CS 61C Spring 2022

OS, Virtual Memory, & IO

Discussion 12

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Responsibilities of the OS include loading programs, handling services, multiplexing resources, and combining programs together for efficiency.

False. While the OS is responsible for loading programs, handling services (such as the network stack and the file system), and multiplexing resources for multiple programs, it is actually responsible for isolating programs from each other so that a given program doesn't interfere with another program's memory or execution.

1.2 The purpose of supervisor mode is to isolate certain instructions and routines from user programs.

True. In the case that a program is buggy or malicious, supervisor mode limits the impact of the program on the computer, since the OS maintains control over all the resources.

1.3 User programs call into OS routines using system calls.

True. System calls, or syscalls, allow user programs to execute the OS routine in supervisor mode before switching back to user mode.

1.4 If a page table entry can not be found in the TLB, then a page fault has occurred.

False, the TLB acts as a cache for the page table, so an item can be valid in page table but not stored in TLB. A page fault occurs either when a page cannot be found in the page table or it has an invalid bit.

1.5 The virtual and physical page number must be the same size.

False. There could be fewer physical pages than virtual pages. However, the page size does need to be the same.

1.6 Polling and interrupts are only relevant concepts for low level programming.

False. Similar concepts apply to almost all types of applications, including web apps, mobile apps, distributed systems, and so on.

1.7 Memory-mapped IO only works with polling.

False. The implementation backing the memory mapping can use interrupt-driven IO (for example, reading files).

2 Addressing

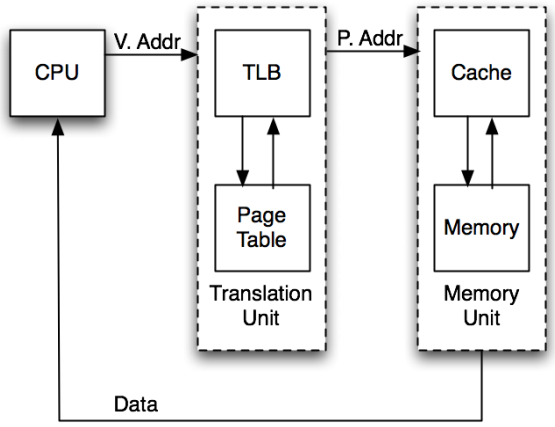
Virtual Address (VA) What your program uses

Virtual Page Number (VPN)	Page Offset

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

Physical Page Number (PPN)	Page Offset

For example, with 4 KiB pages and byte addresses, there are 12 page offset bits since 4 KiB = 2^{12} B = 4096 B.



Pages

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

Valid	Dirty	Permission Bits	PPN
— Page entry (VPN: 0) —			
— Page entry (VPN: 1) —			

Each stored row of the page table is called a **page table entry**. There are $2^{\text{VPN bits}}$ such entries in a page table. Say you have a VPN of 5 and you want to use the page table to find what physical page it maps to; you'll check the 5th (0-indexed) page table entry. If the valid bit is 1, then that means that the entry is valid (in other words, the physical page corresponding to that virtual page is in main memory as opposed to being only on disk) and therefore you can get the PPN from the entry and access that physical page in main memory. The page table is stored in memory: the OS sets a register (the Page Table Base Register) telling the hardware the address of the first entry of the page table. If you write to a page in memory, the processor updates the "dirty" bit in the page table entry corresponding to that page, which lets the OS know that updating that page on disk is necessary (remember: main memory contains a subset of what's on disk). This is a similar concept as having a dirty bit for each cache block in a write-back cache, which we covered in lecture and in Lab 9. Each process gets its own illusion of full memory to work with, and therefore its own page table.

Protection Fault The page table entry for a virtual page has permission bits that prohibit the requested operation. This is how a segmentation fault occurs.

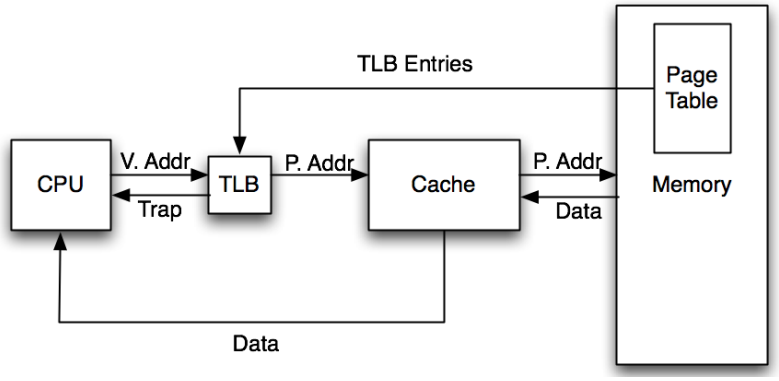
Page Fault The page table entry for a virtual page has its valid bit set to false.

This means that the entry is not in memory. For simplicity, we will assume the address causing the page fault is a valid request, and maps to a page that was swapped from memory to disk. Since the requested address is valid, the operating system checks if the page exists on disk. If so, we transfer the page to memory (evicting another page if necessary), and add the mapping to the page table *and the TLB*.

