Northeastern University
CS5600: Computer Systems: Fall 2021

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 ~~78~~ 93 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.**
- 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/20)	II (xx/27)	III (xx/20)	IV (xx/10)	V (xx/23)	Total (xx/100)

I C basics, Shell, and System Calls (20 points)

1. [6 points] Read the following C program.

```
int b = 4;

void foo(int a) {
  a = 5;
}

void bar(int *a) {
  a = &b;
 }

void baz(int *a) {
  *a = 5;
}

int main() {
  int a1 = 1, a2 = 1, a3= 1;
  foo(a1);
  bar(&a2);
  baz(&a3);
  printf("%d,%d,%d\n", a1, a2, a3);
  return 0;
}
```

What is the output of this program? (namely, what you will see on the screen)

1,1,5
(why? these are valid code; try them)

2. [6 points] Answer the following questions.

(a) Consider the shell command: `$ man 2 brk`

What does this command do? **Explain in 1 sentence.**

This command will display the manual pages of *system call* `brk`.

(b) Name one Unix system call that waits for process termination.

`waitpid`
or any other *wait* system calls.

(c) A process is running while its parent process has terminated.

Name: Solutions **NUID:**

What is this process called? **Circle your answer.**

A. Zombie process

B. Lonely process

C. Orphan process

D. Daemon process

C

3. [8 points] Read the following shell commands and answer questions.

Here are some notes:

- Use \`empty` to represent an empty string.
- If you think there are multiple correct answers, write one of them.
- As a reminder,
  `echo`: write arguments to the standard output
  `cat`: print file contents to the standard output
  `rev`: reverse the order of characters in a line

(a)
```
$ echo abc > file1 2> file2
```

Both file1 and file2 are empty files in the beginning.
Write down the contents of file1 and file2 after running the command.

file1: abc

file2: \empty

(b)
```
$ echo abc | rev
```

What is the command output? (namely, what you will see on the screen).

cba

(c)
```
$ cat doesnotexist.file 2> /dev/null && echo "hello"
```

File `doesnotexist.file` does not exit.
What is the command output? (namely, what you will see on the screen).

\empty

(d)
```
$ (echo a &) && (echo b &) || (echo abc) | (rev)
```

What is the command output? (namely, what you will see on the screen).

a
b

OR

b
a

II Processes and Threads (27 points)

4. [6 points] Two threads in a process share:
(**Circle all that apply**. If you leave it blank, you will get 1 point; otherwise, any missing or wrong choices will be -2 points until 0.)

A. Stack

B. Registers

C. Global variables

D. Program code

E. Heap

C, D, E

5. [6 points] Read the code and answer the question below.
(assume we compile the code with `gcc -O0`, meaning no optimization.)

```
void foo(int a) {
    int b = a + 1;
    int *c = (int*)malloc(sizeof(int));
    *c = b + 1;
}
```

If a `main` function executes `foo(1)`, then `a == 1`, `b == 2`, and `*c == 3`.
Where are these 1 (`a`'s value), 2 (`b`'s value), and 3 (`*c`'s value) located?
Write down where are these *three* locations by choosing from the following options.
(if you think there are multiple correct options, choose any one of them)

A. stack

B. heap

C. data segment

D. code segment

E. registers

1 (a's value): A OR E
2 (b's value): A
3 (*c's value): B

[You need to *write down* (or explicitly mark) three locations and their corresponding options to get all the scores.]

6. [15 points] Below are a piece of C code and the corresponding assembly code (produced by `gcc -O0 -S`). Read them and answer questions below.

As a reminder, here are some assembly instructions and their explanations:

```
movq A, B     // move 64-bit data from A to B
pushq %rax    [ subq $8, %rsp        // subtract 8 from the register
                movq %rax, (%rsp) ]
popq %rax     [ movq (%rsp), %rax
                addq $8, %rsp      ] // add 8 to the register
call 0x123    [ pushq %rip
                movq $0x123, %rip ]
ret           [ popq %rip ]
```

C code:

```
C1   int foo(int a) {
C2       int b = a + 1;
C3       return b;
C4   }
C5
C6   int main() {
C7       int a = 1;
C8       int b = foo(a);
C9   }
```

Assembly code:

```
S1   foo:
S2       pushq   %rbp
S3       movq    %rsp, %rbp
S4       movl    %edi, -20(%rbp)
S5       movl    -20(%rbp), %eax
S6       addl    $1, %eax
S7       movl    %eax, -4(%rbp)
S8       movl    -4(%rbp), %eax
S9       popq    %rbp
S10      ret
S11
S12  main:
S13      pushq   %rbp
```

```
S14     movq    %rsp, %rbp
S15     subq    $16, %rsp
S16     movl    $1, -8(%rbp)
S17     movl    -8(%rbp), %eax
S18     movl    %eax, %edi
S19     call    foo
S20     movl    %eax, -4(%rbp)
S21     movl    $0, %eax
S22     ret
```

