Lecture 15 – Part 1

Scribed by Tommy Chen

"[We are] swimming in the ocean of code. And keeping [our] heads above water.
[…] But we don't get there by going in from the shallow end."
- Eddie Kohler

1 Our Midterms

We started with some tips and considerations for the next test:

Treat the questions like little experiments. Rather than trying to understand every line of the code, try making educated guesses or **hypotheses** about the code and testing them out.

For example, remember the question where we matched assembly code to data structures?

BAM – Hypothesis! The length of the assembly code is directly related to the complexity of the data structure!

So let's see if the **array**, **array of pointers to arrays**, **linked list**, and **binary tree** match to assembly snippets of ascending length.

BAM – Checkin' it! Linked lists and binary trees require loops to traverse. Do those two code snippets have branches? **Yeah!**

The array code should only have one indirect memory reference. The two references to load in arguments aside, does it? **Yep!**

The array of pointers to arrays should have no loops and more than one indirect memory reference. Does it? **Sure!**

BAM – Conclusion! We can conclude with pretty good certainty that the length of assembly code is directly related to the complexity of the data structure.

2 CISCO & Pals

There are a lot of wonderful resources on the Internet on binary math, many of which related to the wonderful pursuit of becoming a CISCO certified network engineer. Here're two of them, one with an Australian accent:

https://learningnetwork.cisco.com/docs/DOC-2787

http://www.youtube.com/watch?v=WcV3nvWXVio

3 Back in Virtual Memory (VM)

We return to our 2-level page table, with which we map **virtual** addresses to **physical** addresses. Let's say our **PDP** and **PTP** are at 0x8000 and 0x9000 respectively. Keep this in mind for later:

Virtual addresses (VA)

0	1	2	3	4	5	6	7	8	9	10

Physical addresses (PA)

0x0000	0x1000	0x2000	0x3000	…	…	…	…	…	…	…
								PDP	**PTP**	
0	1	2	3	4	5	6	7	8	9	10

PTP	A **page table page** stores the mappings between virtual and physical addresses along with associated permissions.
PDP	A **page directory page** contains pointers to page table pages and stores their associated permissions. In our simple OS, our PDP only points to one PTP and works with three permissions.
Page	A block is a chunk of 4096 bytes, just as an int is a chunk of 4 bytes. It's just a useful and generally accepted size for working with virtual memory.

$$4096 = 2^{12} = 0x1000$$

Also useful to know, to convert from page numbers (PN) to addresses ($ADDR$):

$$PN \ll 12 = ADDR$$

So how many addresses (4 bytes) can be stored in a page (4096 bytes)?

$$\frac{2^{12}}{4} = \frac{2^{12}}{2^2} = 2^{10} = 1024\, addresses$$

BAM – Fun fact!

$$2^{10} \approx 10^3$$
$$2^{20} \approx 10^6$$
$$2^{30} \approx 10^9$$

4 Gettin' to the mappin'

But first, let's look at the anatomy of a virtual address (in Little Endian):

PD Index (10 bits)	PT Index (10)	Page Offset (12)
31 22	21 12	11 0

For kicks, maybe we should try mapping virtual address $0x10001456$ to physical address $0x00000456$.

BAM – Question! What is the page directory (PD) index of our virtual address?

$$0x10001456$$

Look at the diagram above! To get the PD index, we need to right-shift out the PT index and page offset – together, 22 bits.

$$0x10001456 \gg 22 = PD\ index$$

BAM – Quick and easy trick!

Remember! Every hexadecimal digit represents **4 bits**. When we shift 4 in binary, we shift 1 in hexadecimal, as in:

$$0x1234 \gg 4 = 0x123$$

So instead of shifting 22, let's think about it as shifting 20, then 2.

$$(0x10001456 \gg 20) \gg 2 = PD\ index$$

Shifting 20 means shifting 5 hexadecimal places:

$$0x100 \gg 2 = PD\ index$$
$$256 \gg 2 = PD\ index$$
$$64 = PD\ index$$

We'll do something with this knowledge soon, but first, a bit about permissions…

5 BAM – Permissions!

It's good to establish what can or cannot be done with a page of memory. Can a process access it? Is it writable? Does it exist? This kind of information can be embedded into a page address as a permission. But *how*?

