CS 61 Lecture 17 – Oct.31

Systems Programming and Machine Organization

Ben Dickensheets, Hirohisa Yamada

The lecture code is contained in branches v01, v02. Use these branches to look at code that has been covered in lecture.
While different versions of git handle this a little differently, something like

```
git checkout -b v02
```

should let you look at the code stored at branch v02. Note that this is much better than trying to reproduce the code changes off the video.

Process Isolation

Computers consist of computer resources such as…

- CPU time
- Memory
- Disk
- Screen

It is OS's job to share all such resources among processes fairly and robustly.

Recall the three attacks in `p-hello.c` that we looked at in the previous lecture. Let's determine which one of these computer resources each attack monopolizes.

1st attack: Infinite loop in the process monopolizes CPU time

Solution: We introduced timer interrupt. This prevented CPU monopolization, by switching between processes at a certain time interval.

2nd attack: Used the cli instruction to disable interrupt. This was an attack on/ inappropriate use of the interrupt controller.

Solution: We reduced process's privilege by disallowing processes to access interrupt hardware.

3rd attack: Inappropriate modification of kernel memory

Solution: Virtual memory protected memory region that contained kernel from processes that are not kernel.

(Fun fact: Macintosh OS9 did not have kernel-protected memory!)

Privilege

'Dangerous' hardware operations may be executed only by privileged code.
We need to tell the hardware whether the current running code is privileged.

The privilege information is stored in a particular register. This is a special register that requires privilege to change (if not, a process would be able to bump up its privilege arbitrarily right?)
In x86 architecture, the low 2 bits of `%cs` register contains this privilege information.

0 -> kernel privilege (always have full control of the machine)
1 -> (not used)
2 -> (not used)
3 -> process privilege
(1 and 2 were initially designed to offer intermediate levels of privilege, but turned out to be not all that useful)

Running gdb code `info registers` will show contents of register. Check for yourself that when in kernel, the lower bits of `%cs` register is 0!

Now we look at how processes and kernels interact with each other. Namely,
How can process contact kernels?
How can kernel drop privilege and run process?

Here is a simple solution: why can't we just have unprivileged processes call kernel functions like SYS_YIELD?

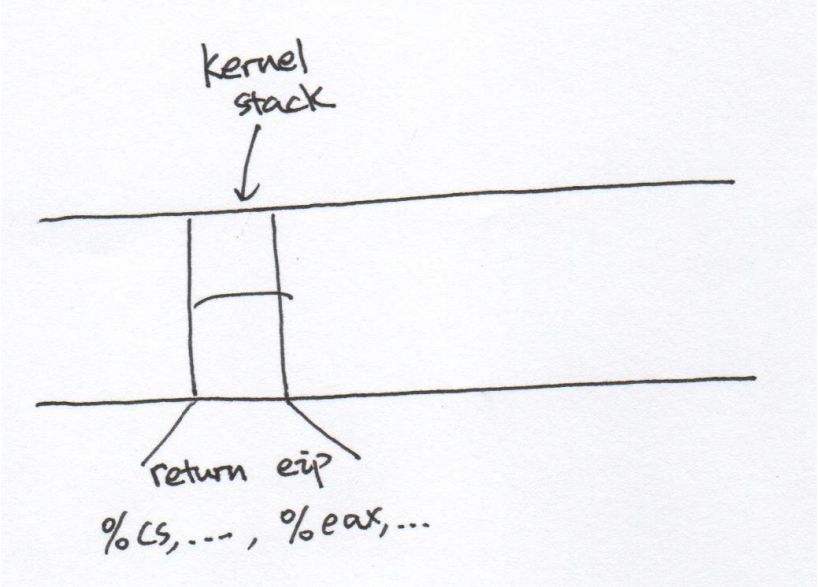
…because the kernel contains a lot of code. Some of which are intended to respond to user access, while some are system calls that are not intended to be exposed to users. With this naïve architecture, we cannot prevent processes from calling 'dangerous' system calls, and attacks like disabling timer hardware can occur.

Process -> Kernel

Kernel installs **exception vectors**, which is an array of entry points for exceptional control transfers. A special call mechanism is setup for these entry points.

The system call that handles this is `int $N,` which activates element N of the exception vector.

What's different from normal function calls is that the kernel cannot trust the user stack. Here's what the memory looks like in OS01:



When `int $N` is called, current register values such as `%esp %eip %cs`, and the return address of the call are stored on kernel stack, and then privilege is switched to kernel privilege.

Question: why is `%cs` copied in? Because when we enter kernel code, the processor changes the privilege and thus the `%cs` is also changed. Therefore, we must store the old value of `%cs` must be stored somewhere.

Kernel -> Process

Now, how do we return to the process? Can we just make a return call? No, this won't change the privilege back to process privilege. The system call `iret` takes care of this by restoring `%esp %eip %cs` from the kernel stack
This can be seen in action in the `schedule` and `run` function is OS01.

