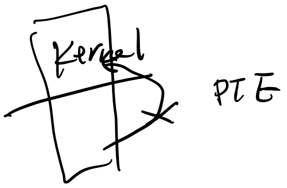
```
** you have one cheat sheet; so questions will not repeat common knowledge
   ** common cmds: ls, touch, cat, echo, cd, mkdir
** you will be asked to explain things; don't rambling; we can reason things


4. virtual memory

  -- what is VM?
     VA => PA

  -- benefits
     (a) programability ("transparency")
     (b) protection
     (c) effective use of resources

  -- translation, paging
    -- multiple ways, but "paging" dominates           4KB
    -- idea: divide memory into pages
    -- [VPN][offset] => [PPN][offset]

  -- VPN => PPN
    -- multiple ways, but "multilevel page table" dominates
    -- x86-64 page tables:
      -- VA: [ 9 | 9 | 9 | 9 (VPN)] [ 12 (offset) ]    48bit
      -- root (cr3)
      -- each 9-bits serves as an index into L1/2/3/4 page tables
      -- page offset (12 bits) points within the final page
    -- x86-32 page tables:
      -- VA: [ 10 | 10 (VPN)] [ 12 (offset) ]    32bit

  -- page walk ⇐ IMPORTANT

  -- page table entry
    -- many useful bits
    -- P, R/W, U/S  (remember these to understand PTE)
      --meaning of P=0/1, R/W=0/1, U/S=0/1
    -- D, A, ... (set by hardware + cleared by OS)

  -- performance
    -- one page walk has to load multiple pages (why?)
    -- accelerate by TLB (translation lookaside buffer)
      -- a cache, with mapping of VPN to PPN
    --cache structure (tag and index; L1 cache lookup trick)

  -- where does OS live?
    -- in every process's address space
    -- but process's code cannot access (why?)
    -- not entirely true now (post-Meltdown)
      --kernel has its own address space
```



```
  -- spectre and meltdown
    -- bypass hardware isolation check
    -- based on sepculative/out-of-order execution + side channel attack

  -- page fault
    -- translating VA to PA "fails"
    -- triggers:
       either because it's not mapped in the page tables or because
       there is a protection violation.
    -- how it works:
      -- CPU prepares "trap frame"                exception
      -- CPU transfers control to kernel's...
      -- ...page fault handler
      -- register cr2 has the faulting VA

  -- page fault usage
    -- many, name a few:
    -- overcommitting memory (canonical usage)
    -- distributed shared memory
    -- copy-on-write <-- fork()

  -- page replacement policy
    -- "cache eviction problem"
    -- many lovely algorithms, but too expensive to use
    -- one that simulates LRU but easy to implement: CLOCK
      -- will use hardware-assistant bit in PTE (which one? Access bit)
    -- thrashing

  -- mmap
    -- a systemcall
    -- map a chunk of file directly into memory space
    -- fast and easy to use (memory's interface)
    -- need supports from VM system
                                                   5x - 100x

5. I/O, Disk, SSD

  -- Port-mapped I/O (PMIO)
    -- x86 instructions: inb, inw, outb, outw, ... --> I/O address
    -- example: read keyboard inputs (handout)
    -- example: set blinking cursor (handout)

  -- memory-mapped I/O (MMIO)    movq  --> Pysical address
    -- memory interface (read/write)
    -- connect to device registers under the hood
    -- example: VGA, write to screen (handout)

  -- polling vs. interrupts
    -- polling: CPU busy queries status
    -- interrupts: hardware signals the CPU when status changes
    -- pros vs. cons

  -- DMA vs. Programmed I/O
    -- Programmed I/O: PMIO and MMIO
    -- DMA: direct memory access
        -- better way for large and frequent transfers
        -- usually go with interrupts but do not have to
    -- {DMA, programmed I/O} x {polling, interrupts}

  -- Disk
    -- geometry of a disk (handout)
      -- platter, cylinder, track, sector
      -- how it works
    -- performance characteristics

  -- SSD
    -- how it works (handout)
      -- flash bank, block, page
      -- operations: read (a page), erase (a block), program (a page)
      -- bummer: wear-out
      -- FTL: flash translation layer
      -- a log-structured FTL
    -- performance characteristics
```

```
6. File systems

  -- fs goals:
    -- persistence
    -- name a set of bytes (file)
    -- easy to remember where are files (directories)
       [now it's unclear if directories is the right way to do this]

  -- files
    -- map name and offset to disk blocks:
       {file,offset} --> disk address
    -- analogies to virtual memory
    -- file mapping types
      -- contiguous allocation
      -- linked files
      -- indexed files (Unix)

    --Unix inode:
       metadata (permissions, times, link count)
       ptr 1  --> data block
       ptr 2  --> data block
       .....
       ptr 11  --> indirect block
                    ptr -->
                    ptr -->
