VAX/VMS

(Fall 2000)

Review:
- Phys mem / VM: goals?
  - ease of use
  - sharing PM
  - protection
- Addr Space
  - code, etc.
- [Dyn. Reloc / Segmentation / Paging] +/-

(11/750 11/780)

Vax 32-bit mini 512 byte

AS: 32-bits 2|21|9

P_0 | P_1 | Sys | reserved

- P_0: text/heap →
- P_1: data/sys/stack ←
- Sys: data, procs, heap, s/w vectors, etc. ← shared

PT: linear (base+bounds)

example translation

V	Prot	M	"OS"	PFN
1	4	1	5	21

(V = valid, M = modify)

(no ref. bit!)

too big! pageable PTs!

TLBs, etc. (split user/sys)

ctxt switch: P_0 BR, P_1 BR, flush TLB

Replacement: no global LRU

why? hog, expense

FIFO, per process "resident set"

sys has resident set too (page table exception)

+ 2nd level global cache

clustering: text in, mod'd out, ~~sys resid~~

swapper: move process in/out

VAX/VMS

Problem

VM for broad class of machines

VAX-11/750, 11/780 — classic 32-bit mini's

Large range of configurations

Concerns

- Effects of single heavily paging program
- Start-up time w/ demand paging
- Disk traffic due to paging
- CPU time spent dealing w/ page info