How expensive are interrupt functions? -> they are pretty expensive because we must change more registers than in regular functions.

Exceptional Control

So we know that there are three types of exceptional control: traps, interrupts, and faults. To remind you, here are some definitions:

Trap: A system call that a process wanted to make—the program tells the kernel "do this for me now"

Interrupt: Exceptional control transfer made by hardware (think interrupt controller!)

Fault: Exception caused by a mistake that the program made.

Question: What is the kernel-stored return `%eip` for each of these exceptions?
For traps and interrupts, the return `%eip` (remember that the return `%eip` tells the processor where to start executing instructions when it returns to the process) is the address of the next instruction in the process. However, the `%eip` stored for a fault is the address of the instruction that caused the fault. We can examine why this should be the case in the following example:

Looking at version 9 of os01, we examine a particular case: the process causes a page fault (a page fault is a virtual memory error) by trying to access restricted memory and the kernel prevents this access and regains control from the rogue process. (This is an important point about correct process isolation and exceptional control transfer: if a process ever tries to violate its privilege or otherwise act improperly, the kernel gets control.)

When the kernel regains control, it retains a copy of the process state in the variable `current`. This copy contains (among other things) a complete copy of the registers *immediately before the faulting instruction is carried out*. The processor has essentially rewound to where it was just before executing the faulting instruction and tells the kernel that if the instruction is carried out, it's going to cause a problem. This is **super awesome**

because it allows the kernel to try to fix the error and restart the program as if nothing happened!

As a side-note the memory map function takes advantage of this sort of control transfer to load information into memory *as needed*—if the memory isn't loaded yet, the process faults and the kernel is able to load the required memory and restart the calling process.

In our example, the kernel can use the information provided by the processor to simply skip the offending instruction and carry on as if nothing wrong happened.

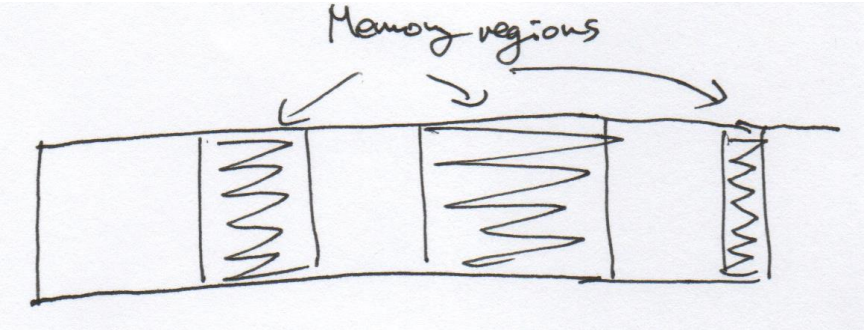
This is also a prime example of nasal demons, since the kernel is allowed to do *absolutely anything* if the process violates c language specifications.

Virtual Memory

This is one of the most powerful ideas concerning the software hardware boundary.

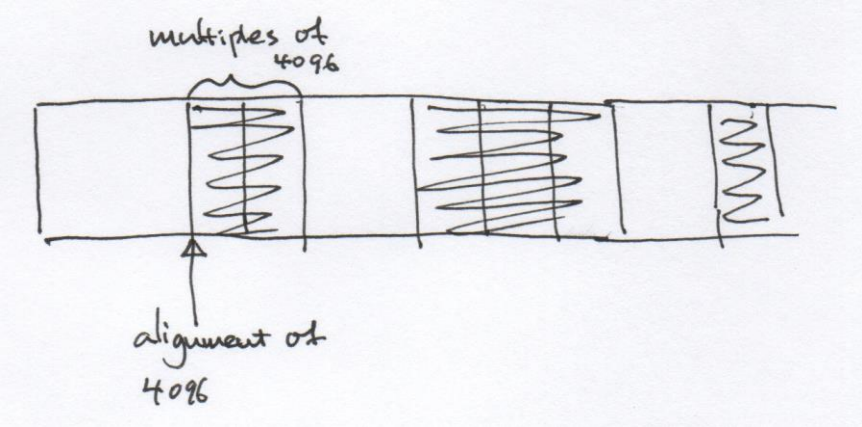
Let's try to implement a virtual memory system of which goal is to protect kernel memory from unprivileged access.
Mark memory regions as either 'accessible only by kernel' or 'accessible by both process and kernel' (there's no such thing as regions 'accessible only by process', since kernel has strictly more privilege).