       ptr 12 --> indirect block
       ptr 13 --> double indirect block
       ptr 14 --> triple indirect block

  -- Directories                                          (block#2)
    -- goal: find a file
    -- Hierarchical directories (Unix)
    -- directory stored on disk just like regular files
      -- difference? metadata (mode) and file contents (name=>inode)
    -- links                                               str    int
      -- hardlink (cannot have cycles; why?)
      -- softlink (can have cycles)

  -- CS5600 FS, fs5600
    [see Lab4; I will skip here]

  --crash recovery
    --we want fs to be *consistent* even after crashes (e.g., power failure)
    --ad-hoc: fsck (unclear guarantee)
    --COW fs (always consistent on disk)
    --journaling (different levels of consistency: metadata vs. data)
      --redo logging
      --undo logging  -->       1. begin txn

7. Security                    2. Writing to Jouranl  -> block#2 (Journal)
                                     + checkpointing  -> block#21
  -- many topics

  -- authentication
    -- three ways to do: based on what you know, what you have, and what you
are
    -- how we store passwords today
    -- fancy authentication

  -- access control
    -- subj -[access]-> obj: pass or reject
    -- Unix: UID/GID, process, file
```

```
      -- process's UID
      -- file's permission bits: rwxrwxrwx
      -- dir's permission bits: rwxrwxrwx
    -- setuid
      --motivation: need to raise privilege level sometimes (like updating
password)
      --setuid bit: run a program on behalf of its owner
      --dangerous, many attacks (see notes)

  -- stack smashing: buffer overflow
    -- stack frame & calling convention
    -- attack (handout)
    -- defenses
      -- nonexecutable stack (NX)  [attack: ROP, BROP]
      -- stack canaries
      -- address randomization


---

1. processes

- an abstraction of a machine

- process manipulation
  -- create process: fork()
    -- almost an exact copy of parent, except %rax (what is this?)
    -- kernel internal data structures get copied as well (how about fd?)
    -- why not createProcess()?
  -- fork/exec separation
  -- parent and child
    -- orphan process and zombie process

- process's view
  -- memory
    -- [draw the memory view of a process]
    -- code, data, stack, and heap
    -- in C, what variables are in which memory area?
  -- registers
    -- %rip
    -- %rbp, %rsp
    -- others
  -- file descriptors (defer to later)

- assembly code (x86)
  -- pushq %rax
  -- popq %rax
  -- call 0x12345
  -- ret

- calling convention
  -- where are func arguments and where is the return value
  -- call-preserved & call-clobbered

-- stack frame and how function call works
  -- [saved %rbp; local variables; call-preserved regs; %rip]
  -- how it works?

-- process-kernel interaction: trap to kernel
  -- system calls
  -- exceptions
  -- interrupts
-- system calls
  -- an interface between processes and kernels
  -- how to know a systems call's function?
    -- man 2 <syscall>
  -- important system calls:
    -- fork, execve, wait, exit
    -- open, close, read, write
    -- pipe, dup2
  -- (briefly) how syscall is implemented

-- file descriptors
  -- an abstraction: a file, a device, or anything that follows open/read/write/close
  -- 0/1/2: stdin/stdout/stderr
  -- redirection and pipe

-- shell
  -- an interface between human and computer
  -- how it works; Lab2
    -- parse commands
    -- run commands (fork/exec)
    -- handle shell operators
      -- redirections
      -- pipe
      -- connecting ops (;, &&, ||)
      -- subshell

-- threads
  -- two threads in the same process share memory
     -- meaning: they share code segment, data segment (i.e., global
        variables), and heap (i.e., memory from malloc)
  -- threads have separate stack and registers
  -- (in the second half) kernel-level threads vs. user-level threads


2. Scheduling

-- OS scheduling problem
  -- kernel needs to decide which process to run
  -- when does kernel need to make this decision?
    -- process state transition graph
       (ready, running, waiting, terminated)
    -- non-preemptive scheduler: running->terminated or running->waiting
    -- preemptive scheduler: all four transitions
  -- scheduling another process
    -- context switch
    -- it has a cost

-- scheduling metrics
  -- turnaround time
  -- response time
  -- fairness

-- CS5600 scheduling game
   -- multiple processes: P1, P2, P3, ...
     -- each with arrival time and running time
   -- 1 CPU
   -- ignore context switch costs
   -- assume no I/O
   -- note: processes arrive at the beginning of each time slot

-- six classic scheduling algorithms
  -- FIFO/FCFS
```

