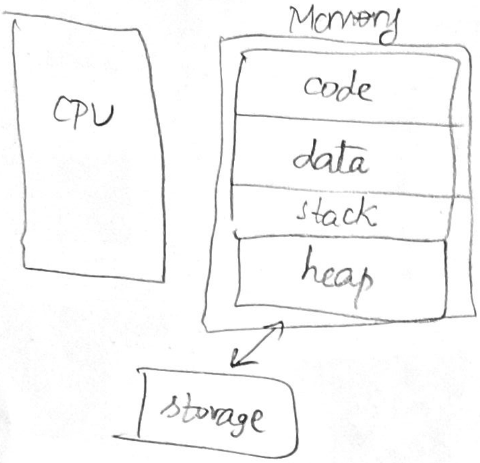


① Process abstraction

② Virtualization

- time sharing
- resource sharing

"limited direct execution"

Isolation — Performance

③ Isolation why?

- process can be buggy
  - crash
  - infinite loop
- process can be malicious
  - take over other processes / users

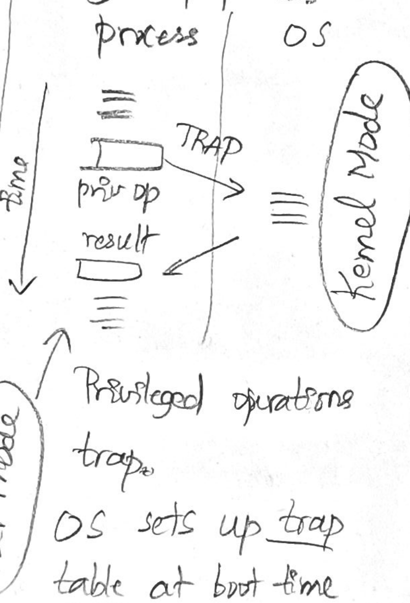
④ System Resources (cpu, mem, disk, net)

* access only after vetting ⇒ but process directly executed.
* Get hardware support

privilege levels

traps

⑤ trap process



Privileged operations trap.

OS sets up trap table at boot time

⑥ trap table — HW data structure

trap#	address of fun/handler
0	→
1	→
:	

→ examples

* Boot time setup

① access the storage

② execute any privileged operation, in general

⑦ system calls

→ library to do reg assembly

⑧ Basic limited direct exec.

OS (kern mode)	process (user mode)
Boot time set up all trap handlers & trap table	(nothing. All this even before process starts)
Run time Initialize text, data, stack w/ args save regs return from trap →	

⑨ sharing between processes

① cooperative approach:

→ wait for sys call

→ yield ()

② Non cooperative approach

Timer Interrupt

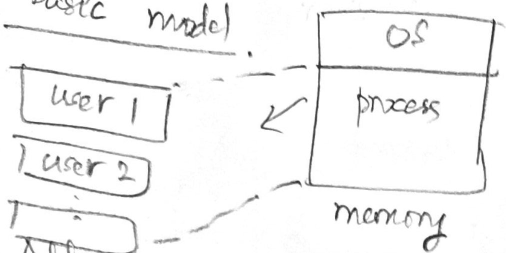
hardware support once again..

⑩ Timer Interrupt

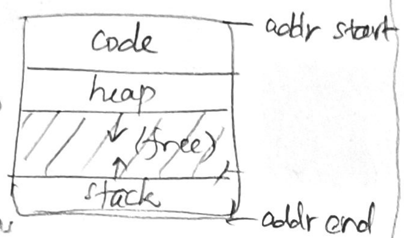
Boot time kernel	hardware
→ start timer interrupt & set up handler	
CONTEXT SWITCH * (whenever chose to move to diff process)	timer interrupt * save regs to kernel stack move to kern mode Jump to handler

11 Virtualizing memory.

Basic model

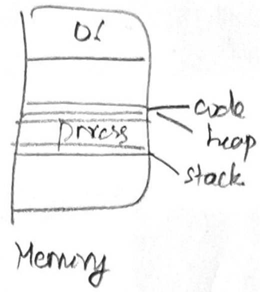


Address space



* virtual addresses
* actually located at different locations

12 Dynamic relocation



User abstraction

→ "fixed" addressing/instructions regardless of location

13 Hardware translation

* Base & bounds

load register, addr (// vaddr)

→ virtual to phy addr

{ if (base) < base + vaddr < bound
  load from base + vaddr

* else fault // trap

Memory Management unit (MMU)

14 primitives for hardware address translation.

