CS 350 Notes

Matthew Visser

Oct 11, 2011

1 Paging

Page size p, virtual address v. Page sizes are always a power of two.

Page number:

$$\left\lfloor \frac{v}{p} \right\rfloor \tag{1}$$

This can be represented by `v >> y`.

Offset is

$$v \pmod{p} \tag{2}$$

1.1 A Page Size Example

We have a 32-bit address, and the offset is 10 bits. How many bits do we have for the page number? What's the page size? How many pages do we have in an address space?

1. We have 22 bits for the page number.
2. Page size is $2^10 = 1$KB bytes.
3. Number of pages are $2^22 = 4$M pages.

There are some advantages to have small pages

- Reduce fragmentation
- more efficient memory usage

but there are some disadvantages

- large page table.
- processes have many pages, always doing lookups.

Typical page sizes are 256 bytes — 8 kB.

2 Using Page Tables

The kernel writes the address of the page table for a process in the *page table base register*. When the MMU wants to use address lookup it

- looks in the page table base register for the page table,
- divide the address into a page number and offset,
- perform lookup on the page,
- and then construct the physical address.

See diagram on slide 7 of `vm.pdf`.

There are some things in place to protext memory:

- The kernel can set the *valid bit* of addresses set to 0.
- The MMU can enforce a read only bit as well, so a program can't write to memory that is read only.
- Many MMU's have a bit called the *no-execute bit*.

What the kernel does with the page table:

- Saves the state of the page table on every context switch.
- create and manage the page tables
- manage physical memory
- handle MMU errors

The MMU needs to

- translate addresses.
- check for and raise exceptions

The array with an entry for each frame is called the *core map*.

2.1 Midterm Problem on Address Translation

From the winter 2008 midterm.

Page table of process: *p*.

The process has 9 pages numbered 0-8.

Page Number	Physical Address
0	0x8d10
1	0x1004
2	0x004a
3	0x5500
4	0x2220
5	0x2221
6	0x2222
7	0x222a
8	0x5558

Doing this translation with a 4 bit offset, we get the following.

Virtual Addresses	Physical Address
0x00003a8	0x5500a8
0x0001004	NO TRANSLATION
0x0000022	0x8d1022

Doing this with a 12-bit offset, we get:

Virtual Addresses	Physical Address
0x00003a8	0x8d103a8
0x0001004	0x1004004
0x0000022	0x8d10022

2.2 Remaining Issues

We have a few issues

- address translation speed
- sparseness
- where do we put the kernel in this picture?

2.2.1 Speed of Address Translation

- Execution of each machine instruction may involve one or more translations.
  - one for fetching an instruction
  - One or more for instruction operands
- Every translation is a memory lookup, which is slow.

- We can get around this using a *translation look-aside buffer (TLB)* in the MMU.

2.2.2 The TLB

- Each entry contains a (page number, frame number) pair.
- if the address translation is cached in the TLB, then the page table is avoided.
- If the hardware controls this TLB, then the hardware does the address translation using it (unless there is a page fault).
- If this is a software TLB, then the kernel has to keep this page table cached.
- Some MMUs load the TLB themselves, others raise an exception if a page number isn't int he TLB, and the kernel must load the translation to it.
  - MIPS has a software controlled TLB.
- The hardware dictates the format of the TLB. The kernel gets to control how the TLB is used and what is loaded to it in a software controlled TLB.
- TLB is much faster because it's a simple hardware lookup.
- If we switch processes, there will be a lot of exceptions thrown because old mappings will be populated.
  - We either accept the faults, or
  - We flush the TLB every time we have a context switch.

2.2.3 The MIPS TLB

. . . or not. Next time!