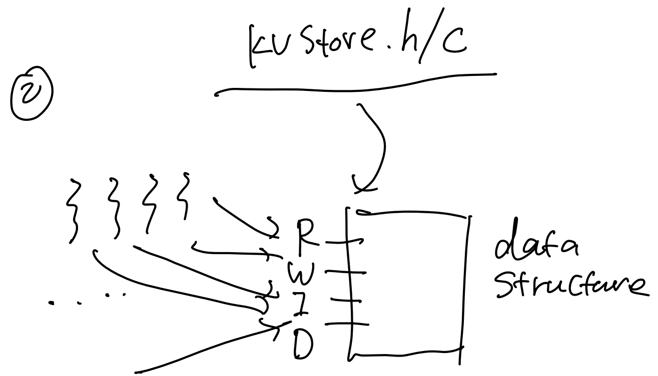
Week 8.a
CS3650
02/26 2024
https://naizhengtan.github.io/24spring/

1. Lab3
2. Survey
3. Midterm review
4. Q&A

• Lab3.

① yes → slack.txt



Leaderboard

Search

Rank	Submission Name	Time taken (in seconds)
1	Michelle	0.04
2	Nolan	0.1
3	TCA99	0.27
4	Eric Sun	0.39
5	tom	11.64
6	Griffin Cooper	12.06
7	sus	13.42
8	Kenneth	31.06
9	sb	31.89
10	CS3650 Baseline	59.6

+20 points

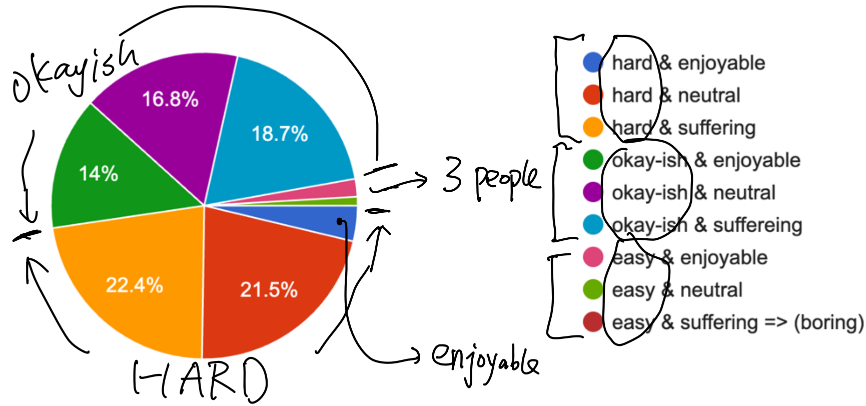
>1000x

60s

Survey results for the session (Week1—6):

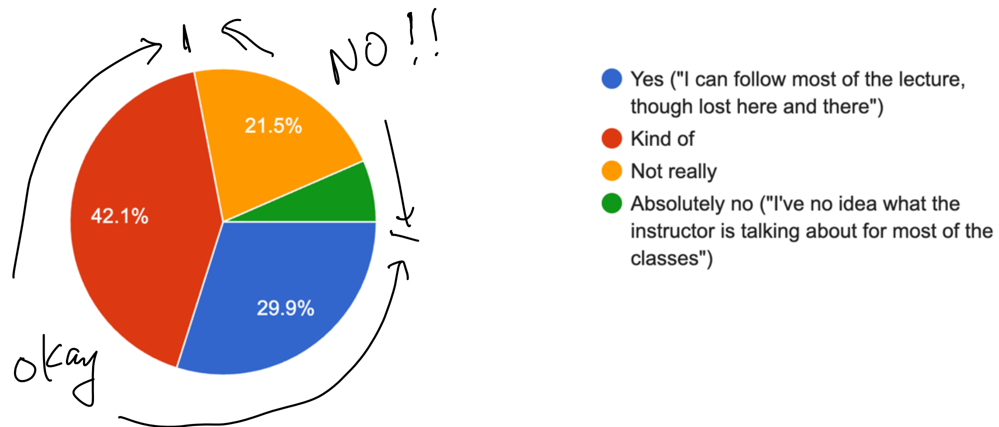
Choose your current feeling about CS3650:

107 responses



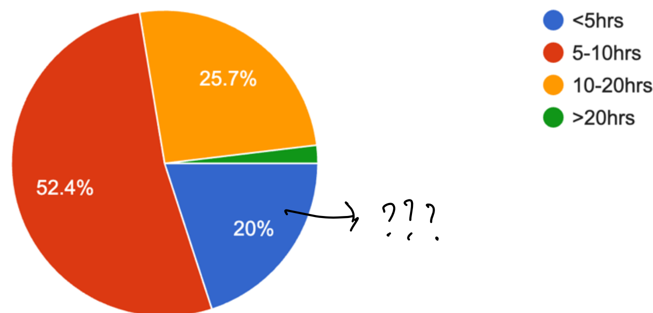
Do you agree, "I get the necessary background to follow lectures"?