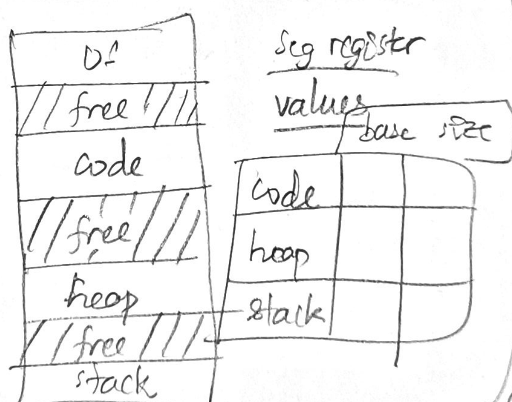
1. privilege modes
2. base / bound registers
3. check vaddr within bound
4. privileged insn to update base / bounds ⇒ "sensitive"
5. privileged insn to register exception handlers
6. raise exceptions on invalid address

OS must start managing memory

15 segmentation

empty mostly addr spaces

⇒ generalize to many base and bounds.

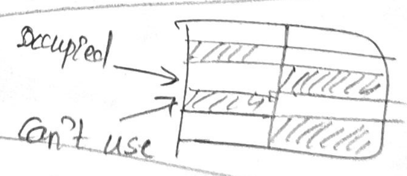


16 hardware for segmentation

1. tell hardware which segment?
   top k bits of addr
2. Memory protection bits.
   read? write? execute?

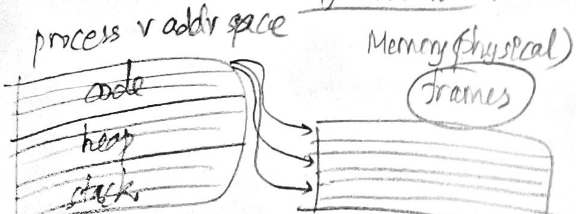
problems

1. context switch
   → save & restore seg descript-ors
2. non compact memory layouts!



17 Paging

use fixed size chunks. Many of them.

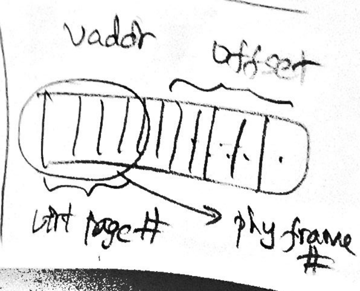


* can't use dedicated registers any more

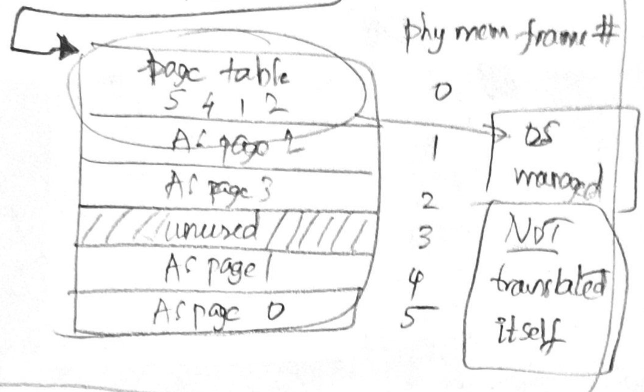
⇒ page table in memory.

