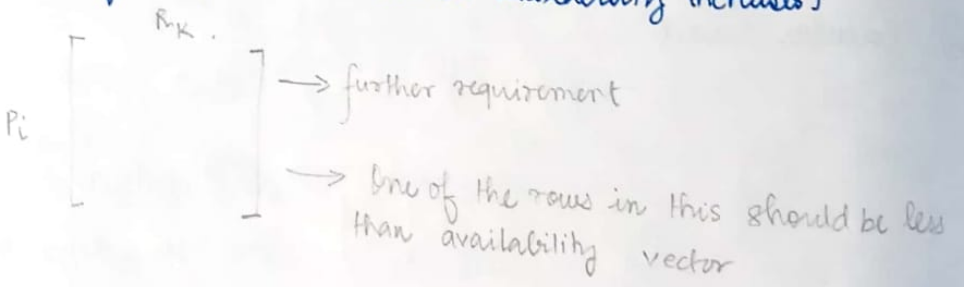
Unsafe state

System resource allocation is always in a safe manner

Availability vector.

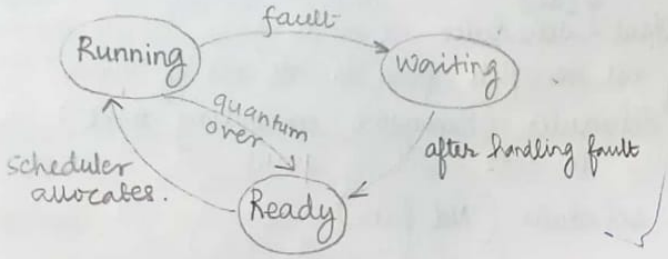


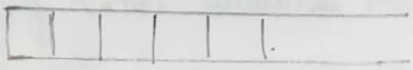
After a process finishes execution, the availability vector is modified (certain resources availability increases)



(*) Deadlock in Java: One class calls method of other ~~ob~~ class & vice versa.

Process Scheduling



Run queue  → priority queue.

(linux implements it as a RB tree).

task struct {

}

In case of multi-core, every core has its own run queue.

per-cpu : one-scheduler.

31-08-2017
Thursday

Timeline of Linux Schedulers

Initially: Fast & Easy Scheduler (Simple queue. Remove a process from front, execute it, if not complete push it to end of queue).

(ii). Fair Scheduler (Expensive computing time) Puts jobs of higher priority towards front of queue

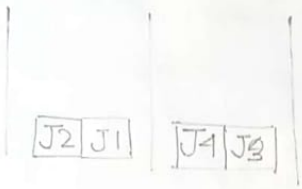
(iii). Fast Scheduler (Not fair)

(iv). Completely Fair Scheduler (CFS): from 2.623

Interactive job response time

High throughput for long running jobs

Some job may not complete because of some I/O requirement. So it is not fair to remove it from the queue once it's time slice is exhausted



2 cores

It is better to have J1 & J3 → CPU Intensive
J2 & J4 → I/O Intensive

rather than having similar jobs in the same core

Problems with the round-robin scheduler.

1. A job does not get full time slice.
2. A job waiting for an event.

Modification in CFS: time ordered queue

Order the jobs according to the CPU time received by it. The structure used in this is a R-B Tree. Pick the left-most child in the tree.

See: In sched.h, there is a task_struct which contains stack, flags. In the run queue, the pointer (sg) to the task_struct is stored as tree's node.

check: Time slice of Linux scheduler = 25 ms.

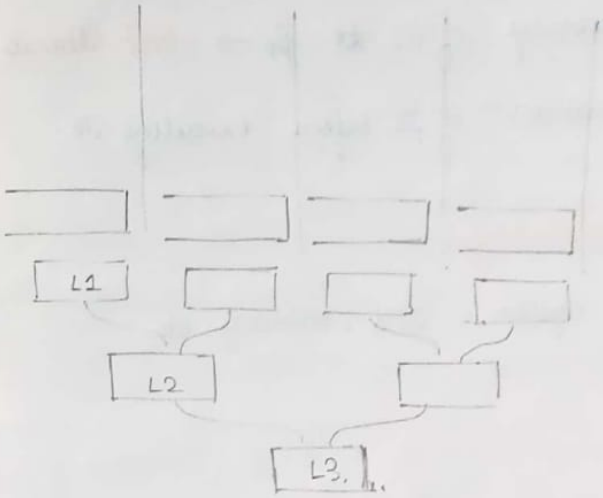
load : run queue length.

