# CS 61C Fall 2023 — OS, Virtual Memory, & I/O — Discussion 12

## 1 Precheck

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, 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), 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, such as by accessing the same memory address.

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 An operating system uses context switches to allow for multiple processes to run simultaneously across multiple CPUs.

False. Context switches are used to switch between tasks on *one* CPU, so that another task can continue while one task is waiting on something else. This illusion of simultaneous execution happening in multiprogramming is different from the true parallel execution done on multiple CPUs, which is multiprocessing.

1.4 Having virtual memory helps protect a system.

True. By dedicating specific pages to a program, the OS can ensure that a program does not access pages it's not been given access to, providing isolation between programs.

1.5 The virtual address space is limited by the amount of memory in the system.

False. The physical address space is limited by the amount of physical memory in the system, the size of the virtual address space is set by the OS.

1.6 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.

1.7 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.

1.8 For I/O, polling is better than interrupts when I/O events occur regularly at a fast rate.

True. Polling is especially good for data transfer when the transfer rate is predictable and varies little, since we can set the polling rate to match the transfer rate and avoid the overhead of interrupts for each transfer.

1.9 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

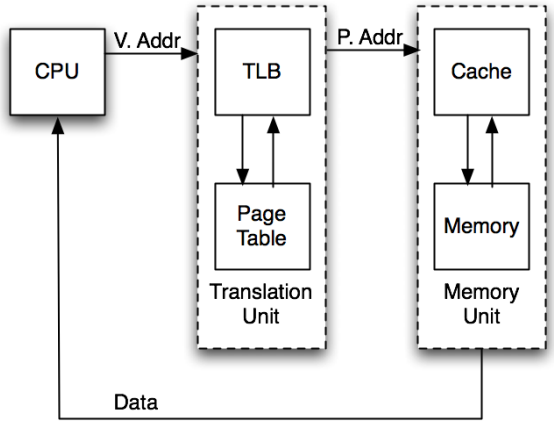
**Virtual Address (VA)** What your program uses

| 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. 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 translation lookaside buffer (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 or the valid bit = 0, it's a TLB miss. Typically fully associative:

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

2.1 What are three specific benefits of using virtual memory?

- Illusion of access to entire address space (bridges memory and disk in memory hierarchy).
- Avoids memory address conflict between programs by simulating a separate 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 (i.e. we are switching page tables to those for another process)?

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). Suppose RAM (main memory) has 12 bit addresses. At some time instant, the TLB for the current process is the initial state given in the table below, and we have one free physical page = 0x7. Suppose the least recently used physical page is 0x9. 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 following access pattern, and also write out the physical addresses corresponding to each location accessed. Indicate TLB hit or miss (if it is a TLB miss, indicate whether there is a page fault). Update the page table as needed.

**Access Pattern**

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

**Initial TLB**

| VPN | PPN | Valid | Dirty | LRU |
|---|---|---|---|---|
| `0x01` | `0x1` | 1 | 1 | 0 |
| `0x00` | `0x0` | 0 | 0 | 7 |
| `0x10` | `0x3` | 1 | 1 | 1 |
| `0x20` | `0x2` | 1 | 0 | 5 |
| `0x22` | `0xb` | 0 | 0 | 7 |
| `0x11` | `0x4` | 1 | 0 | 4 |
| `0xac` | `0x5` | 1 | 1 | 2 |
| `0xff` | `0xf` | 1 | 0 | 3 |

**Initial Page Table**

This page table does is simplified and includes the status bits valid and dirty. LRU is omitted for simplicity.

| VPN | PPN | Valid | Dirty |
|---|---|---|---|
| 0x00 | 0x0 | 0 | 0 |
| 0x01 | 0x1 | 1 | 1 |
| ... | ... | ... | ... |
| 0x13 | 0xc | 0 | 0 |
| ... | ... | ... | ... |
| 0x23 | 0x8 | 1 | 0 |
| ... | ... | ... | ... |
| 0x34 | 0xd | 0 | 0 |
| ... | ... | ... | ... |
| 0xff | 0xf | 1 | 0 |

How many bits are the offset, VPN, and PPN?

Number of offset bits = $\log_2 256 = 8$

Number of VPN bits = 16 - 8 = 8
Number of PPN bits = 12 - 8 = 4

**Final TLB**

| VPN | PPN | Valid | Dirty |
|---|---|---|---|
| 0x01 | 0x1 | 1 | 1 |
| 0x13 | 0x7 | 1 | 1 |
| 0x10 | 0x3 | 1 | 1 |
| 0x20 | 0x2 | 1 | 1 |
| 0x23 | 0x8 | 1 | 0 |
| 0x11 | 0x4 | 1 | 0 |
| 0xac | 0x5 | 1 | 1 |
| 0x34 | 0x9 | 1 | 1 |

**Final Page Table**

| VPN | PPN | Valid | Dirty |
|---|---|---|---|
| 0x00 | 0x0 | 0 | 0 |
| 0x01 | 0x1 | 1 | 1 |
| ... | ... | ... | ... |
| 0x13 | 0x7 | 1 | 1 |
| ... | ... | ... | ... |
| 0x23 | 0x8 | 1 | 0 |
| ... | ... | ... | ... |
| 0x34 | 0x9 | 1 | 1 |
| ... | ... | ... | ... |
| 0xff | 0xf | 1 | 0 |

1. `0x11f0` (**Read**): TLB hit (VPN 0x11 exists in the TLB and valid bit = 1). As shown in the TLB, VPN = 0x11 maps to PPN = 0x4. The offset bits are the rightmost 8 bits of the virtual address= 0xf0. PA: 0x4f0; LRUs: 1, 7, 2, 5, 7, 0, 3, 4
2. `0x1301` (**Write**): TLB miss (VPN 0x13 is not currently in the TLB). The page table located in main memory needs to be accessed. Reading the entry of VPN = 0x13 in the page table, the valid bit = 0. Therefore, the page is not located in RAM (page fault). Map VPN `0x13` to any existing free pages = PPN `0x7`. Set valid and dirty in both the page table and TLB = 1. PA: 0x701; LRUs: 2, 0, 3, 6, 7, 1, 4, 5
3. `0x20ae` (**Write**): TLB hit, set dirty due to write, PA: 0x2ae; LRUs: 3, 1, 4, 0, 7, 2, 5, 6
4. `0x2332` (**read**): TLB miss. Reading the VPN = 0x23 entry in the page table, we see that the valid bit = 1, so there is not a page fault. The VPN = 0x23 is mapped to PPN 0x8. We replace the row in TLB with the highest LRU with this mapping and set valid = 1. PA: 0x832; LRUs: 4, 2, 5, 1, 0, 3, 6, 7
5. `0x20ff` (**Read**): TLB hit. PA: 0x2ff; LRUs: 4, 2, 5, 0, 1, 3, 6, 7
6. `0x3415` (**Write**): TLB miss and replace last entry. From the page table, the valid bit for VPN = 0x34 is 0. Page fault. There are no more free pages, so the least recently used page with PPN = 0x9 is evicted. In both the page table and TLB, map VPN `0x34` to PPN = `0x9`, set dirty and valid. PA: 0x915; 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<br>• Low overhead when data is available<br>• 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<br>• 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<br>• Can wait on many things at once<br>• Good for: Devices that take a long time to respond, especially if you can do other work while waiting. | • Nondeterministic when interrupt occurs<br>• interrupt handler has some overhead (e.g. saves all registers, flush pipeline, etc.)<br>• Higher latency per event<br>• 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 bits 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)
```