Per process data structure

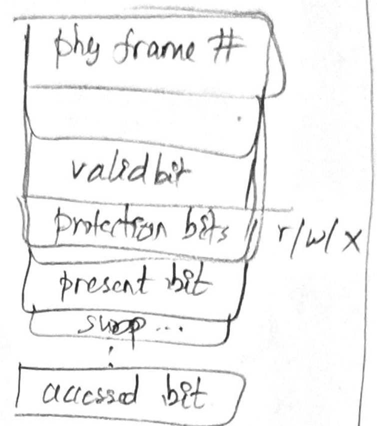
18 Translation



(19) example



(20) Page table entry



(21) translation (hardware)

VP# = (vaddr & vmask) >> SHIFT

PTE addr // linear pagetable
= Page Table Base Ptr + (vp# × sizeof (PTE))

translate to get phy frame#

offset = vaddr & offset mask.

Phy addr = (pf# << SHIFT) | offset

(22) translation

↳ check valid, protection, PTE valid.

Else trap! →
- Minor fault (retrieve shared)
- Major fault (retrieve from disk)
- Invalid fault (vaddr not valid)

(23) Paging is ~~too~~ inefficient / slow

- at least two mem access
- Large memory just for page tables

(24) Address translation cache (TLB)

- efficient memory, small → cache PTEs
- first check TLB
  - hit ⇒ use (protection etc).
  - miss ⇒ walk page table

(25) Handling TLB miss

- hardware managed TLB (Intel x 86)
- software handled (RISC) (use traps)

(26) TLB entry

vp #	pf #	protection bits etc.

Fully associative cache

(27) TLB + context switches

(*) Per process data structure of pagetables

option (1) TLB flush

option (2) tagged TLB
- use addr space ID

(28) smaller page tables?

→ multi level page tables
page directory + table

→ superpages

→ hybrid segment & paging

→ swap page tables to disk

(29) TLBs make paging feasible

~~→~~, phusically indexed versus virtually indexed caches

(30) Principles applied to entire operating systems!

- Xen
- amazon ec2

+ impact

(31) system virtualization

1. OS assumes it is the most privileged entity.
2. OS virtualization. why?
   ⇒ time sharing OSes
   ⇒ unit of software/app deployment — any software & config can be run
3. actually properties of both OS & hardware arch matter (together provide isolation)

(32) virtual machine monitor (VMM) 1974

→ virtualize all system resources (processor, mem, disk/net/I-O, ...)

3 requirements of interest (popek & goldberg)

1. equivalence
   – same as direct execution on underlying phy machine
2. Resource control/safety
   → VMM must have complete control of resources
3. efficiency
   → mostly execute without VMM intervention (not a "requirement" for functionality. But need it in practice.)

(33) ISA classification

1. privileged insn's
   → must trap in user mode
2. control sensitive
   → can change processor mode or "whether something will trap in 1st place" (eg. change mem bounds)
3. Behaviour sensitive
   → effect of instruction depends on mode or bounds registers

(33) theorem (basic result)

(unmodified guest OS) efficiency virtualized if $S2 \cup S3 \subseteq S1$ } FULL virtualizn

→ "trap and emulate" virtualization (for privileged instructions)

→ sufficient condition, — x86 does not satisfy it (classically) → now Intel VT-x and AMD-V

→ unmodified guest OS

→ workaround tech

1. Dynamic recompilation – replace critical insn's at run time.
   - cache emulation code
   - assist through hardware
2. paravirtualization → port before running (Xen)

(34) Containers

1. kernel namespaces (access isolation) < process, network, mnt
2. Control groups (resource isolation) < process, pid, cpu, mem/blkio
3. union filesystems ← copy on write ← layering

(35) xen

(1) cpu → ring 1 os
(else trap ring 0)

- synchronous hypercalls
- system calls
  - easy.
  - same handlers.
- page faults
  - extended stack frame
  - (cr3 unreadable)

(2) I/O virtualization

Ring buffers
async producer-consumer
descriptor-data separation / zero copy.

(3) Memory — guest OSes manage pagetable
(types) writes validated by xen.
avoid effect of TLBflush on xen

(4) Management
- separate policy from mech
- dom0.