CSCI 4210 — Operating Systems
Exam 1 Prep and Sample Problems (document version 1.1)

- Exam 1 is on Monday, February 24
- Exam 1 will be a 110-minute exam (6:00-7:50PM) in DCC 308
- If you have extra-time accommodations, then you will have either 165 minutes (50%) or 220 minutes (100%); your exam will be in Amos Eaton 217; for 50% extra-time, your start time is 5:00PM; for 100% extra-time, your start time is 4:00PM
- No calculators, cellphones, etc.; please be sure to turn off electronic devices before the exam
- You will be allowed to use one double-sided or two single-sided 8.5"x11" cribsheets
- Make-up exams are given only with an official excused absence (`http://bit.ly/rpiabsence`); also re-read the syllabus
- Exam 1 covers everything through Tuesday, February 18, including Lecture Exercises 1 and 2 and Homeworks 1 and 2; this includes general C programming, the C preprocessor, static and dynamic memory allocation, pointer arithmetic, I/O buffering, character strings, file descriptors, process creation, process management, pipes, FIFOs (named pipes), and CPU scheduling
- To prepare for the exam, focus on the posted code examples, practice problems in the lecture exercises, and suggested "to do" items, as well as other code modifications you can think of; read `man` pages and review the behavior of all system calls and library functions we have covered; also see sample questions below
- **All work on the exam must be your own; do not copy or communicate with anyone else about the exam both during and after the exam until it is graded**

Sample problems

Practice problems for Exam 1 are provided on the pages that follow. Feel free to post your solutions in the Discussion Forum; and reply to posts if you agree or disagree with the proposed approaches/solutions.

- Assume all system calls and library functions complete successfully (unless otherwise noted).
- Assume that the parent process ID is `777`, with child processes numbered sequentially `800`, `801`, `802`, etc.
- Assume that we are running a 64-bit architecture.
- Assume all header files are included such that any given code compiles successfully without warnings or errors.

(v1.1) Solutions will be posted on Friday, February 21 and reviewed on Monday, February 24.

1. Review the C program below.

```
int main()
{
  int rc;
  printf( "ONE-%u\n", getpid() );
  rc = fork();
#ifdef PARTC
  rc = fork();
#endif
  printf( "TWO-%u\n", getpid() );
  if ( rc == 0 ) { printf( "THREE-%u\n", getpid() ); }
  if ( rc > 0 ) { printf( "FOUR-%u\n", getpid() ); }
  return EXIT_SUCCESS;
}
```

   (a) What is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities.

   (b) What is the **exact** terminal output and the **exact** contents of the `output.txt` file if `stdout` is redirected to a file as shown below?

```
bash$ ./a.out > output.txt
```

   (c) What is the **exact** terminal output if `PARTC` is defined when this code is compiled?

2. Review the C program below.

```
int main()
{
  int x = 2222;
  printf( "PARENT: x is %d\n", x );

  pid_t p = fork();
  printf( "PARENT %d: forked...\n", p );

  if ( p == 0 )
  {
    printf( "CHILD: happy birthday to %d\n", getpid() );
    x += 18;
    printf( "CHILD: %d\n", x );
  }
  else
  {
#ifndef PARTC
    waitpid( p, NULL, 0 );
#endif

    printf( "PARENT %d: child completed\n", getpid() );
    x -= 2000;
    printf( "PARENT: %d\n", x );
  }

  return EXIT_SUCCESS;
}
```

(a) What is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities.

(b) How does the terminal output change if `x` is initialized to -2222?

(c) How would the terminal output change if `PARTC` is defined when the code is compiled?

