354

→ Parallel Learning

(in the dark)
Little Operating Systems

\[ \text{(OS)} \]

\[ \Rightarrow \text{late 60's, early 70's} \]

Diagram:
- CPU
- Mem
- Mouse
- Kbd
- Graphics
- Disk
Virtualization

=> CPU virtualization

  1 physical CPU

  => many virtual CPUs

=> Time slicing / sharing

  \[ A A A A A \quad \text{\|} \quad B B B \ldots \]

=> OS: when it runs

  prog A => OS isn't running

  how to regain control of CPU?

  => h/w mechanism

    (timer interrupt)
interrupt:

happen between instructions handled by OS

(timer)

A ----> A

OS timer int. handler

(timer)

A ----> A

OS

B ----> B

(privilged)

-> protected mode: OS runs in

-> memory virtualization

goal: each running program (process)

have own private, large address space (memory)
Main technique: Paging

Hard to do efficiently \[\Rightarrow\text{time} \Rightarrow\text{space}\]

Basics:

- Fixed size page
- Process usually \((4\text{KB})\)

(virtual address space) \(\mapsto\) virtual pages

\(4\text{KB frames}\)

(today: GBs)

Phys Memory

\(P_A\)

\(P_B\)
Address Translation:

Virtual address (process generates)

\[ \Rightarrow \] Physical address (can use to get info from mem)

Implication:

\[ \Rightarrow \] Structure holds \( V \rightarrow P \) translations

\[ \Rightarrow \] One page table per process

![Diagram](image-url)
VA: 0 ... 31

VA: 5 bits
16 8 4 2 1

VA 10 0 1 1 \Rightarrow (19)

4 pages, 8 bytes each
4 \times 8 = 32

virtual address

movb 19, %al

which byte in the page?

virtual page

#
(VPN)

legal PA: 0...63
VA: [1 1 0 0 1 1]

VPN (2)

PA: [1 1 1 0 1 1] (59)

Page Table:

(Array)

3 2 7

(PFN) 1 1 1
ARG seed 1005
ARG address space size 256
ARG phys mem size 1024
ARG page size 16
ARG verbose False
ARG addresses -1

The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
If the bit is 1, the rest of the entry is the PFN.
If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by each entry of the page table.

Page Table (from entry 0 down to the max size)
00
0x80000000
0x80000022
0x80000024
0x80000016
0x8000001f
0x80000014
0x80000033
0x80000025
0x80000000
0x80000015
0x80000010
0x80000000
0x80000000
0x80000000
0x80000000
0x80000000
0x80000026

Virtual Address Trace
VA 0x00000064 (decimal: 100) --> PA or invalid address?
VA 0x0000005b (decimal: 91) --> PA or invalid address?
VA 0x000000d8 (decimal: 216) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
Page Table Entry (PTE)

Valid bit
0 or 1

Physical Frame Number

Large Virt + 0th page (unmapped)

A.S.

1
Code

2
Code

3
Global

4
Heap

Stack

Page Table

Process: generates VA

if page table entry
valid bit = 0

⇒ illegal mem access

int * p = 0;
printf("%d", * p);
OS role in phys mem management:

When process gets created:

- mem for page table of the new process
- pages for code, heap, stack
- Proc. Virt AS

![Diagram of memory management](image)

- Code
- Heap
- Stack
- Heap2
- Free list (tracks free pages)

OS allocates free pages:

- malloc
- may grow heap
- PT: updated etc.
Page Table Entry:

- **Valid bit**
  - Phys. Frame Number
  - **Read/Write**
    - 0 or 1
      - 0 ⇒ only read
      - 1 ⇒ read/write

V.A.S.(A)

- Code
- Code (read only)
- Heap
- Stack
- r/w
- r/w

Run same prog many times (concurrently)
Problems:
1) Too slow
2) page tables too big

\[ \Rightarrow \text{Too slow} \]
\[ \Rightarrow \text{mov } VA \]
\[ \text{1000, } \%\text{eax} \]

fetch
decode
execute

how many mem references?
\[ \text{eip: } VA \]
\[ 1 \text{ fetch } \]
\[ 2 \text{ mem load } \]
\[ \text{refs} \]

transl ation: 2 more mem refs (to page table)