  -- SJF/STCF
  -- RR
  -- Priority
  -- MLFQ
  -- Lottery

-- beyond our scheduling game
  -- incorporating I/O into scheduling
  -- predicting future (simulating SJF/STCF)
  -- complicated scheduling in practice (Reconfigurable Machine Scheduling Problem)
  -- scheduling has a lot to do with others factors, for example, mutex

3. Concurrency

-- thread APIs
  -- thread_create(function pointer, args)
  -- thread_exit()
  -- thread_join(thread id)

-- safe path: how to write concurrency code
  -- a systematic way to address concurrency problem
    -- this is one way, but is a good way, probably the best way that we're aware of

  -- Monitor (mutex + condition variable)
  -- 6 rules from Mike Dahlin (memorize them!)
  -- four-step design approach

  -- Mutex: providing mutual exclusion
    -- APIs
      -- init(mutex)
      -- acquire(mutex)
      -- release(mutex)
    -- providing a critical section
      -- entering c.s.: "I can mess up with the invariant"
      -- leaving c.s.: "I need to restore the invariant"

  -- Condition variable: providing synchronization scheduling
    -- APIs
      -- cond_init(cond)
      -- cond_wait(cond, mutex)
      -- cond_signal(cond)
      -- cond_broadcast(cond)
    -- an important interface, cond_wait(...)
      -- meaning: 2 steps,
         (1) unlock mutex and sleep (until cond signaled)
         (2) acquire mutex and resume executing

  -- other synchronization primitives
    -- semaphore
    -- barrier
    -- spinlocks
    -- reader-writer locks
    -- read-copy-update (RCU)
    -- deterministic multithreading

-- dangerous concurrency world: concurrency problems

  -- if not sequential consistency
    -- hard for human to reason
    -- cases in handout week5.a panel 4

```
  -- concurrency problems under sequential consistency

    -- no synchronization:
      -- multiple threads working on some shared state
      -- atomicity-violation (race conditions)
         [examples: linked list and producer/consumer in handout week5.a]
      -- order-violation

    -- with incorrect synchronization
      -- deadlock
        -- well-studied problem (and intellectually challenging!)
        -- four conditions
            1. mutual exclusion
            2. hold-and-wait
            3. no preemption
            4. circular wait
        -- solutions
      -- livelock
      -- starvation
      -- priority inversion


notes about exam
---

-NOTE: remember your NUID
  --you need to write down your NUID
  --your husky card doesn't have your NUID (WHY??)

- a lot to read, not much to write

- 11 pages; A4; one-sided
  -- stapled
  -- feel free to disassemble (easier to read code and answer questions)
  -- if you do so, remember to fill in your name and NUID in the bottom of
each page

- you will need to write some code on paper
  -- we tolerate minor syntax errors
  -- you can always write down your assumptions

- assume you remember "common knowledge"
  --you now have a cheat sheet
  --example, "echo abc > /tmp/file1"
```

• Program ← VA.

2^{33} bits

Northeastern University
CS5600: Computer Systems: Spring 2022

Final Exam

- This exam is **120 minutes.** 1:30
- Stop writing when “time” is called. *You must turn in your exam; we will not collect it.* Do not get up or pack up in the final ten minutes. The instructor will leave the room 123 minutes after the exam begins and will not accept exams outside the room.
- There are **11** problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.
- **This exam is closed book and notes. You may not use electronics: phones, tablets, calculators, laptops, etc.** You may refer to ONE two-sided 8.5x11” sheet.
- If you find a question unclear or ambiguous, be sure to write any assumptions you make.
- Follow the instructions: if they ask you to justify something, explain your reasoning and any important assumptions. **Write brief, precise answers. Rambling brain dumps will not work and will waste time.** Think before you start writing so that you can answer crisply. Be neat. If we can’t understand your answer, we can’t give you credit!
- If the questions impose a sentence limit, we will not read past that limit. In addition, *a response that includes the correct answer, along with irrelevant or incorrect content, will lose points.*
- Don’t linger. If you know the answer, give it, and move on.
- **Write your name and NUID on the document in which you are working the exam.**

Do not write in the boxes below.

I (xx/18)	II (xx/6)	III (xx/10)	IV (xx/12)	V (xx/21)	VI (xx/20)	VII (xx/13)	Total (xx/100)

Name:

NUID: