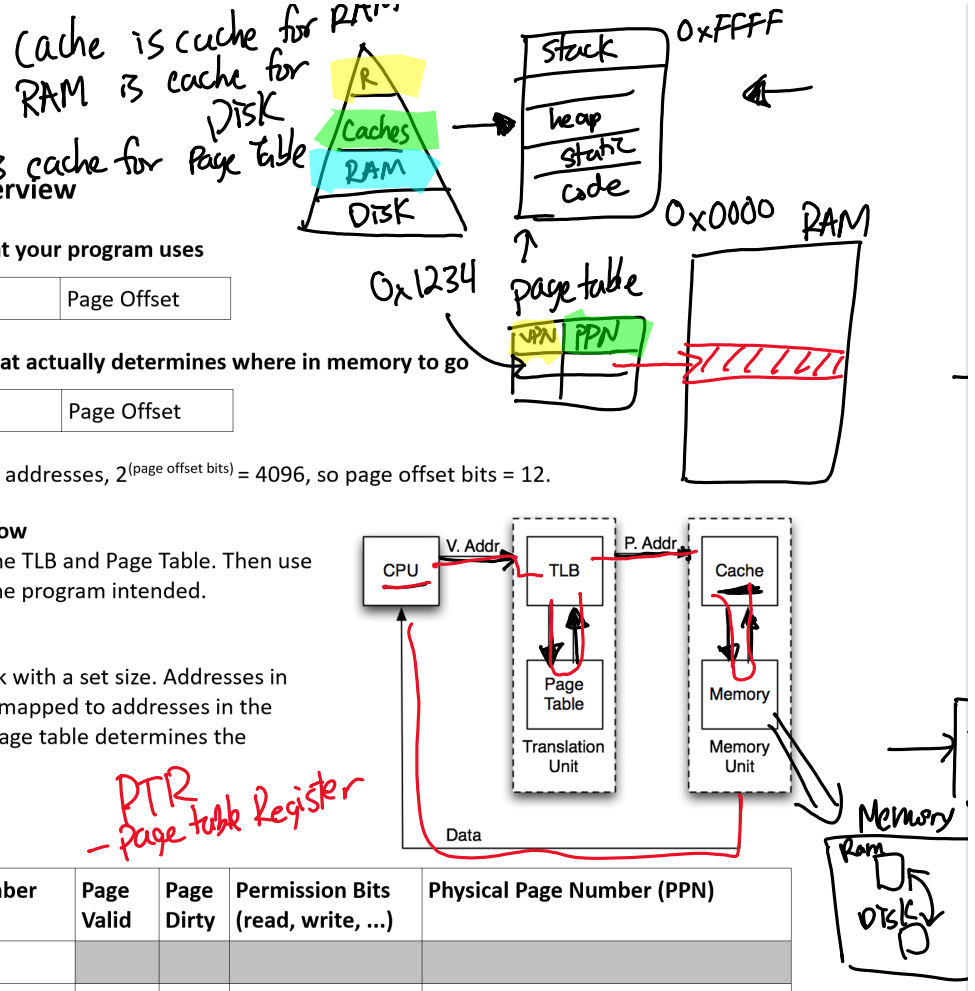
Jon Discussion 13 1

Wednesday, August 1, 2018 9:39 AM



Virtual Memory Overview

Virtual address (VA): What your program uses

Virtual Page Number	Page Offset

Physical address (PA): What actually determines where in memory to go

Physical Page Number	Page Offset

With 4 KiB pages and byte addresses, $2^{(\text{page offset bits})} = 4096$, so page offset bits = 12.

The Big Picture: Logical Flow
Translate VA to PA using the TLB and Page Table. Then use PA to access memory as the program intended.

Pages
A chunk of memory or disk with a set size. Addresses in the same virtual page get mapped to addresses in the same physical page. The page table determines the mapping.

PTR - page table Register

The Page Table

Index = Virtual Page Number (VPN) (not stored)	Page Valid	Page Dirty	Permission Bits (read, write, ...)	Physical Page Number (PPN)
0				
1				
2				
...				
(Max virtual page number)				

Each stored row of the page table is called a **page table entry** (the grayed section is the first page table entry). The page table is stored *in memory*; the OS sets a register telling the hardware the address of the first entry of the page table. The processor updates the "page dirty" in the page table: "page dirty" bits are used by the OS to know whether updating a page on disk is necessary. Each process gets its own page table.

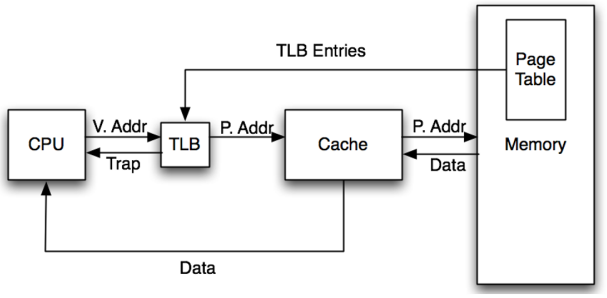
- **Protection Fault**--The page table entry for a virtual page has permission bits that prohibit the requested operation
- **Page Fault**--The page table entry for a virtual page has its valid bit set to false. The entry is not in memory.

The Translation Lookaside Buffer (TLB)

A cache for the page table. Each block is a single page table entry. If an entry is not in the TLB, it's a TLB miss. Assuming *fully associative*:

TLB Entry Valid	Tag = Virtual Page Number	Page Table Entry		
		Page Dirty	Permission Bits	Physical Page Number
...	...	...	...	...

The Big Picture Revisited

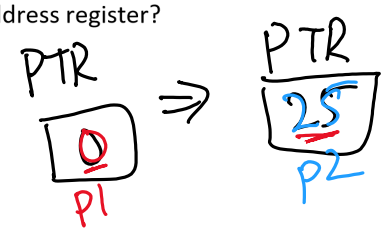


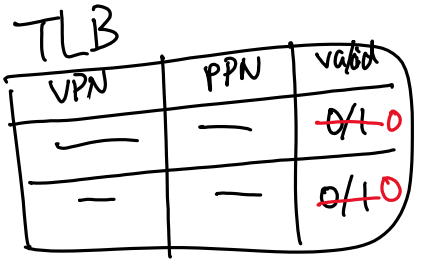
Exercises:

1) What are three specific benefits of using virtual memory?

1) every program has full addr space & contiguous
2) protection for data
3) illusion of ∞ memory

2) What should happen to the TLB when a new value is loaded into the page table address register?





page offset: 8 bit
$256 = 2^8$

3) A processor has 16-bit addresses, 256 byte pages, and an 8-entry fully associative TLB with LRU replacement (the LRU field is 3 bits and encodes the order in which pages were accessed, 0 being the most recent). At some time instant, the TLB for the current process is the initial state given in the table below. Assume that all current page table entries are in the initial TLB. Assume also that all pages can be read from and written to. Fill in the final state of the TLB according to the access pattern below.

final state of the TLB according to the access pattern below.

VPN : total bits − page offset
: 16 − 8 = 8

Free physical pages: 0x17, 0x18, 0x19

Access pattern:

1	Read	0x11f0
2	Write	0x1301
3	Write	0x20ae
4	Write	0x2332
5	Read	0x20ff
6	Write	0x3415

VPN: 0x11 hit
VPN: 0x13 miss
VPN: 0x20 hit
VPN: 0x23 miss
VPN: 0x20 hit
miss

Initial TLB

VPN	PPN	Valid	Dirty	LRU
0x01	0x11	1	1	0
~~0x00~~ 0x13	~~0x00~~ 0x17	~~0~~ 1	~~0~~ 1	7
0x10	0x13	1	1	1
0x20	0x12	1	~~0~~ 1	5
~~0x00~~ 0x23	~~0x00~~ 0x18	~~0~~ 1	~~0~~ 1	7
0x11	0x14	1	0	4
0xac	0x15	1	1	2
~~0xff~~ 0x34	~~0x16~~ 0x19	1	~~0~~ 1	3

8

LRU1	LRU 2	LRU 3	LRU 4
1	2	3	4
7	0	1	2
2	3	4	5
5	6	0	1
7	7	7	0
0	1	2	3
3	4	5	6
4	5	6	7

Final TLB

VPN	PPN	Valid	Dirty	LRU
0x01	0x11	1	1	5
0x13	0x17	1	1	3
0x10	0x13	1	1	6
0x20	0x12	1	1	1
0x23	0x18	1	1	2
0x11	0x14	1	0	4
0xac	0x15	1	1	7
0x34	0x19	1	1	0

same

LRU5	LRU 6
4	5
2	3
5	6
0	1
1	2
3	4
6	7
7	0

Dis 10

3. Data race and Atomic operations.

The benefits of multi-threading programming come only after you understand concurrency. Here are two most common concurrency issues:

- **Cache-incoherence**: each hardware thread has its own cache, hence data modified in one thread may not be immediately reflected in the other. The can often be solved by bypassing cache and writing directly to memory, i.e. using volatile keyword in many languages.
- The famous **Read-modify-write**: Read-modify-write is a very common pattern in programming. In the context of multi-thread programming, the **interleaving** of R,M,W stages often produces a lot of issues.

To solve problem with Read-modify-write, we have to rely on the idea of **undisrupted execution**.

In RISC-V, we have two categories of atomic instructions:

- Load-reserve, store-conditional (undisrupted execution across multiple instructions)
- Amo.swap (single, undisrupted memory operation) and other amo operations.

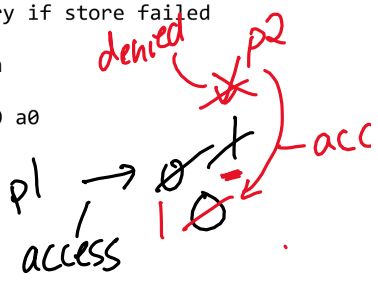
Both can be used to achieve atomic primitives, here are two examples.

Test-and-set

```
Start: addi t0 x0 1       #locked state is 1
       amoswap.w.aq t1 t0 (a0)
       bne t1 x0 Start    #if the lock is not
                          #free, retry

       … #critical section

       amoswap.w.rl x0 x0 a0    #release lock
```

Compare-and-swap

```
#expect old value in a1,
#desired new value in a2
Start: lr a3 (a0)
       bne a3 a1 Fail #CAS fail
       sc a3 a2 (a0)
       bnez a3 Start #retry if store failed

       … #critical section

       amoswap.w.rl  x0 x0 a0
Fail:  #failed CAS
```



Instruction definitions:

- **L**oad-**r**eserve: Loads the four bytes from memory at address `x[rs1]`, writes them to `x[rd]`, sign-extending the result, and registers a reservation on that memory word.
- **S**tore-**c**onditional: Stores the four bytes in register `x[rs2]` to memory at address `x[rs1]`, provided there exists a load reservation on that memory address. Writes `0` to `x[rd]` if the store succeeded, or a nonzero error code otherwise.
- **Amoswap**: Atomically, let t be the value of the memory word at address `x[rs1]`, then set that memory word to `x[rs2]`. Set `x[rd]` to the sign extension of `t`.

Question: why do we need special instructions for these operations? Why can't we use normal load and store for `lr` and `sc`? Why can't we expand `amoswap` to a normal load and store?