Solution

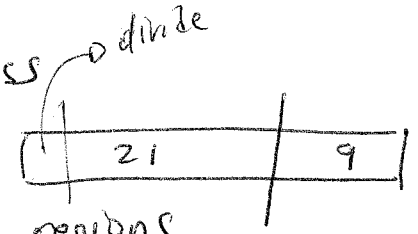
Simple VM w/ lots of good little ideas

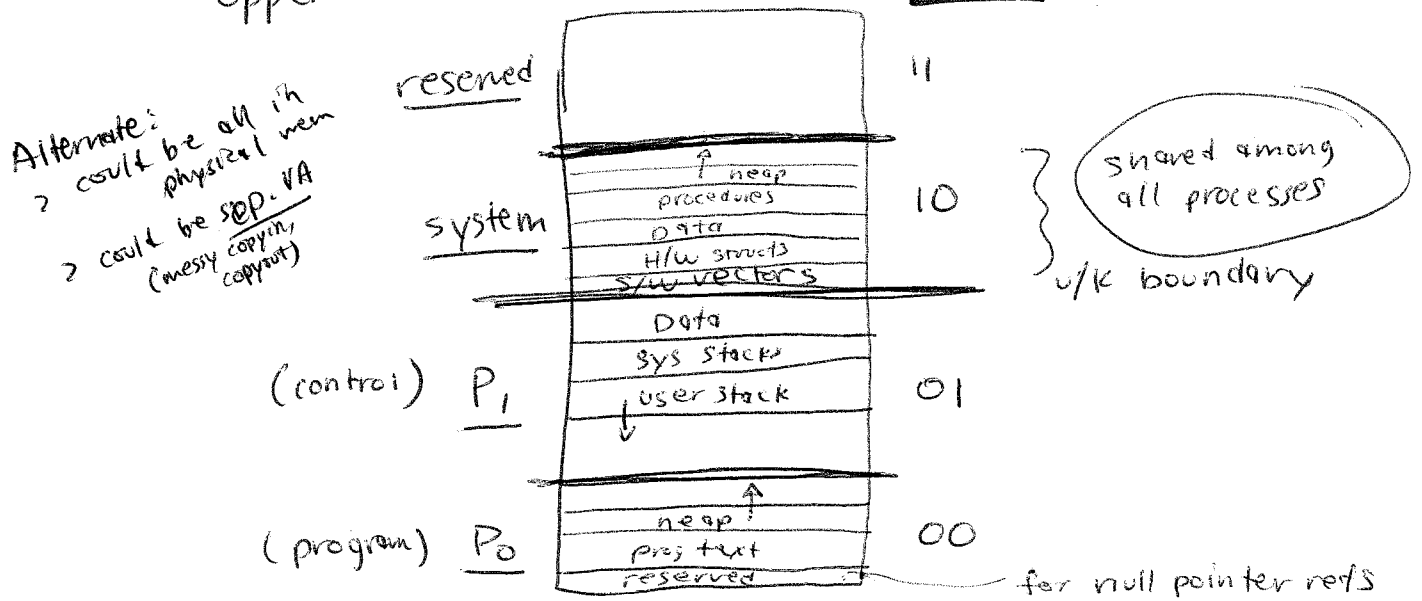
VAX-11 H/W

32-bit VA per process

512 byte pages

upper 2 bits divide into regions



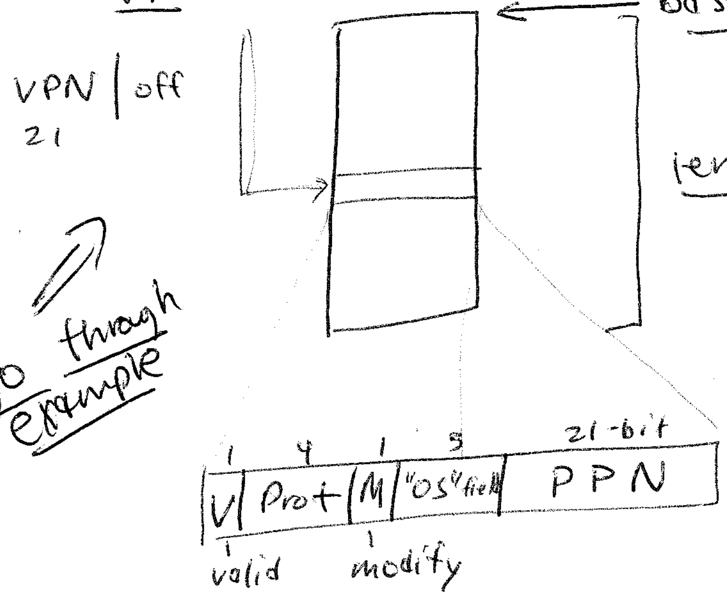


Page Tables

Each region defined by page table [live in system space]

> Contiguous array of mappings (**linear page table**)

VA



base register

length register (valid mappings)

go through example

no ref bit!

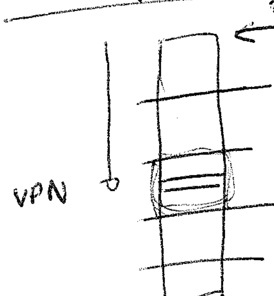
> **Problems** w/ linear table?
(can get too big!)
solution: make P_0, P_1 page tables page-able!

⇒ 1) load VA ~~page fault~~ (translate)   VPN / offset

where does P_0 PT live?

2) lookup in P_0 page table (in system **VA**)

P_0 base register

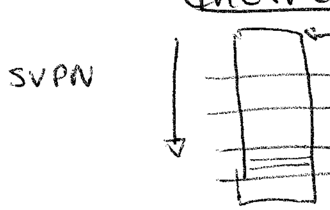


base register + VPN
⇒ **SVA**

> use **TLB** to make fast
(split user/ system)
[flush user only on ctxt switch]

3) load SVA ~~(Another fault!) (is possible!)~~
another translation

Sys base register ← (virtual or physical?)



Pager

> Lots of systems use global LRU

Q: why bad?
- could have a hog
- could be expensive to implement (scan ref. bits) (how to implement "perfect" LRU?)

Instead, use "process local" replacement

Resident set: current set of pages

OS keeps per process FIFO

Problem: ? FIFO is not a good policy

Solution: Add second-level global cache

free page list
mod. page list
} add to tail

If fault on a page, check lists; if there, reclaim

Q: what is pathological case?

=> w/ "right" amount of global memory, can approach LRU ... what are the trade offs?

> Clustering: 512-byte pages: too small

try to group I/Os into multi-page chunks

1) Faulting in text pages (if consecutive VM are consec. on disk) some help from linker
2) Writing out modified list to paging file (sort by process)
3) Reading back from paging file

Q: which is biggest win?

Other Paging Things

> Demand zero + COW [being lazy: a good thing]

> System-space paging
  OS code/data can be paged
  system "resident set"
  Exception: Process page tables
  > page of page table is on process's private res. set list
  > only removed when all mappings invalid
  [why? avoids silliness of page being in memory but page table for that page is not]

Swapper

Q: when is swapping useful?
  keep high-pri resident, avoid thrashing
  > Separate process (also writes out dirty pages)
  > will not load process until physical space for RS is available
  (Q: problems?) **starvation**

> Swap: write out entire resident set

App interface

- Expand P_0, P_1 (unix corollary?)
- Inc/Dec Res set size } privileged!
- Lock/Unlock pages }
- Create/Map pages > Produce record of page-fault activity

Summary

> Neat, simple clear VM
> Lots of small good ideas
> clustering, lazy writes, process-local replacement