# Lecture 17 (November 14)

- Fall 2017 Midterm Discussion
    1. Execve, does not create new process
    2. Mapping in dynamically linked libraries
    3. Hard links maps filenames to inodes
    4. Sleep until the consumer wakes it
    5. Using type conversion, read returns number of bytes written. We must convert back to the number of unsigned long, minus one because maximum index of array is one less
    6. Terminate a process, find out when a child process is terminated, start and pause a process.
    7. Contain virtual address, memory of a process is a virtual address space.
    8. Yes, initial memory is allocated using execve then use mmap. Kernel is responsible for managing memory, communicate with kernel with system call.
    9. When a program runs, it assumes stdin, stdout are set up properly.
    10. If mmap file into memory, it can allocate entire size of file, put alot of pressure of memory system. Read system call can go through the file in chunks and use less memory
    11. No, someone can modify between the while loop and the decrement, need special instructions that test and initialize at the same time. Semaphores are magic and can not be implemented in regular C code. (Something like fork and execve will appear on the final exam)

- How do you do kernel hacking?
    - Be humble
        - Don't know how everything works and nobody else does
    - Verify your assumptions as you go
        - Perform lots of experiments
        - Incremental development, try to test every line to see if it does what you think it does, compile and run very often.
    - Check for errors **ALWAYS**
        - will save you from later errors
        - kernel has to "live cleanly"
    - Applies to any large scale coding
    - Find another part of the kernel that is close to what you want to do. Read that code, use their approach where possible
        - Functionality that you want to implement might already exist in the kernel, look at how they do things and follow the pattern.
        - You don't understand all the abstractions and assumptions, so "pattern match" to minimize trouble
        - Refine understanding as you go, implementation might be for something else
        - Realize that the assumptions behind code you find don't necessarily match your own.
    - Understand the "flow of control" inside the program
        - in order to manipulate a large program, you might not understand the little abstractions, but

        have to have an idea of the basic flow of control in the program
        - understand architecture
        - division of responsibilites
        - purpose of data structures/objects
    - Looking at getnewpid.c
        - Why ask to do specific operations for question 4
        - Wanted us to start navigation in /kernel/sys.c
- Scheduling
    - Scheduler decides when something should run, process can only run on one core at a time.
    - Resource allocation problem, who gets to run when?
    - How does kernel control when a program runs?
    - Make sure one program doesn't take over everything?
    - How does the kernel maintain control?
        - CPU is given entirely to userspace processes to run most of the time
    - Kernel needs to run when it needs to run
        - and no process should be able to stop it
    - *The interrupt table*
        - pointers to code that the CPU runs when it receives different hardware or software interrupts
        - since the kernel runs first, it gets to set the interrupt table
        - only supervisor code can change the table, not user code
    - So when the ethernet card receives data...
        - the ethernet card sends an interrupt to the kernel
        - the CPU calls the kernel code for handling ethernet data
    - When the clock generates a timer interrupt...
        - The CPU calls the kernel code for handling timer interrupts
        - because there is a timer, after a certain time, the kernel gets woken up
    - In order to process an interrupt, an CPU core has to be taken over
        - that core was probably running a userspace process
    - Scheduling is all about what to do after having kicked a userspace process off of a core
    - Normally on a core
        - userspace process is running
        - interrupt happens
        - core switches to supervisor mode, runs kernel code
        - last part of kernel code is scheduler, chooses which userpsace code to run
        - go back to top
    - Kernel is entered via interrupts, exited via the scheduler
    - Scheduler determines whats the next task to do
    - Entry and exit to the kernel has to do low-level tasks like changing CPU modes, manage CPU registers
        - must be written in assembly
    - Most OS kernels minimize this low-level code
    - What criteria should the scheduler use in deciding what to run?

        - "Fairness"
            - Everyone should get a turn
            - CPU is passed around
            - Nobody should be left that they never get the CPU
            - "starvation" not getting a turn to use the CPU
            - Prevent starvation
            - Absent other conditions, equal allocation of resources
        - Bias resources towards "foreground" tasks in interactive system
            - never biased enough
            - always hacks, heuristics
    - Memory overcommitment
        - it is possible to allocate way more memory than can be ever used
        - goes into "memory debt"