107 responses



How many hours on average do you spend on CS3650 each week?

105 responses



CS3650 Midterm review

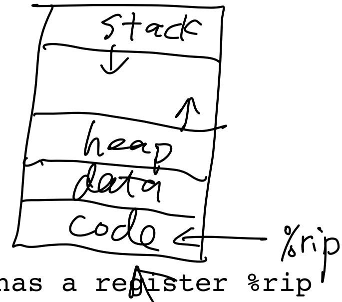
Overview
  1. computer organization & C
  2. processes
  3. concurrency & synchronization
  - Lab2: syscalls & shell
  - Lab3: concurrency programming

1. computer organization & C

- everything is 0s and 1s
  -- 1 byte == 8 bits
  -- integer: singed vs. unsigned; two's complement
  -- capacity: KB, MB, GB, TB, PB
  -- 1GB DRAM != 1GB disk

- OS
  -- providing services to user-level programs
  -- managing the resources
  -- abstracting the hardware

- hardware abstraction
  - CPU
    -- ALUs and registers
  - memory: an array of bytes
    -- address: index to the array
  - disk: an array of blocks



- how a program (helloworld) runs?
   * a file on disk
   * load to memory; have code and data segmentes
   * the code segment is a sequence of instructions
   * CPU needs to run the instructions
   * how does CPU know which instructions to run? CPU has a register %rip
   * one way to see how a CPU works:

```
     while (true) {
       instruction <- fetch from memory pointed by %rip
       execute instruction
       %rip <- address of next instruction
     }
```

- a life cycle of a program

- C basics
  -- control flow
  -- functions and scope
  -- types and operators
     -- char, int, float, double
     -- precedence and associativity
  -- expectation: fluently read C code and write C code with minor syntax errors

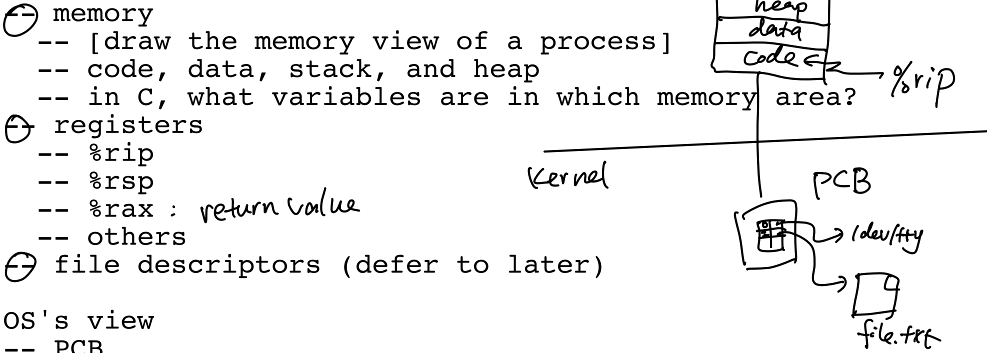
- memory manipulation in C
  -- a pointer = a memory address
  -- C strings: a char array with terminating '\0'
  -- C arrays
  -- struct and malloc/free
  -- printf
  -- main and arguments

2. processes

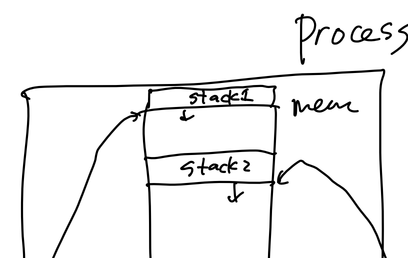
- an abstraction of a machine

- process manipulation
  -- create process: fork()
    -- why not createProcess()?
  -- fork/exec separation
  -- parent and child
    -- orphan process and zombie process

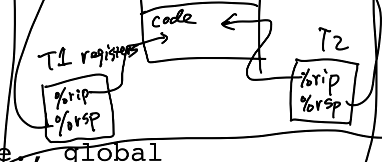
- programer's view




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

- OS's view
  -- PCB
  -- file descriptor table
  -- context (registers and other metadata)  → context switch

- system calls
  -- an interface between processes and kernels   (processes → running programs)
  -- how to know a systems call?
    -- $ man 2 <syscall>
  -- important system calls:
    -- fork, execve, wait, exit
    -- open, close, read, write
    -- pipe, dup2

- 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
    -- internal and external cmds  → programs
      -- how to tell? use "which"
    -- run commands (fork/exec)
    -- handle shell operators
      -- redirections
      -- exit status ($?)