An address to a page, when properly aligned, is always a multiple of $0x1000$ – the size of a page, like: $0x12000$, $0x4000$ or $0x6000$.

Look at all those 0's! In every page address, the last 12 bits (3 hexadecimal digits) are 0. This allows us to store some information there. When we want to use the actual address, we just treat these last 12 bits as 0's. This is how permissions are stored – in those last 12 bits.

So which permissions are we using in class and how are they represented?

PTE_P	Exists	1	$0b001$
PTE_W	Writable	2	$0b010$
PTE_U	Accessible to processes	4	$0b100$

BAM – Sidenote! There are lots of other permission flags too! And they're super exciting!

To embed an address with its permissions, we take the bitwise OR of an address with its permissions:

$$0x1000 \mid 0b001 = 0x1001$$

And to check if an address has a permission, use the bitwise AND:

$$0x1001 \,\&\, 0b001 = 0b001$$
$$0x1001 \,\&\, 0b010 = 0b000$$

6 Gettin' Back To That Mappin'

So back to our first problem: "For kicks, maybe we should try mapping virtual address $0x10001456$ to physical address $0x00000456$."

We already found that the page directory index of our virtual address is 64, so at the **65th cell of our page directory**, we want to point to **our page table** with **PTE_P**, **PTE_W**, and **PTE_U** permissions. We know that our page table is at $0x9000$ (if you don’t remember, scroll *all* the way to the top). Knowing all of this, let’s put it into action:

$$0x9000\ |\ 0b100\ |\ 0b010\ |\ 0b001 = 0x9007$$

So we put $0x9007$ into the **65th cell of our page directory**. Now what?

Now we go to the **2nd cell of our page table**. Why 2nd? Well look at our virtual address again:

$$0x1000\mathbf{1}456$$

In the region for our **page table index**, there is only a 1 at the rightmost bit, so our page table index is 1.

And at the **2nd cell of our page table**, we store our physical address, $0x00000$ (derived from the PD and PT indices of $0x00000456$) along with our permissions again:

$$0x0000\ |\ 0b100\ |\ 0b010\ |\ 0b001 = 0x0007$$

BAM – OH SNAP! We successfully mapped virtual address $0x10001456$ to physical address $0x00000456$.

BAM – PS!

What about mapping virtual address $0x10001789$ to physical address $0x00000789$?

This will have the **exact same mapping**. The PD and PT indices are the same. Think about the PD and PT indices as **house addresses** and think about the page offset as **the specific room in the house**. Mapping virtual addresses to physical addresses cares only about **pages**, not locations within pages.

7 Are you sure that's contiguous?

BAM – IMPORTANT!

What's so great about virtual memory? Imagine two contiguous pages in virtual memory ($ex. 0x8000\ and\ 0x9000$) mapped to two non-contiguous physical pages ($0x0000\ and\ 0x2000$). Our program will **treat these pages as contiguous even though physically they're not contiguous. You can have an array span these two virtual pages if you want to! Neat, huh?**

Bo Han
Thursday 10/25 Second half of lecture.
Topics covered: Another virtual address space example, applying virtual memory

abbreviations used:

pa →	physical address
pdp →	page directory page
pp →	physical page
ptp →	page table page
va →	virtual address
vp →	virtual page

[Time 00:00]
Example: another virtual address space for another process.

A new virtual address space requires a new page directory.
New page directory is located at physical address (pa) 0x10000

Let's make identity mappings between virtual pages (vp) and physical pages (pp).

Let the notation vp $addr mean the virtual page starting at $addr. Same for pp.

Let's add vp 0x2000 -> pp 0x2000

What goes into the 0 slot of the page directory page (pdp)?
A page table page (ptp) is necessary.
Let's create one at pp 0x3000
Let's make identity mappings for vp 0x0000 to 0x4000 and pp 0x0000 to 0x4000

pdp at pp 0x10000

Index (1024 total)	Value
0	0x3007
1 to 1023	0x0

ptp at pp 0x3000

Index (1024 total)	Value
0	0x0007
1	0x1007
2	0x2007
3	0x3007
4	0x4007
5 to 1023	0x0

[Time 04:20]
There are problems with these mappings!
Pages that should be unknown are exposed:
- index 3 of ptp at pp 0x3000 gives access to the ptp at pp 0x3000
- index 0 of ptp at pp 0x3000 gives access to data

This is a problem because this process can access/modify all physical memory!
- The process can modify its own page table (it has write access)

[Time 08:00]
Let's say the process wants to access pp 0x10000. How would it do so given the mappings above?
We want to modify index 2 of ptp at pp 0x3000. What va is that location?
    0x3008 because index 0 is at 0x3000. The values are 4 bytes, so two values beyond index 0 is 0x3008.