In a multi-core architecture, if there is load imbalance across cores, some process is distributed to other cores. Load Balancer

Problems :-

(i) More L1 cache misses.

(ii) Network traffic on bus.



05-09-2017
Tuesday

General Purpose Systems : fairness to all jobs (CFS)

Two metrics :-

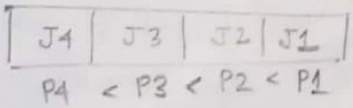
(i) Throughput (No of processes executed in unit time)

(ii) Average response time.

Embedded Systems :

Hard Real time system : If a job misses the deadline → system failure

Soft Real time system : Miss → not system failure
User perceives it
Priority level 0-100.
Priority based scheduling.



P4 < P3 < P2 < P1

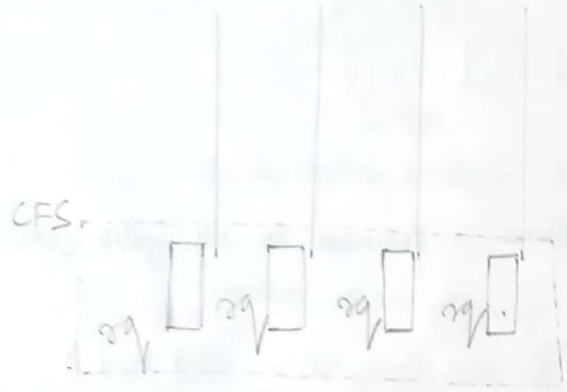
Pick job of highest priority & execute to finish.

Issue :- A higher priority job pre-empts lower priority job. Eg - In case a new job of priority $p_0 (> p_1)$ comes in, leads to priority inversion.

Say J1 is executing its critical section & hasn't signalled semaphore yet. The new job displaces J1 (J1 goes to wait state) & remains in a forever wait. This can be transcended by increasing priority of J1 to p_0 → Inheritance & complete execution of J1 before executing J0.

Multi-Core Scheduling.

Throughput of system = CPU, memory, etc.




```
Load_Balancer (){

        migrate(..);

    }
```

Performance will take a hit because if a process is migrated from Core1 to Core4, there will be L1 & L2 cache miss.

Process variables :- ff, iops, cache misses at different levels

06-09-2017
Wednesday

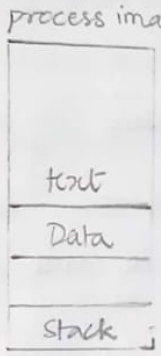
Memory Management

Secondary storage	
Main Memory	
Cache	LLC L2 L1
Registers	
Processor	

Two hierarchies of memory managements :-

(1) Pages (fixed size page)

Virtual Memory :- Allows execution of program requiring more space that main memory limit.



100 pages (Each page is 8kb).

The process resides mostly in secondary storage (∴ swap area)

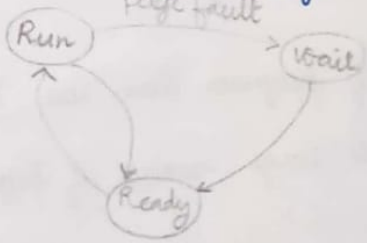
Virtual memory → swap (100 pages).
→ physical memory (5 pages)

If a curtain page isn't present in main memory : page fault

Process running comes to wait state. The memory manager takes over. It brings the page to the main memory.

Then the process is moved to ready state.

This won't work if size of data segment > main memory size.



Text part is READ ONLY. (Re-entrant code).
Data has to be READ-WRITE.

Note: Sometimes Text is not Re-entrant. Eg- Scheme language allows fns to be declared as variables.

So ideally text pages never become dirty. Only data can become dirty.

- Consider quick Sort - It requires entire array to be present in memory while sort. Hence if we have a big array, it cannot be sorted.

