Administrivia
- 2 readings for Thursday
- HW1 due 9/16
- Scribe: Adrienne

Motivating scenario for work on secure operating systems:
- One machine, many users who don't necessarily trust each other.
- Want some kind of security mechanism that will protect you from each other.

Motivating example:

2 user system, Alice and Bob
Alice uses computer once, then Bob uses it after that

<u>Confidentiality</u>
- Bob can spy on Alice's left-behind files
- Alice can spy on Bob (keylogger)

<u>Integrity</u>
- Alice can modify bootloader, SW, etc. so Bob doesn't get OS that he expects

<u>Availability</u>
- Alice could deny Bob service

Two lessons:
1. Whoever gets to the computer first wins
2. OS is critically relied upon

Two solutions:
1. Alice and Bob get their own computers
    a. Valid as long as you're willing to pay for *n* computers for *n* users
    b. This is the abstraction for the ideal, gold-standard OS: multi-user systems should be as secure as if the users had their own machines
2. One computer (in a cave, i.e. no network access), and one trusted operator. Alice and Bob don't have physical access to the machine; whenever they want access to it they have to talk to the operator, who will wipe the computer and perform Alice's computation for her.

Refresher on memory management

When the system boots, it sets up MMU-to-physical system mappings so that the OS code (kernel) is loaded into one portion of the memory address space and users are loaded into another. The virtual memory mappings can only point to the appropriate portion of RAM – so if Alice uses virtual address X and the OS also uses virtual address X, the two X's will actually point to different places in physical memory.

If you want to run multiple user mode programs, you need to have multiple memory mappings. Also need a scheduler and a timer.

Interrupts transfer control from user mode to supervisor mode, where the OS can access the kernel, do scheduling, etc.

Paging: if Alice wants to fill up RAM, Bob's RAM contents will be paged out to lower-tier memory and then swapped back in when it's Bob's process's turn again

<u>Isolation-only model</u>:

Confidentiality:
- CPU usage leaks; can see the difference between a process running alone and it running with others on the system
- Bob shouldn't be able to see Alice's paged out memory or vice versa
- If Alice gets scheduled often enough, she could be able to infer Bob's page accesses / memory layout
- Prisoner wall-banging

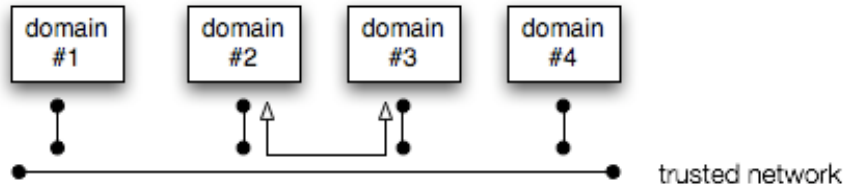
Integrity:
- Yes, Alice can't even mention Bob's portion of memory, because all of the memory addresses that she can mention will be mapped to her own space in RAM
- Caveat: need hard drive protection too…otherwise files could be tampered with before they even get in to RAM

Availability:
- Yes, barring loss of CPU access during scheduling

<u>Controlled sharing</u>: OK, now we want to add in the ability to communicate in a controlled fashion

<u>Protection domain</u>:
- (public) code + (private) state
- code is protected even though it's public – shouldn't be modifiable by external entities
- have one outside entry point, like in the OS…kernel has its own data structures with a single public entry point, which is done by triggering an interrupt. CPU then switches to supervisor mode so that OS can go inside and modify its private state

<u>Message passing between domains</u>:



- Only way to communicate between domains is through the trusted network's message passing system (a bus network)
- Trusted referee knows all of the domain's identifiers to deliver messages
- The trusted network should guarantee no eavesdropping, and integrity of messages should remain intact (no tampering)

- Provides mutual distrust

Object-oriented programming: object = protection domain
Sep. kernel:
Clark-Wilson security model: think of your system as protected subsystems plus some invariants that each protected subsystem maintains. This came out of the banking world…a subsystem for each thing that the bank does (like, one maintains bank accounts, another does conservation of money)

So how would you implement message passing on protected domains in an OS:

Sendmsg(recipent ID, data)
- Sender will invoke system call (sendmsg) with recipient ID and data
- An interrupt will invoke supervisor mode
- Kernel can then get the data from the sender's memory and put it into the receiver's memory
    - Copy the data and put the copy in the receiver's memory
        - Bad for performance
        - Could cause accounting problems (DOS?) for the receiver
    - Map the sender's memory into the receiver's memory
        - TOCTTOU problem (time of check to time of use) – what if the sender changes the data before the receiver gets to see it? (Race condition)
        - Change the way we do the mapping – change the sender's address space to be read-only so that they can't access it afterwards; or copy on write, so the OS will only make a copy once the sender has modified the data
        - Need to be careful about mapping: what if you map a page, but the actual message size is only two words? Then the receiver gets too much data, and unrelated memory mysteriously becomes write-only
    - In practice, a copy somewhere is what's used (to deal with mutual distrust between sender and receiver, accounting issues)