Translation Lookaside Buffer

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 Valid	Tag (VPN)	Page Table Entry		
		Page Dirty	Permission Bits	PPN
— *TLB entry* —				
— *TLB entry* —				



To access some memory location, we get the virtual page number (VPN) from the virtual address (VA) and first try to translate the VPN to a physical page number (PPN) using the translation lookaside buffer (TLB). If the TLB doesn't contain the desired VPN, we check if the page table contains it (remember: the TLB is a subset of the page table!). If the page table doesn't contain an entry for the VPN, then this is a page fault; memory doesn't contain the corresponding physical page! This means we need to fetch the physical page from disk and put it into memory, update the page table entry, and load the entry into the TLB, Then, we use the physical page and the offset of the physical address in the page to access memory as the program intended.

2.1 What are three specific benefits of using virtual memory?

- Illusion of infinite memory (bridges memory and disk in memory hierarchy).
- Simulates full address space for each process so that the linker/loader don't need to know about other programs.
- Enforces protection between processes and even within a process (e.g. read-only pages set up by the OS).

2.2 What should happen to the TLB when a new value is loaded into the page table

address register?

The valid bits of the TLB should all be set to 0. The page table entries in the TLB corresponded to the old process/page table, so none of them are valid once the page table address register points to a different page table

3 VM Access Patterns

3.1 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.

Free Physical Pages `0x17`, `0x18`, `0x19`

Access Pattern

1. `0x11f0` (**Read**)
2. `0x1301` (**Write**)
3. `0x20ae` (**Write**)
4. `0x2332` (**Write**)
5. `0x20ff` (**Read**)
6. `0x3415` (**Write**)

Initial TLB

VPN	PPN	Valid	Dirty	LRU
`0x01`	`0x11`	1	1	0
`0x00`	`0x00`	0	0	7
`0x10`	`0x13`	1	1	1
`0x20`	`0x12`	1	0	5
`0x00`	`0x00`	0	0	7
`0x11`	`0x14`	1	0	4
`0xac`	`0x15`	1	1	2
`0xff`	`0xff`	1	0	3

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

1. `0x11f0` (**Read**): hit, LRUs: 1, 7, 2, 5, 7, 0, 3, 4
2. `0x1301` (**Write**): miss, map VPN `0x13` to PPN `0x17`, valid and dirty, LRUs: 2, 0, 3, 6, 7, 1, 4, 5
3. `0x20ae` (**Write**): hit, dirty, LRUs: 3, 1, 4, 0, 7, 2, 5, 6
4. `0x2332` (**Write**): miss, map VPN `0x23` to PPN `0x18`, valid and dirty, LRUs: 4, 2, 5, 1, 0, 3, 6, 7
5. `0x20ff` (**Read**): hit, LRUs: 4, 2, 5, 0, 1, 3, 6, 7
6. `0x3415` (**Write**): miss and replace last entry, map VPN `0x34` to `0x19`, dirty, LRUs, 5, 3, 6, 1, 2, 4, 7, 0

4 Polling & Interrupts

4.1 Fill out this table that compares polling and interrupts.

Operation	Definition	Pro/Good for	Con
Polling	Forces the hardware to wait on ready bit (alternatively, if timing of device is known, the ready bit can be polled at the frequency of the device).	• Low Latency • Low overhead when data is available • Good For: devices that are always busy or when you can't make progress until the device replies	• Can't do anything else while polling • Can't sleep while polling (CPU always at full speed)
Interrupts	Hardware fires an "exception" when it becomes ready. CPU changes PC register to execute code in the interrupt handler when this occurs.	• Can do useful work while waiting for response • Can wait on many things at once • Good for: Devices that take a long time to respond, especially if you can do other work while waiting.	• Nondeterministic when interrupt occurs • interrupt handler has some overhead (e.g. saves all registers, flush pipeline, etc.) • Higher latency per event • Worse throughput

5 Memory Mapped I/O

5.1 For this question, the following addresses correspond to registers in some I/O devices and not regular user memory.

- `0xFFFF0000`—Receiver Control: LSB is the ready bit, there may be other bits set that we don't need right now.
- `0xFFFF0004`—Receiver Data: Received data stored at lowest byte.
- `0xFFFF0008`—Transmitter Control: LSB is the ready bit, there may be other bit set that we don't need right now.
- `0xFFFF000C`—Transmitter Data: Transmitted data stored at lowest byte.

Recall that receiver will only have data for us when the corresponding ready bit is 1, and that we can only write data to the transmitter when its ready bit is 1. Write RISC-V code that reads byte from the receiver and writes that byte to the transmitter (busy-waiting if necessary).

```
                lui t0 0xffff0
receive_wait:   lw t1 0(t0)
                andi t1 t1 1            # poll on ready of receiver
                beq t1 x0 receive_wait
                lb t2 4(t0)             # load data
```

```
transmit_wait:  lw t1 8(t0)             # poll on ready of transmitter
                andi t1 t1 1
                beq t1 x0 transmit_wait # write to transmitter
                sb t2 12(t0)
```