CS 61C Fall 2021 — AMAT, OS & Virtual Memory

Discussion 10: November 1, 2021

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.

2 AMAT

Recall that AMAT stands for Average Memory Access Time. The main formula for it is:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

In a multi-level cache, there are two types of miss rates that we consider for each level.

Global: Calculated as the number of accesses that missed at that level divided by the total number of accesses *to the cache system.*

Local: Calculated as the number of accesses that missed at that level divided by the total number of accesses *to that cache level.*

2.1 An L2$, out of 100 total accesses to the cache system, missed 20 times. What is the global miss rate of L2$?

$\frac{20}{100} = 20\%$

2.2 If L1$ had a miss rate of 50%, what is the local miss rate of L2$?

$\frac{20}{50\% * 100} = \frac{20}{50} = 40\%$. We know that L2$ is accessed when L1$ misses, so if L1$ misses 50% of the time, that means we access L2$ 50 times.

Suppose your system consists of:

1. An L1$ that has a hit time of 2 cycles and has a local miss rate of 20%
2. An L2$ that has a hit time of 15 cycles and has a global miss rate of 5%
3. Main memory where accesses take 100 cycles

2.3 What is the local miss rate of L2$?

L2$ Local miss rate $= \frac{\text{Global Miss Rate}}{\text{L1\$ Miss Rate}} = \frac{5\%}{20\%} = 0.25 = 25\%$

2.4 What is the AMAT of the system?

AMAT = 2 + 20% x 15 + 5% x 100 = 10 cycles (using global miss rates)
Alternatively, AMAT = 2 + 20% x (15 + 25% x 100) = 10 cycles

2.5 Suppose we want to reduce the AMAT of the system to 8 cycles or lower by adding in a L3$. If the L3$ has a local miss rate of 30%, what is the largest hit time that the L3$ can have?

Let H = hit time of the cache. Using the AMAT equation, we can write:
$2 + 20\% * (15 + 25\% * (H + 30\% * 100)) \leq 8$
Solving for H, we find that $H \leq 30$. So the largest hit time is 30 cycles.

3 Forking

3.1 One of the many responsibilities of the OS is to load new programs, and in order to do this it creates a new process and loads in the program to execute. In Linux, the system call to create a new process is `fork()`. `fork()` creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process. In the parent process, fork()

returns the process ID of the child or -1 if the fork has failed. In the child process, it returns 0.
Use this information to complete the code block below, which creates a child process to change the value of y while the parent process changes the value of x. Assume any call to `fork()` is successful.

```
int x = 10;
int y = 0;
int pid = __________;
if(__________){
    y++
}
else{
    x--;
}
```

```
int x = 10;
int y = 0;
int pid = fork();
if(pid == 0){
    y++
}
else{
    x--;
}
```

3.2 After the code segment completes, what will be the values of x and y for the parent?

```
x = 9;
y = 0;
```

Notice that only the value of x changes. This is because `fork()` creates a new process, with a separate address space.

3.3 After the code segment completes, what will be the values of x and y for the child?

```
x = 10;
y = 1;
```

Notice that only the value of y changes. This is because `fork()` creates a new process, with a separate address space. This enforces the separation between processes that provides security within a system.

4 Addressing

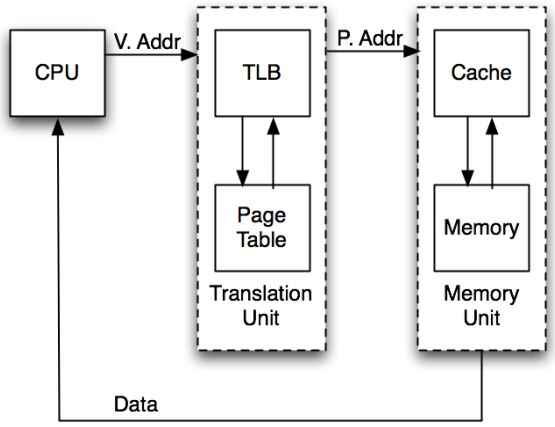
Virtual Address (VA) What your program uses

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.

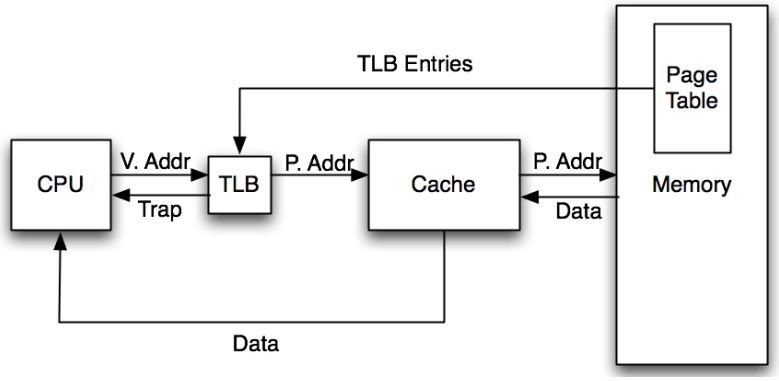
Page Fault The page table entry for a virtual page has its valid bit set to false.

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.

4.1 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).

4.2 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

5 VM Access Patterns

5.1 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