- assembly code (x86-64)
  -- movq A,B
  -- pushq %rax
  -- popq %rax
  -- call foo
  -- ret

- calling convention
  -- where is the 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?

- context switches
  -- switching between to processes
  -- trap to kernel; save context; schedule another process; restore context; resume
  -- processes do not realize being interrupted

- 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)  exit
    -- non-preemptive scheduler: (1) running->terminated or (2) running->waiting
    -- preemptive scheduler: all four transitions

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

- RR: round-robin
```



3. concurrency

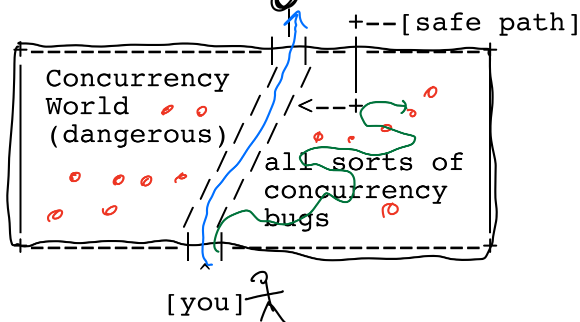


- threads
  - -- inaccurate: "abstracting a CPU core"
  - -- two threads in the same process share memory
    - -- meaning: they share code segment, data segment (i.e., global variables), and heap (i.e., memory from malloc)
  - -- threads have separate stack and registers
  - -- [will be crystal clear after studying virtual memory]

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

- mental model: dangerous concurrency world

[correct impl]



[you]

- dangerous concurrency world: concurrency problems
  - (1) if not sequential consistency (hardware)
    - -- hard for human beings to understand
    - -- cases in handout week6.a panel 1
  - (2) concurrency problems under sequential consistency
    - -- no synchronization:
      - -- multiple threads working on some shared state
      - -- race condition
        [examples: broken linked list and (broken) producer/consumer in handouts]
  - (3) compiler optimization
    - -- the example of printing 0s and 1s (scribble Week6.a)
  - (4) with incorrect synchronization
    - -- potential concurrency bugs of violating rules
      [e.g., using "if" instead of "while" to check waiting condition]

(1) safe path: how to write concurrency code
- -- safety first; this is one way, but is tested by time

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

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

- (2) Condition variable:
  - -- why? providing synchronization scheduling (motivating by the soda example)
  - -- APIs
    - -- cond_init(cond)
    - -- cond_wait(cond, mutex) // UGLY
    - -- cond_signal(cond)
    - -- cond_broadcast(cond)
  - -- an important interface, cond_wait(...)
    - -- meaning: 2 steps,
      (1) unlock mutex and wait (until cond signaled)

(2) acquire mutex and resume executing
-- why includes mutex (ugly interface)?

- four-step approach

4. we will cover lab2 and lab3

- do give a brief pass to your code
- expectation:
  -- if you've done your job, you can easily answer the questions
  -- if you're not, you still can but will spend a lot of time

notes about exam

- NOTE: bring your NUID

- a lot to read, not much to write

- a lot of code
  -- don't panic
  -- most lines are not important. They are there for completeness reasons.
  -- only several lines matter.

- 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 on the bottom of each page

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

- you don't have to remember the "jargons"
  -- but you do need to remember "common terms"
  -- how to distinguish the two?
    -- Imagine you're a system programmer. When you talk to your peers, are you comfortable to say the words without explanation?
    -- "connecting input and output of two commands? Use pipe."
       (most people are okay with this.)
    -- "try to predicting the future? Use EWMA."
       (most people are not okay with this; EWMA means exponentially weighted moving average btw.)
    -- "what's the register to hold the 5th arguments of a function?"
       (we will not ask such questions, as common programmers won't remember. The answer is %r8 btw.)

- we're reasonable people
  -- we're able to understand and reason things
  -- we're not idiots
  -- we will strictly follow the rules
     ("I don't know the rules [so they do not exist]" does not apply)

[broadcast the first page of the midterm
 leak some information, but that's fine.]

Q&A

Logistic questions

Content questions
  [shrug with no answer]

Northeastern University
CS3650: Computer Systems: Spring 2024

Midterm Exam

- This exam is **90 minutes**.
- 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 93 minutes after the exam begins and will not accept exams outside the room.
- There are **9** 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.**
- 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/24)	II (xx/24)	III (xx/28)	IV (xx/24)	Total (xx/100)

Name:

NUID: