Northeastern University
CS5600: Computer Systems: Spring 2022

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 **12** 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/18)**	**III (xx/18)**	**IV (xx/24)**	**V (xx/20)**	**Total (xx/100)**

Name: Solutions

NUID:

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

1. **[10 points]** Read shell commands below and answer questions.

   - 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
     `shuf`: generate random permutations of inputs

(a) `$ man man`

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

Display the manual of command `man`.

(b) `$ cat students.txt | shuf -n 1`

What does this command do? **Explain in 1-2 sentence.**
(note: your answer does not have to explain "`-n 1`".)

Shuffle the contents of file `students.txt` and print one line on screen. Notice that this is our lottery script using in class.

(c) `$ (false || true) && echo True`

What is the output of this command?

True

(d) `$ echo ABC;echo DEF > file1`

What are the contents of file1 after executing this command?

DEF

(e) `$ cat doesnotexist.file 2> /dev/null && echo "cs5600"`

File `doesnotexist.file` does not exit.
What is the output of this command?

`\empty`

2. [4 points] Fill in a piece of C code below to create a zombie process.

- Your code should always (instead of probably) create a zombie process.
- If you cannot remember syscall arguments, write placeholders and explain what you mean in comments.

One possible answer:

```
int main() {
    if (fork() == 0) { // child
        exit(0);
    } else {       // parent
        while(1) {}
    }
}
```

3. [6 points] Read the following C program. Note that we omit headers for simplicity.

```
int x = 5;

void foo(int p) {
    int x = 6;
    p = x;
}

void bar(int *p) {
    p = &x;
}

void baz(int *p) {
    *p = x;
}

int main() {
    foo(x);
    printf("foo: %d\n", x);
    int x = 0;
    bar(&x);
    printf("bar: %d\n", x);
    baz(&x);
    printf("baz: %d\n", x);
}
```

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

```
foo: 5
bar: 0
baz: 5
```

II Processes and Threads (18 points)

4. [6 points] Assume thread T1 and T2 are two threads in the same process. Are the following statements True or False?
Circle your answers.

[True / False] T1 and T2 share the same code segment.

[True / False] If T1 calls syscall `exit()`, it will exit. But, T2 will be still alive.

[True / False] If T1 has acquired a mutex m1, T2 can acquire another mutex m2.

True, False, True

5. [6 points] Read the code and choose all possible outputs.

```
int main() {
    int x = 1;
    if (fork() == 0) {
        x = x + 1;
        printf("%d,", x);
    } else {
        x = x * 2;
        wait(NULL);
        printf("%d\n", x);
    }
}
```

(hint: notice that these are two processes.)
Circle all that apply.
(If you leave it blank, you will get 2 points; otherwise, any missing or wrong choices will be -2 points until 0.)

A. 2,2

B. 2,4

C. 3,2

D. 2,3

E. 1,2

A

Notice the hint. Since these are two processes, the two `x` in parent and child point to different locations in physical memory. Changing one does not affect the other.

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

```
int x = 5599;
int main() {
    x = x + 1;
    int *y = (int*) malloc(sizeof(int));
    int z = x + 1;
    *y = z + 1;
}
```

After executing `main` (before quitting), `x == 5600`, `z == 5601`, and `*y == 5602`.

Where are these values located in memory?
Choose from: A. stack, B. heap, C. data segment, D. code segment.
Write down your answer below.

5600 (x's value):

5601 (z's value):

5602 (*y's value):

C, A, B

III Scheduling and Concurrency (18 points)

7. [18 points] Consider the setup below, which follows the scheduling game from class. Time is divided into epochs. Jobs arrive at the **very beginning** of the epoch listed under "arrival"; after they have been given the number of CPU epochs listed under "running", they leave. Assume processes do no I/O and no synchronization.

	arrival	running
P1	0	5
P2	1	2
P3	2	3

(4 points) The system runs the Shortest Time-to-Completion First (STCF), a classic preemptive scheduling algorithm.
Write down the process scheduled for each epoch.

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

(4 points) Later, the system administrator assigns priorities to processes as follows:

```
P1: low
P2: medium
P3: high
```

Then, the system runs the preemptive priority scheduling algorithm.
Write down the process scheduled for each epoch.

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

(4 points) Later, the programmers add synchronization to the processes as follows. Note that P1 and P3 share the same `lock`.

```
P1:
  acquire(&lock);
  run1();  // takes 5 epochs
  release(&lock);


P2:
  run2();  // takes 2 epochs


P3:
  run3a();  // takes 2 epochs
  acquire(&lock);
  run3b();  // takes 1 epochs
  release(&lock);
```

The system still uses the preemptive priority scheduling algorithm.
Write down the process scheduled for each epoch now.

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

Notice this is priority inversion.

(3 points) **What is the average turnaround time for the above schedule?**

(9 + 4 + 8) / 3 = 7

(3 points) **What is the average response time for the above schedule?**

(0 + 0 + 0) / 3 = 0

IV Concurrency (24 points)

8. [4 points] Write down any four of the six rules (commandments) in Mike Dahlin's "Coding Standard for Programming with Threads".

Answer:

Any four of the following:

1. Always do things the same way
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` instead "if"
6. (Almost) never `sleep()`

9. [10 points] Consider the producer-consumer problem that we intensively discussed in class.

- Assume there are multiple producers and multiple consumers.
- We have a mutex `m` and two condition variables `nonfull` (waiting on this c.v. until the buffer is not full) and `nonempty` (waiting on this c.v. until the buffer is not empty).
- `count` is the number of item in the buffer. `BUFFER_SIZE` is the capacity of the buffer.
- Assume that each of the patterns below is enclosed in the mutex's acquire/release critical section.

Which of the following patterns abides by the concurrency commandments and common notions of program correctness?