[Time 10:35]
Prof. Kohler makes comment possibly about previous year's CS61 professor Prof. Chong, who has an Australian accent. (intention of comment unclear)

*(uint32_t *)0x3008 = 0x10007 replaces the mapping at index 2 of ptp at pp 0x3000 (this code creates a pointer from 0x3008 and dereferences it). Changes in bold:

pdp at pp 0x10000

Index (1024 total)	Value
0	0x3007
1 to 1023	0x0

ptp at pp 0x3000

Index (1024 total)	Value
0	0x0007
1	0x1007
2	**0x10007**
3	0x3007
4	0x4007
5 to 1023	0x0

Now, the process has access to pp 0x10000 with full permissions (the additional 7 is for permission bits PTE_U, PTE_P, PTE_W). It therefore has access to its own pdp.

[Time 12:00]
Prof. Kohler: "Okay?"
Class: "Okay."

[Time 12:44]
Question: Why has there only been one ptp in the pdps we've seen thus far?
Answer:
We're dealing with small address spaces, so we only need one page table per page directory. There can be up to 1024 page tables per page directory, enough for all physical memory on a 32-bit system. However, in Weensy OS, only 2 MB (2,097,152 bytes) of physical memory needs to be accessed.

[Time 13:44]
Question: Why do we need two pdps then?
Answer:
We have two processes. Every process has a distinct address space, so we need a new pdp for each process.

[Time 14:40]

Applying virtual memory for performance
The lecture code used is in the os02 directory.

Let’s look at a cool application of virtual memory.

Let’s run the third program, a recursive program to print out the triangular numbers up to triangular number 999.

We expect that running this will print out 1001 lines, one for the hello message and the one line for every triangular number.

Actually, we get only 127 lines, with an error message from the kernel indicating a page fault caused by this process: “Write missing page from” a particular instruction pointer. What happened?
The stack pointer kept moving down and eventually ran off of the single stack page we allocated for it.

How do we solve this (assuming no modification of the source code for this program)?

[Time 19:20]
What if we allocated two stack pages? Or eight? Or nine?
We decrease memory utilization. The processes that won’t need that much stack space won’t use it. We should do stuff only when needed.

[Time 21:10]
What if we gave the process stack space only when needed?

Why don’t we use the page fault error message from the kernel as a trigger to allocate a new page for the process?

[Time 25:20]
How do we code this?

```
// We get the faulting address:
uint32_t addr = rcr2();
// page is pa of an unused page
void *page = page_alloc_unused();
// addr & ~0xFFF sets the page offset to 0
virtual_memory_map(
      current->p_pagedir, addr & ~0xFFF, (uint32_t) page, SIZE, 7
);
run(current);
```

[Time 28:26]
It works!
This is what happens every time you need to use more pages. This is an example of virtual memory for performance.

[Time 30:15]
Virtual memory is also very dangerous.

Insert some assembly code:
```
asm(“movl $0x41000,%eax; movl %eax,%esp”);
```

This changes the stack pointer to point to the kernel. This triggers a page fault, and the kernel will try to allocate a new page. The machine now reboots because there’s too many errors in the kernel.

[Time 33:00]
How do we fix this?

```
// We get the faulting address:
uint32_t addr = rcr2();

// make sure the faulting address isn’t in the kernel
if(addr >= PROC_START_ADDR) {
      // page is pa of an unused page
      void *page = page_alloc_unused();
      // addr & ~0xFFF sets the page offset to 0
      virtual_memory_map(
            current->p_pagedir, addr & ~0xFFF, (uint32_t) page, SIZE, 7
      );
      run(current);
}
```