(2 points) Which assembly instruction will be executed after running line `S10` (i.e., `ret`)?
Write down the line number. (for example, `S1`)

S20

(4 points) After running line `S2` (the first instruction of function `foo`), the top element on stack is the content of register `%rbp`.
What is the second top element on stack? (namely, the element returned by the second stack pop)

`%rip`, OR content of `%rip`, OR `S20`

(3 points) What is the assembly code that represents line `C7`?
Write down the line number or numbers. (for example, `S1` or `S1`–`S22`)

S16

(3 points) During a function call, the caller needs to save call-clobbered registers.
Where is this saving in `main` function when calling `foo`?
Write down the line number or numbers. (for example, `S1` or `S1`–`S22`; if there is none, write none.)

none

[Because of my mistake, you will have 3 points for FREE!]

III Concurrency (20 points)

7. [6 points] Write down the six rules (commandments) in Mike Dahlin's "Coding Standard for Programming with Threads".

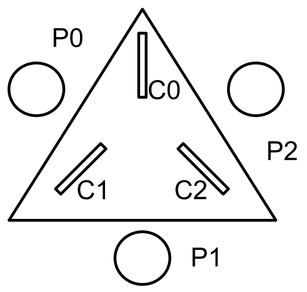
Answer: (The order doesn't matter.)

1. Always do things the same way

Name: Solutions **NUID:**

2. Always use monitors (condition variables + locks)
3. Always hold lock when operating on a condition variable
4. Always grab lock at beginning of procedure and release it right before return
5. Always use `while` for `cond_wait` not `if`
6. never sleep.

8. [14 points] In this question, you will solve a variant of the classic dining philosophers problem. You and two of your friends seat around a triangle table. Between every two people, there is a single chopstick. A person has to pick up two chopsticks (on their left and right) to eat. If people cannot get both chopsticks, they should not pick up any chopsticks. Here is a visualization:



Here is how we model the problem: the table is global state. Each person has an integer pid from 0 to 2. We model each person as a thread, with the pseudocode:

```
Table_t table;  // global variable

while (1) {
  talk();
  getChopsticks(&table, pid);
  eat();
  putChopsticks(&table, pid);
}
```

You need to implement the Table as a monitor, and finish the functions in the next page. Follow the six concurrency rules/commandments that you answered in the last question.

Note:

- The next page provides some structure, including two helper functions, `left()` and `right()`.
- Finish the three `TODO`s in the next page.
- Threads should *sleep* if they cannot make progress.
- Your solution must be deadlock-free.
- Ignore starvation.

- You should write valid C code, but we tolerate minor syntax errors.
- If you cannot remember the exact names of library functions or arguments, you can use vague names following a comment "`// vague`". (if necessary, explain your code in comments.)
- You can assume that mutexes and condition variables will be automatically initialized.
- You do not have to include header files (assume you have all of them already).

```
// Helper functions. Given a person's pid, these return the
// number of the left chopstick and the right chopstick.
// (Chopsticks are numbered from 0 to 2.)
int left (int pid) { return pid; }
int right(int pid) { return ( pid + 1 ) % 3; }

// a table with three chopsticks
typedef struct table {
    int chopstick_status[3];  // 0: available, 1: not available
} Table_t;

mutex_t m; // vague
cond_t cv; // vague

void getChopsticks(Table_t *table, int pid) {
    acauire(&m); // vague
    int left_chop = left(pid);
    int right_chop = right(pid);

    // if I cannot get both chopsticks, I should wait
    while(table->chopstick_status[left_chop] != 0 ||
          table->chopstick_status[right_chop] != 0)
    {
        cond_wait(&cv, &m); // vague
    }

    table->chopstick_status[left_chop] = 1;
    table->chopstick_status[right_chop] = 1;

    release(&m); // vague
}

void putChopsticks(Table_t *table, int pid) {
    acauire(&m); // vague

    table->chopstick_status[left_chop] = 0;
    table->chopstick_status[right_chop] = 0;

    broadcast(&cv); // vague
      // [Note: signal also works for this problem.
      //  If there are more than three people, will signal still work?]

    release(&m); // vague
}
```

IV Scheduling (10 points)

9. [10 points] Read the scheduling problem below and answer questions.
(This problem follows the scheduling game assumptions in class.)

```
process      arrival     running
P1              0           5
P2              2           4
P3              3           1
P4              4           1
```

(4 points) What are the scheduling decisions when using STCF?
Note: STCF is Shortest Time-to-Completion First, a classic preemptive scheduling algorithm. Of course, in general, STCF cannot be implemented because an algorithm cannot infer how long each process will run, but here we are given that information.

Write down the process scheduled for each time slot. (as we did in class)

Answer:

```
0    1    2    3    4    5    6    7    8    9    10
P1   P1   P1   P3   P4   P1   P1   P2   P2   P2   P2
```

[Points are assigned to those "important" decision making moments.]