07-09-2017
Thursday

Page faults cannot be made 0 because at least the first time they need to be brought from memory.

Reference string : Successive ~~memory~~ page accesses by a process.
Eg. P1 P3 P2 P4..

Locality of reference: If a given memory location is accessed, chance that a close-by location will be next accessed is really high.

Note: Page size ≠ Program size. Leads to segmented memory management.

Working set of pages : At any given point of time, the program is using repeatedly using some pages.

Eg-
```
for {
    P1
    P2
    P3      } Working set
}
```

If program has less pages than working set size, it leads to large number of pages faults.

Finding out working set window size

Observe the reference string & adjust working set window size by identifying pattern.

Note. (i) Compiler can sometimes modify loops in program to serialize memory accesses.

(ii) ~~i = 3;~~ Compiler removes this
i = 4;

Sometimes this is not desired for eg. when i is a memory-mapped location.

Inter-process memory protection is very important.

12-09-17
Tuesday

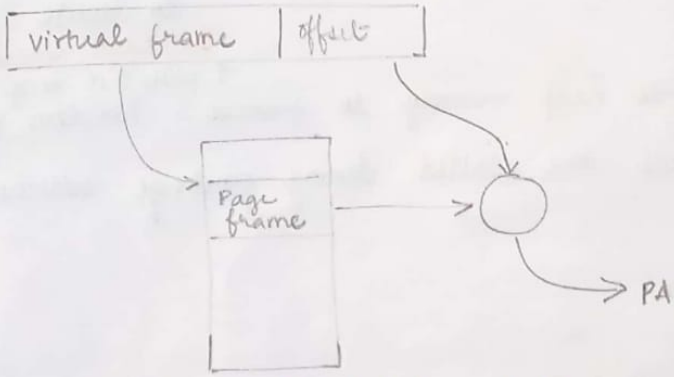
32 bit address space

Page Size = 4 kb.

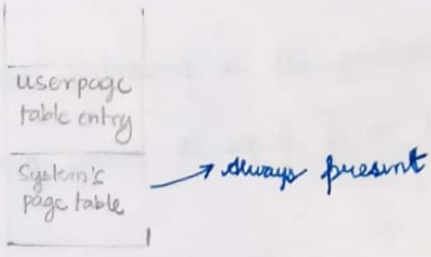
2^{20} Page Table Entries

4 byte per PTE ⇝ Total = 4 mb. Page Table size

100 processes ⇝ 400 mb (Too large).



TLB: Transfer Lookahead buffer.



pagefault_handler () {

1. Page replacement algorithm.
2. Page is clean & target.
3. Start another process for I/O.
4. control to scheduler.

}

Increase efficiency : Multi-level paging. Eg RISC: 4 Levels of paging
decrease size & no. of pages to be swapped
to main memory.

Page Replacement Algorithm

Local replacement : intra-process → a fixed number of page per process.
Global replacement : inter-process

↓

(i) Least Recently Used (LRU)
Check the page reference string to decide.

Ballady's anomaly : No. of page faults ↑ with ↑ in no. of pages.

OS gives Heap memory to process. Problem of garbage collection
Processes are stalled during garbage collection.

(ii). LIFO ×

(iii) FIFO : good.

1 2 3 4

Locality of reference : Working Set is present in memory.
Instead of giving a fixed no. of pages to every process, give it it's working set size.

If a page size is really large, leads to internal fragmentation

Reducing no of page faults

(i) Increase page size

(ii) Give a process it's working set size no of pages.

(iii) Improved replacement algorithms.

Page Replacement algorithm

LRU Policy

Move clockwise until you find a page with 0 bit. Replace that page with new incoming page & set bit to 1. In the process of moving clockwise, set pages with bit = 1 with bit = 0

Clock Replacement Algo.

Optimizations

(i) Second-chance algorithm:-

(ii) Dirty page algorithm (If it's dirty, don't replace)

FIFO Policy : Implement using a simple queue.

C Program : "Dynamic" Memory Management

Local vs Global

↓

Target the process pages

Static pages of a process remain with it for entire execution.

