CSCI 4210 — Operating Systems
Exam 1 Prep and Sample Questions (document version 1.0)

- Exam 1 is scheduled for the extended window that starts at 4:10PM EDT on Tuesday 6/29 and ends at 2:10AM EDT on Wednesday 6/30
- Exam 1 will be a 120-minute exam, but you will have a full **three-hour window** to complete and submit your exam solutions
- Submitty will start the three-hour "clock" for you when you first "touch" the exam, so please plan accordingly by avoiding any distractions or interruptions; and note that **you must start your exam by 11:10PM EDT on Tuesday 6/29 to have the full three-hour window**
- There will be a mix of auto-graded and free response questions on the exam; for auto-graded question(s), you will submit your code in the same manner as the homeworks; for free response questions, please place all of your answers in **a single PDF file called** `exam1.pdf`
- Exam 1 is open book(s), open notes; given that you are working remotely, you may use any and all of the posted course materials, including all previous questions and answers posted in the Discussion Forum
- 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 live lecture Thursday 6/24, including all assignments due through that date; such coverage essentially focuses on C programming, static and dynamic memory allocation, pointer arithmetic, I/O buffering, character strings, file descriptors, process creation, process management, signals, pipes, and the shell
- To prepare for the exam, focus on the coding examples and suggested "to do" items, as well as other code modifications you can think of; review `man` pages and behavior for all system calls and library functions we have covered
- For free response questions, be as concise as you can in your answers; long answers are difficult to grade
- Pretend that you are taking this exam in West Hall Auditorium; therefore, all work on the exam **must** be your own; **do not even consider copying from other sources or communicating with others**
- **Any copying or collaborating with others will result in a grade of zero on the exam; this includes posting in the Discussion Forum or other public or private forums**

Sample problems

In addition to the above, 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.

Some selected solutions will be posted by Sunday 6/27.

1. Given the following C program, what is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities. Assume all system calls complete successfully. Also assume that the parent process ID is `777`, with child processes numbered sequentially `800`, `801`, `802`, etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

What is the exact terminal output and the exact contents of the `output.txt` file if `stdout` is redirected to a file as follows:

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

2. Given the following C program, what is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities. Assume all system calls complete successfully. Further, assume that the parent process ID is 777, with child processes numbered sequentially 800, 801, 802, etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
  int x = 2021;
  printf( "PARENT: x is %d\n", x );

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

  if ( pid == 0 )
  {
    printf( "CHILD: happy birthday to %d\n", getpid() );
    x += 19;
    printf( "CHILD: %d\n", x );
  }
  else
  {
    waitpid( -1, NULL, 0 );

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

  return EXIT_SUCCESS;
}
```

How would the output change if the `waitpid()` system call was removed from the code?

What is the exact terminal output and the exact contents of the `output.txt` file if `stdout` is redirected to a file as follows:

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

3. Given the following C program, what is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities. Assume all system calls complete successfully. Also assume that we are running this on a 64-bit architecture.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

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;
}
```

Add code to properly deallocate all dynamically allocated memory. In other words, make sure there are no memory leaks.

Rewrite the code above to eliminate all remaining square brackets.

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

4. Given the following C program, what is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities. Assume all system calls complete successfully.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  printf( "ONE\n" );
  fprintf( stderr, "ERROR: ONE\n" );
  int rc = close( 2 );
  printf( "==> %d\n", rc );

  printf( "TWO\n" );
  fprintf( stderr, "ERROR: TWO\n" );
  rc = dup2( 1, 2 );
  printf( "==> %d\n", rc );

  printf( "THREE\n" );
  fprintf( stderr, "ERROR: THREE\n" );

  return EXIT_SUCCESS;
}
```

What is the exact terminal output and the exact contents of the `output.txt` file if `stdout` is redirected to a file as follows:

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

How about if we redirect as follows:

```
bash$ ./a.out 2> output.txt
```

5. Given the following C program, what is the **exact** terminal output? Further, what is the **exact** contents of the `output.txt` file? If multiple outputs are possible, succinctly describe all possibilities. Assume all system calls complete successfully.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

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

#if 0
  close( 1 );  /* <== add this line later.... */
#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;
}
```

How do the output and file contents change if the second `close()` system call is uncommented?

Add code to redirect all output on `stdout` and `stderr` to the output file. Can you accomplish this without changing any of the given code?

6. Given the following C program, what is the **exact** terminal output? Further, what is the **exact** contents of the `output.txt` file? If multiple outputs are possible, succinctly describe all possibilities. Assume all system calls complete successfully. Further, assume that the parent process ID is `777`, with child processes numbered sequentially `800`, `801`, `802`, etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

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

#if 0
  close( 1 );  /* <== add this line later.... */
#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 rc = fork();

  if ( rc == 0 )
  {
    printf( "SUNS %d?\n", getpid() );
    fprintf( stderr, "ERROR %d ERROR\n", getpid() );
  }
  else
  {
    printf( "%d\n", waitpid( -1, NULL, 0 ) );
  }

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

How do the output and file contents change if the second `close()` system call is uncommented?

How do the output and file contents change if the third argument of `waitpid()` is changed to `WNOHANG`?

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?

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
   ```

9. In the `bash` shell, write a piped command to count the number of `"*.c"` and `"*.h"` files in the current working directory. Remember that the pipe character in the shell is `|`.

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

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

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