(2 points) What is the average turnaround time?
Write down a float number.

(7 + 9 + 1 + 1) / 4 = 4.5

(2 points) What is the average response time?
Write down a float number.

(0 + 5 + 0 + 0) / 4 = 1.25

[As mentioned in the question, we use "game assumptions in class". The calculation follows what we did *in class*.]

(2 points) For this problem, which algorithm may produce a better average response time?
Circle one answer. (If you think there are multiple correct answers, choose one.)

A. FIFO (First-In-First-Out)

B. RR (Round-Robin)

C. FCFS (First-Come-First-Serve)

D. SJF (Shortest-Job-First)

B

V Lab1 and Lab2 (23 points)

10. [10 points] Below is an implementation of Lab1 Caesar encoding for numbers. Read the code and answer questions. (Note: in C, the result of `(-1 % 10)` is `-1`.)

```
char *encode_numbers(char *plaintext, int shift) {
    int size = strlen(plaintext);
    int i = 0;
    for (; i < size; i++) {
        char c = plaintext[i];
        if (c >= '0' && c <= '9') {
            int pos = (c - '0') + shift;
            if (pos < 0) {
                pos = (10 + (pos % 10)) % 10;
            } else {
                pos = pos % 10;
            }
            plaintext[i] = '0' + pos;
        } else {
            return "ERROR";
        }
    }

    return plaintext;
}
```

(8 points) What are the outputs of function `encode_numbers` with the following four inputs? **Write down the values of** `ret1`–`ret4`. (each answer should be a string.)

```
    char input1[2] = "1";
    char* ret1 = encode_numbers(input1, 1);

    char input2[2] = "1";
    char* ret2 = encode_numbers(input2, -1);


    char input3[3] = "-1";
    char* ret3 = encode_numbers(input3, 1);


    char input4[3] = "-1";
    char* ret4 = encode_numbers(input4, -1);
```

ret1: ”2”
ret2: ”0”
ret3: ”ERROR”
ret4: ”ERROR”

(2 points) Read the code snippet below which invokes `encode_numbers`.

```
   char input[2] = "1";
   char *ret = encode_numbers(input, 1);
   printf("%s", input);
```

Name: Solutions NUID:

Write down the output. (namely, what will appear on the screen)

2

11. [13 points] Below is an implementation of Lab2 `cd` built-in command. We remove error handling code for simplicity. Read the code and answer questions.

As a reminder, here is the definition of `command_t` in `cmdparse.h`.

```
typedef struct command command_t;

struct command {
    char *argv[MAXTOKENS+1];    // command argument array
    char *redirect_filename[3]; // filenames to be used for redirections
    command_t *subshell;        // pointer to subshell command line
    controlop_t controlop;      // control operator applied between
                                //   this command and the next
    command_t *next;            // pointer to the next command
                                //   in the command line
};
```

Code snippet of Lab2 implementation:

```
command_t cmd;

... // update 'cmd' to be a valid 'cd' command

int pid;
if ((pid = fork()) == 0) {
    if (strcmp(cmd->argv[0], "cd") == 0) {  // Line A
        int ret = chdir(cmd->argv[1]);
        if (ret != 0) {
            exit(2);
        }
        exit(0);
    }
} else {
    int status, exitval;                    // Line C
    waitpid(pid, &status, WUNTRACED);
    exitval = WEXITSTATUS(status);

    if (strcmp(cmd->argv[0], "cd") == 0) {  // Line B
        if (exitval == 0) {
            chdir(cmd->argv[1]);
        }
    }
}
```

Answer the questions in the next page.

(3 points) Is it true that Line A is always executed before Line B?
(Line A and B are labels in the comments.)
Answer Yes or No.

Yes.

[because `waitpid` makes the parent wait.]

(4 points) In the child process, the code has already called the system call `chdir` once. Why `chdir` is invoked a second time in the parent process?
Explain why in 1 sentence.
(hint: consider what `cd` does from a shell-user's perspective.)

Only invoking `chdir` in the parent (namely, the shell process) will change the current directory that shell-users can see.

(6 points) We want to support `stdout` redirection operator (namely, >) for command `cd`; for example, `cd /foo/bar > /dev/null`.
Circle the options below to implement this.

First, we need to close the standard output file:

```
A. close(0);
B. close(1);
C. close(2);
```

Second, we need to open the redirected file:

```
A. open(cmd->redirect_filename[0], O_CREAT|O_RDWR|O_TRUNC, 0666);
B. open(cmd->redirect_filename[1], O_CREAT|O_RDWR|O_TRUNC, 0666);
C. open(cmd->redirect_filename[2], O_CREAT|O_RDWR|O_TRUNC, 0666);
```

Third, we need to put the above two lines code:

```
A. immediately before Line A
B. immediately before Line B
C. immediately before Line C
```

B B A

End of Midterm