(d) What is the **exact** terminal output and the **exact** contents of the `output.txt` file if `stdout` is redirected to a file as shown below?

```
bash$ ./a.out > output.txt
```

3. Review the C program below.

```
int main()
{
  char * a = "POLYTECHNIC";
  char * b = a;
  char * c = calloc( 100, sizeof( char ) );

  printf( "(%s)(%s)(%s)\n", a + 10, b + 9, c + 8 );

  char ** d = calloc( 100, sizeof( char * ) );
  d[7] = calloc( 20, sizeof( char ) );
  d[6] = c;
  strcpy( d[7], b + 5 );
  strcpy( d[6], b + 4 );

  printf( "(%s)(%s)(%s)\n", d[7], d[6], c + 5 );

  float e = 2.71828;
  float * f = calloc( 1, sizeof( float ) );
  float * g = f;
  float * h = &e;

  printf( "(%3.2f)(%2.2f)(%2.3f)\n", *f, *g, *h );

  return EXIT_SUCCESS;
}
```

(a) What is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities.

(b) Add code to ensure there are no memory leaks. Specifically, call `free()` at the earliest possible points in the code.

(c) Rewrite the code above to eliminate all square brackets.

(d) How might the terminal output change if `malloc()` is used here instead of `calloc()`?

4. Review the C program below.

```
#define SPARKS 1

int main()
{
  close( 2 );
  printf( "HELLO\n" );

#ifdef QUIET
  close( SPARKS );
#endif

  int fd = open( "output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644 );
  printf( "==> %d\n", fd );
  printf( "WHAT?\n" );
  fprintf( stderr, "ERROR\n" );

  close( fd );

  return EXIT_SUCCESS;
}
```

(a) What is the **exact** terminal output and the **exact** contents of the `output.txt` file? If multiple outputs are possible, succinctly describe all possibilities.

(b) How do the terminal output and file contents change if `QUIET` is defined when this code is compiled?

(c) Add code to redirect all output on `stdout` and `stderr` to the output file. Can you accomplish this without removing or changing any of the existing code?

5. Review the C program below.

```
#define SPARKS 0

int main()
{
  close( 2 );
  printf( "HELLO\n" );

#ifdef QUIET
  close( SPARKS + 1 );
#endif

  int fd = open( "output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644 );
  printf( "==> %d\n", fd );
  printf( "WHAT?\n" );
  fprintf( stderr, "ERROR\n" );

  int p = fork();

  if ( p == 0 )
  {
    printf( "ABCD %d?\n", getpid() );
    fprintf( stderr, "ERROR %d ERROR\n", getpid() );
  }
  else
  {
    printf( "%d\n", waitpid( p, NULL, SPARKS ) );
  }

  printf( "BYE %d\n", getpid() );
  fprintf( stderr, "HELLO\n" );
  close( fd );
  return EXIT_SUCCESS;
}
```

(a) What is the **exact** terminal output and the **exact** contents of the `output.txt` file? If multiple outputs are possible, succinctly describe all possibilities.

(b) How do the terminal output and file contents change if `QUIET` is defined when this code is compiled?

(c) How do the terminal output and file contents change if `SPARKS` is defined to be `WNOHANG` instead of 0?

(d) If we also redirect `stdout` to a file (e.g., `./a.out > xyz.txt`) on the terminal, what is the **exact** contents of the `xyz.txt` file?

6. Review the C program below.

```
int main()
{
  int * pipefd = calloc( 2, sizeof( int ) );
  int rc = pipe( pipefd );
  pid_t p = fork();

  if ( p == 0 )
  {
    rc = write( *(pipefd+1), "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 26 );
    printf( "PID %u: Wrote %d bytes\n", getpid(), rc );
  }
  else
  {
    char * buffer = calloc( 26, sizeof( char ) );
    rc = read( *(pipefd+0), buffer, 16 );
    printf( "PID %u: Read [%s]\n", getpid(), buffer );

#ifdef PARTB
    rc = read( *(pipefd+0), buffer, 16 );
    printf( "PID %u: Read [%s]\n", getpid(), buffer );
#endif

#ifdef PARTC
    rc = read( *(pipefd+0), buffer, 16 );
    printf( "PID %u: Read [%s]\n", getpid(), buffer );
    rc = read( *(pipefd+0), buffer, 16 );
    printf( "PID %u: Read [%s]\n", getpid(), buffer );
#endif
  }

  return EXIT_SUCCESS;
}
```

(a) What is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities.