Circle all that apply.

A.
```
if (count == BUFFER_SIZE)
     cond_wait(&nonfull, &mutex);
```

B.
```
while (count == BUFFER_SIZE)
     cond_wait(&nonfull, &mutex);
```

C.
```
if (count > 0)
     signal(&nonempty);
```

D.
```
while (count > 0)
     signal(&nonempty);
```

E.
```
if (count > 0)
      broadcast(&nonempty);
```

B, C, E

A is incorrect because of using `if` for `cond_wait`.
B is standard usage of condition variables.
C is correct (notice that the `if` here is not about `cond_wait`!).
D is incorrect (in terms of "program correctness"): it loops without doing anything useful.
E is correct: for consumer-produce example, waking up multiple producers/consumers is fine.

10. [10 points] A bank decides to use fine-grained locking to protect transactions.

- The code below is an implementation for two clients, Alice and Bob.
- Alice and Bob have their own `balance` and mutex `mtx`.
- Function `transfer` moves money (of size `trans`) from one account (i.e., `from`) to another account (i.e., `to`).
- Assume all the variables are initialized correctly.

```
int balance[2];    // 0 for Alice, 1 for Bob
mutex_t mtx[2];    // 0 for Alice, 1 for Bob

bool transfer(int from, int to, int trans) {
  acquire(&mtx[from]);
  acquire(&mtx[to]);

  bool result = false;
  if (balance[from] > trans) {
    balance[from] = balance[from] - trans;
    balance[to] = balance[to] + trans;
    result = true;
  }

  release(&mtx[to]);
  release(&mtx[from]);
  return result;
}
```

(5 points) The code above is buggy—it has a deadlock. What interleaving will trigger the deadlock? **Describe a deadlock interleaving in 2-3 sentences.**

If two threads, one for Alice (0) and one for Bob (1), together run this function `transfer`, Alice's thread may have mutex `mtx[0]` and Bob's thread may have mutex `mtx[1]`. Now, there is a deadlock—the two threads hold the locks that the other thread requires.

(5 points) Rewrite function `transfer` to eliminate the deadlock.
Write your code below.

```
bool transfer(int from, int to, int trans) {
    acquire(&mtx[0]);
    acquire(&mtx[1]);

    bool result = false;
    if (balance[from] > trans) {
      balance[from] = balance[from] - trans;
      balance[to] = balance[to] + trans;
      result = true;
    }

    release(&mtx[1]);
    release(&mtx[0]);
    return result;
}
```

```
This is one solution, which only works for this specific case of Alice and Bob.
If your solution acquires the lock in order, it is fine.
```

V Lab1 and Lab2 (20 points)

11. [10 points] Below is an implementation of Lab1 Caesar encoding.
Two functions, `shift_digit` and `shift_letter`, follow the Caesar Cipher described in Lab1 and encode a digit (like 9) and a letter (like Z), respectively. Their implementation has been omitted.

Read the code and answer questions below.

```
char shift_digit(char digit, int key);
char shift_letter(char letter, int key);

char *encode_num_str(char *plaintext, int key) {
    int size = strlen(plaintext);
    int i = 0;
    for (; i < size; i++) {
        char c = plaintext[i];
        if (/* CONDITION1 */) {
            plaintext[i] = shift_digit(c, key);
        } else if (/* CONDITION2 */) {
            plaintext[i] = shift_letter(c, key);
        } else {
            return "ILLCHAR";
        }
    }

    return plaintext;
}
```

(3 points) **What should be filled in the `CONDITION1`?**

```
c >= '0' && c <= '9'
```

(3 points) **What should be filled in the `CONDITION2`?**

```
 (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')
```

(4 points) What are the outputs of the code snippets below?

```
    // case 1
    char num[2] = "1";
    char *ret1 = encode_num_str(num, 1);
    printf("%s", ret1);

    // case2
    char str[4] = "abc";
    char *ret2 = encode_num_str(str, 2);
    printf("%s", ret2);
```

Write down the outputs below.

case 1: 2

case 2: cde

12. [10 points] Below is one implementation of Lab2's `cmd_exec` function. There are five slots for you to fill (A–E).
(Note: for a pipe syscall, `pipe(pipe_fds)`, data written to `pipe_fd[1]` appears on (i.e., can be read from) `pipe_fd[0]`.)

```
pid_t cmd_exec(command_t *cmd, int *pass_pipefd)
{
    pid_t pid = -1;
    int pipe_fds[2] = {-1, -1};
    if (cmd->controlop == CMD_PIPE) {
        pipe(pipe_fds);
    }

    if ((pid = fork()) == 0) {      // child process (executing cmd)
      if (pipe_fds[0] != -1) {      // if we created pipe this time
        dup2(pipe_fds[/* A */], /* B */);
        close(pipe_fds[0]);
        close(pipe_fds[1]);
      }
      if (*pass_pipefd > 0) {      // pipe from last time
        dup2(*pass_pipefd, /* C */);
        close(*pass_pipefd);
      }
      ...
    } else {                       // parent process (Shell)
      if (pipe_fds[0] != -1) {     // if we created pipe this time
        *pass_pipefd = pipe_fds[/* D */];
        close(pipe_fds[/* E */]);
      } else {                     // if no pipe, restore
        *pass_pipefd = STDIN_FILENO;
      }
        ...
    }
    ...
}
```

Write down what you should fill in A–E:

A: 1
B: 1
C: 0

D: 0
E: 1

If you know that pipe is about stdin and stdout, you will realize that answers are only about `0` and `1` (of course, other equivalent answers also work). By both the pipe's semantic (left cmd outputs to right cmd inputs) and also your Lab2 knowledge (cs5600sh handling cmd from left to right), you should be able to see the answers.

End of Midterm