Let's look at memory regions…

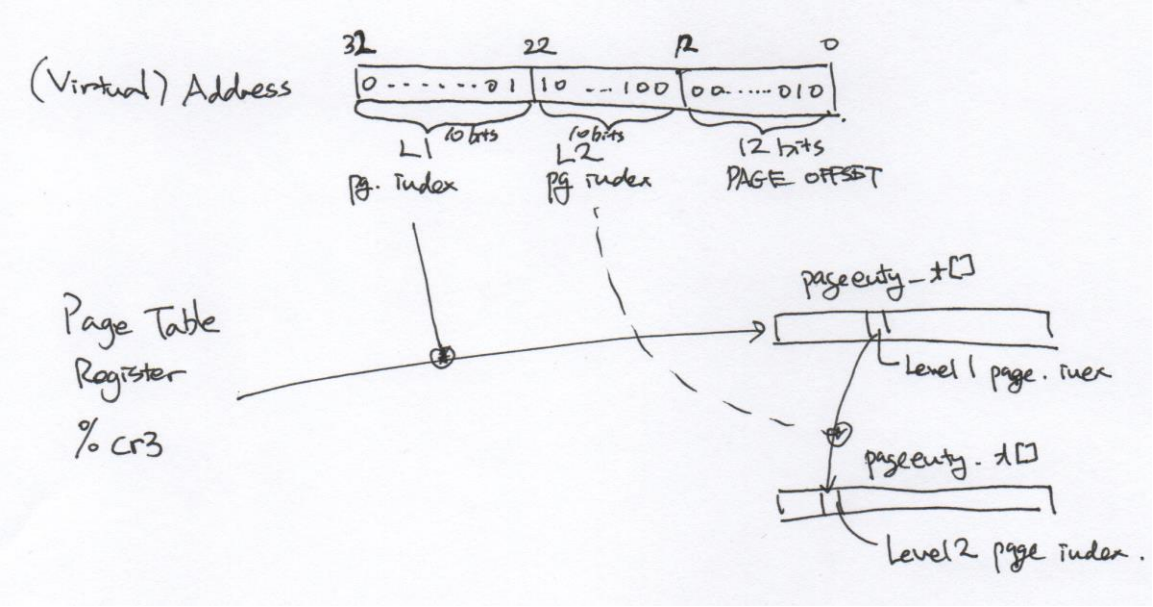


Are there any restrictions on such memory regions? Suppose we wanted 0x01 to be kernel only, 0x02 – 0x05 to be accessible by process… etc. How much memory will we need to represent the memory? Same amount as the memory itself!

We want a compact representation of memory regions. In order to achieve this, we restrict memory regions so that we can only talk about consecutive 4096 aligned bytes. Each of these regions are aligned on 4096 bytes. Each of these regions is called **pages**.



Thus we can reduce the size of memory representation by a factor of 4096. (For instance, 4GB region can be reduced to 1MB). However, the hardware attempts to reduce even further by implementing a **2 level page table.**



Let us divide the 32 bits in an address into three. The right most 12 bits will refer to page offset, next 10 bits as level 2 page index, and the last 10 bits as level 1 page index.

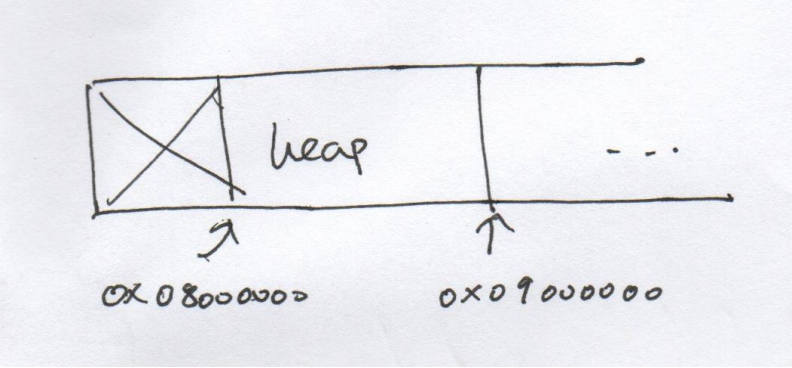
When a virtual address is dereferenced, first it looks at the Level 1 page index, and accesses that index on Level 1 page table. The entry in the page table will point to a Level 2 page table, so now we look at the Level 2 page index, and access that index in the Level 2 page

table. This entry will point to a page in the physical memory, so finally we look at the page offset, and access the data at that offset in the page.

Note that entries in the page table will have NULL, if the process does not use that region. This allows saving up memory, which is the strength of having a two level page table.

The weensyos has memory address in the range 0x00000000 – 0x002FFFFF. Top 10 bits of both 0x00000000 and 0x002fffff are 0. Therefore, there is only one entry in L1 page table, and thus, only one L2 page table. Therefore, in order to represent all the memory space available in weensyos, you need two pages of page tables (one for L1, and one for L2).

Let's go back to Linux. Following is a typical memory map for a linux system:



How many entries in the L1 page table do we need to represent the heap region in this memory map?

Again, we look at the first 10 bits of the lower bound of heap, 0x08000000, and the upper bound of the heap 0x08ffffff, which are respectively 0x20 and 0x23. So, we need 4 entries in L1 page table to represent this region.