Dynamic pages (heap) needn't remain for entire duration

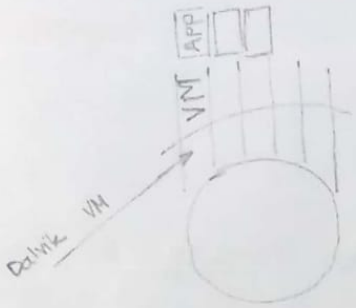
Memory Leaks : Program takes up dynamic pages but doesn't release it back.

Drastic performance degradation due to memory max utilisation.

When a process completes it's pages are re-claimed but not the pages of the heap.

Note: Java: Garbage Collector. Programmer cannot free memory

Reference Counter Code : When the no. of objects referring a given object becomes 0 → becomes garbage.



Android OS

ARM based : No hard disk (no swaps) → small memory footprints.

JVM: Stack based machine → Implicit arguments for procedures.
↳ designed for platform independence.

Advantage: (i) Simplicity
(ii) Easily compile HLL

JustInTime Compilation [Converts stack based to register based]

Disadvantage: (i) Large overhead for HLL. (8 times slower) Because there is an intermediate compiler working on JVM which is complicated.

Java Compiles its files into .class format which contains

magic
constant pool
Superclass
Fields
Methods

66%

33%

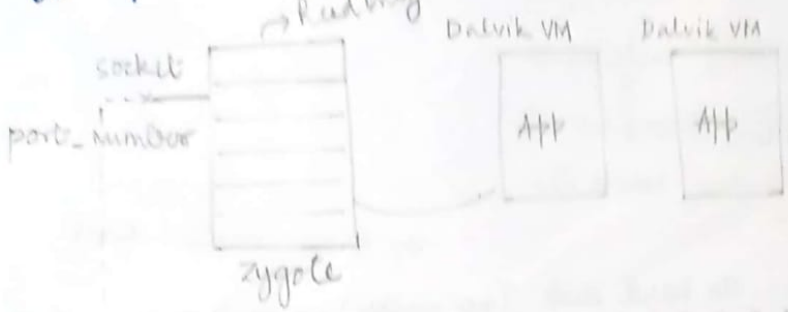
Note: Newer Java Compiler saves executables as ".dex". Multiple classes in the same executable. Shared constant pool (Heterogeneous Pool). Compression happens by avoiding duplicacy. Types are treated as string constants & usage is done by indexing.

JVM booting is very slow: loading to memory at cold start

Soln to speed up :-

(i) Heap pages read only.

(ii) Zygote process (VM pages)



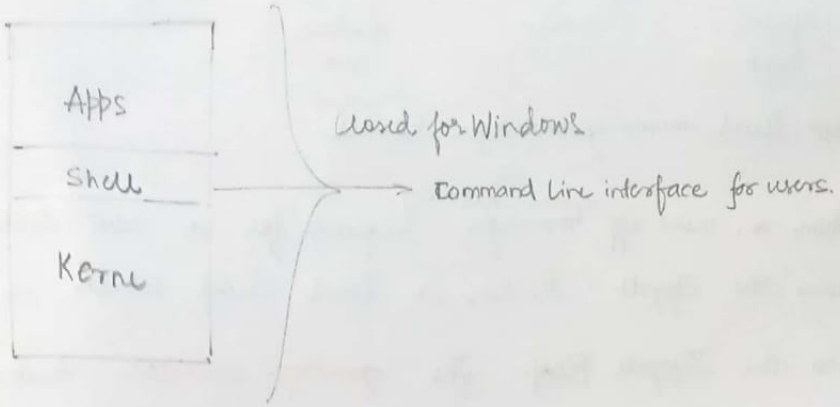
Note: Send mail : telnet . cse . iitm . ac.in 25
↳ port-number.

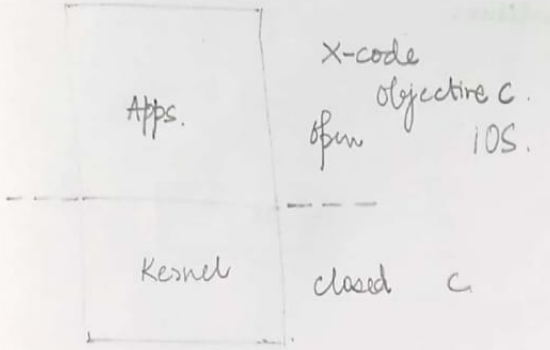
Dalvik VM : register based → efficient for ARM.