(b) How does the terminal output change if `PARTB` is defined when this code is compiled?

(c) How does the terminal output change if `PARTC` is defined when this code is compiled?

(d) Fix the bugs in the given code by adding code where necessary. You are not allowed to change or remove any of the given code.

7. Describe what happens when a child process terminates; also, what happens when a process that has child processes terminates?

8. How does a shell such as `bash` actually execute commands? And given this, how could pipe functionality be implemented? For example, how do the `ps` and `wc` commands get executed such that the output from `ps` is fed in as input to `wc`?

   ```
   bash$ ps -ef | wc -l
   ```

   How about for the piped commands below?

   ```
   bash$ ps -ef | grep goldsd | wc -l
   ```

   Can you expand the `shell.c` example code to support pipes? As a hint, see the `fork-ps-grep.c` example.

9. When working with one or more pipes, why is it important to close unused pipe descriptors as early as possible? How does behavior change when using named pipes (i.e., FIFOs)?

10. Why might `fork()` fail? Why does a "fork-bomb" cause a system to crash? How can a "fork-bomb" be avoided?

11. Why is it important to use `free()` to deallocate dynamically allocated memory?

12. Describe (using code snippets) at least three ways to cause a segmentation fault.

13. Describe (using code snippets) at least three ways to cause a memory leak.

14. Describe (using code snippets) at least three ways to cause a buffer overflow.

15. Why is output to `stdout` and other file descriptors generally buffered by default? Why is output to `stderr` never buffered?

16. Write C code to create two pipes, then call `fork()` to create a child process. On the first pipe, the parent sends all keyboard input entered by the user. Note that the user can use `CTRL-D` to indicate `EOF`.

    The child process is responsible for reading the data from the first pipe and returning on the second pipe all alpha and newline characters (i.e., only the `'\n'` character and characters identified via `isalpha()`), ignoring all other characters.

    The parent outputs these characters to an output file called `alpha.txt`.

    Next, create a second version of this code that uses named pipes and therefore two completely separate processes, i.e., do not use `fork()`.

17. Write a program that creates n child processes, where n is given as a command-line argument. Number each child process $x = 1, 2, 3, \ldots, n$ and have each child process display a line of output showing its `pid` every x seconds, i.e., the first child process displays its `pid` every second, the second child process every two seconds, etc.

    When the parent process receives a `SIGINT` signal (`CTRL-C`), it must in turn send a `SIGTERM` signal via `kill()` to each child process to terminate each child process. Each process should display exactly 16 lines of output, then terminate, unless it is terminated by a signal.

    Be sure the parent process runs until all child processes have terminated.

18. Memory leaks and buffer overflows may exist in some of the code on this exam.

    (a) How many bytes are dynamically allocated and not deallocated via `free()` throughout this exam? Only include memory directly allocated by the given code, i.e., ignore memory that is dynamically allocated by `printf()`, etc. Also ignore any code that you add. Be sure to count all memory leaks in all running processes.

    Write the total number of bytes as a number (e.g., `1234`).

    (b) Questions on this sample exam may have code that works but buffer overflows occur. Do any questions have buffer overflows? If so, which questions have buffer overflows and what goes wrong?

19. For the processes described below, apply the FCFS, SJF, and SRT algorithms. Then for each process, calculate the wait time, turnaround time, and the number of preemptions (i.e., the number of times that the given process is preempted). If two or more events occur simultaneously, use ascending process ID order as the tie-breaking order (e.g., `P1` is placed in the queue before `P2`).

Process ID	CPU burst time	Arrival time
P1	1 ms	3 ms
P2	12 ms	1 ms
P3	5 ms	0
P4	4 ms	1 ms

20. Repeat the above problem for the RR algorithm; use a time slice of `3 ms`. Repeat again with a time slice of `4 ms`.

21. Repeat the above problem but when a preemption occurs, add the preempted process to the beginning of the queue instead of the end of the queue. In other words, switch it with the next process on the ready queue.