H/W: cache
specific cache: for address translation

Translation Lookaside Buffer (TLB) H/W
[ Address Translation Cache ]

CPU

generate: VA

<table>
<thead>
<tr>
<th>VPN</th>
<th>offset</th>
</tr>
</thead>
</table>

|PFN| offset |

TLB: holds "popular" Virt → Phys translations
TLB: has 32-512 entries

TLB entry:

<table>
<thead>
<tr>
<th>V</th>
<th>VPN</th>
<th>PFN</th>
</tr>
</thead>
</table>

Valid

Switch: be careful =? $P_A \rightarrow P_B$

CPU:

$VA \rightarrow VPN | offset$

Look in TLB:

Yes $\Rightarrow$ TLB hit

$PFN | offset$

$PA \Rightarrow memory$

$ho \Rightarrow$ TLB miss

Go to wem: Page Table

(fill in TLB w/ translation)
Issues!

Each process has its own virt Addr. Space

TLB: 0 3 A

VPN PFN PID

AGetable

PTA

PTB

What to do?

1) flush TLB on ctxt switch (set valid = 0)

2) (Add PID) bits of all TLB entries per entry

not used

(3) could save (restore TLB contents)
Translation:

TLB miss:
how to find page table entry?

⇒ hardware-managed TLB:
Page Table Base Register

ON TLB miss:
use PTBR +
(VPN * sizeof (PTE))

PT of currently running process
Software-managed TLB:

- Let OS decide page table structure

TLB hit: same (fast, h/w)

TLB miss:
  - No PTBR
  - Trap into OS:
    - TLB miss handler
      - OS: looks up entry in data structure (Page Table)
        - Insert translation into TLB (new instructions)
        - Ret-from-trap
      - H/W: retry same inst
Last Problem:
Page tables: too big

V. Addr. Space

32-bit AS, 4 KB pages

~1 M entries

Page Table

~1 M entries

valid
Translation:

VA \Rightarrow PA

VPN \Rightarrow index \Rightarrow array
Virt AS:

32 KB

VPN: 10 bits
5 bits

Page: [32 bytes]

$2^{10} = 1024$ entries?

(linear)
Page Table

15-bit V.Addr.

Page:
32 bytes

Page table
entry size:
= 1 byte

Page Directory

32 entries
(1 per page of page table)

1024
= 32

$\frac{1024}{32} = 32$
Some basic assumptions:
- The page size is an unrealistically-small 32 bytes
- The virtual address space for the process in question (assume there is only one) is 1024 pages, or 32 KB
- physical memory consists of 128 pages

Thus, a virtual address needs 15 bits (5 for the offset, 10 for the VPN). A physical address requires 12 bits (5 offset, 7 for the PFN).

The system assumes a multi-level table. Thus, the upper five bits of a virtual address are used to index into a directory; the directory entry (PDE), if valid, points to a of the table. Each table holds 32 page-table entries (PTEs). Each PTE, if valid, holds the desired translation (physical frame number, or PFN) of the virtual in question.

The format of a PTE is thus:
  VALID I PFN6 ... PFN0

and is thus 8 bits or 1 byte.

The format of a PDE is essentially identical:
  VALID I PT6 ... PT0

You are given two pieces of information to begin with.

First, you are given the value of the directory base register (PDBR), which tells you which the directory is located upon.

Second, you are given a complete dump of each of memory. A dump looks like this:

```
  0: 08 00 01 15 11 1d 1d 1c 01 17 15 14 16 1b 13 0b ...
  1: 19 05 1e 13 02 16 1e 0c 15 09 06 16 00 19 10 03 ...
  2: 1d 07 11 1b 12 05 07 1e 09 1a 18 17 16 18 1a 01 ...
...```

which shows the 32 bytes found on pages 0, 1, 2, and so forth. The first byte (0th byte) on 0 has the value 0x08, the second is 0x00, the third 0x01, and so forth.

Use the PDBR to find the relevant table entries for this virtual page. Then find if it is valid. If so, use the translation to form a final physical address. Using this address, you can find the VALUE that the memory reference is looking for.

Of course, the virtual address may not be valid and thus generate a fault.
Summarize:

\[ \Rightarrow \text{Virtualization} \]

\[ \Rightarrow \text{CPU} \]

\[ \Rightarrow \text{Memory} \]

END