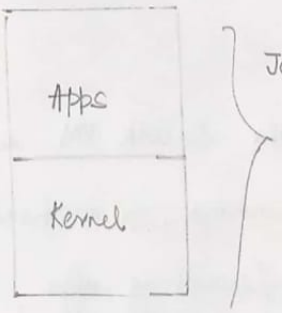
Kernel's memory management
VM's memory management
} Kernel allocates some memory to VM which in turn manages that.

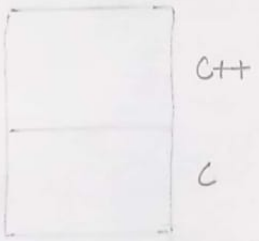
Least Frequently Used

OS Design

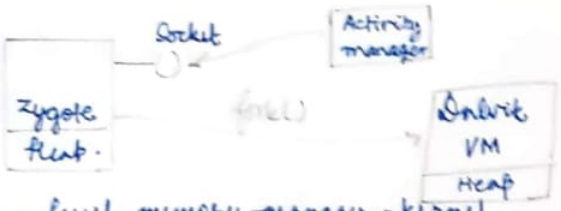








Note: Android OS can be crashed by making repeated requests for Dalvik VM from the Zygote socket.

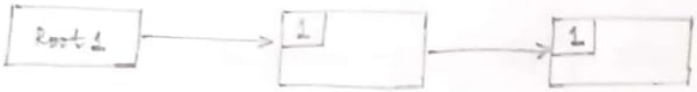


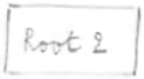
Low-level memory manager - kernel

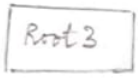
When a new app comes, its activity manager requests for a new Dalvik VM from the Zygote. It has a local heap which is mapped onto the Zygote Heap. The garbage collector kicks in when the memory is low.

Garbage collection Algorithms

(i) Mark & Sweep.







If the heap size allocated to the Dalvik VM is

(a) Low :- inefficient for app running, GC overhead.

(b) High :- latency in creating / switching apps, smooth performance.

(ii) Generation Collector.

(a) Several generations of objects.

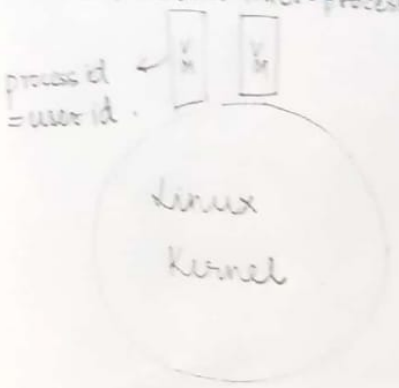
(b). Object surviving for longer time in young generation move it to older generation

(c) Perform mark & sweep in younger generation.

The objects which survived garbage collection means that it is being used. Hence it is moved to older generation. Those which are removed are no longer needed.

In case there are not many objects to remove from younger generation, we perform mark & sweep on older generation.

Q. Can a process modify its own page table entries? No because this would threaten inter process protection.



app → uid access list

s owners | group | others.

- rwx rwx rwx

↳ system privilege (access to root file systems).

Eg- User password program.

r → read
w → write
x → execute
} permissions . changed by "chmod" in Linux.

uid → manifest file ⇝ set permissions at kernel level.

Any device is considered as a device file having privileges.

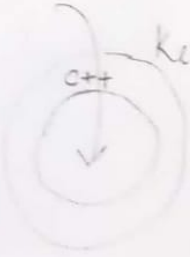
Android apps communicate with each other using binders.

System call.

Send.msg	code

Filter → wrapper code around sys_call.

```
{
  file_obj fo;
  fo.msg();  → each system call
               intercepted by filter.
}
```



SE Linux (Security Enhanced) has extra security everytime a new object comes up.