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

- 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 hints and solutions are shown on the next few pages.

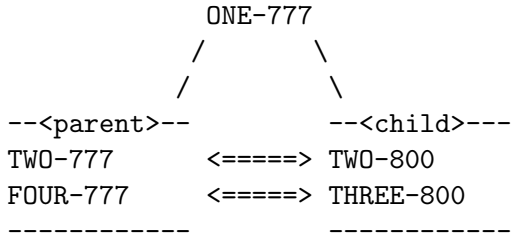
1. Given the following C program, what is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities. Assume all system calls complete successfully. Also assume that the parent process ID is `777`, with child processes numbered sequentially `800`, `801`, `802`,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

SOLUTIONS: When output to the terminal, there are six possible outputs, detailed below. In the diagram, the bidirectional arrows indicate blocks of output that could be interleaved.




```
ONE-777                 ONE-777                 ONE-777
TWO-777                 TWO-777                 TWO-800
FOUR-777                TWO-800                 TWO-777
TWO-800                 THREE-800               THREE-800
THREE-800               FOUR-777                FOUR-777

ONE-777                 ONE-777                 ONE-777
TWO-777                 TWO-800                 TWO-800
TWO-800                 TWO-777                 THREE-800
FOUR-777                FOUR-777                TWO-777
THREE-800               THREE-800               FOUR-777
```

When output is redirected to a file, `stdout` switches to being block-buffered; therefore, we have only two possible outputs, as illustrated below (i.e., either the entire `<parent>` block is output first or the entire `<child>` block is output first). No other interleaving occurs.

```
--<parent>--   -or-   --<child>---
ONE-777               ONE-777
TWO-777               TWO-800
FOUR-777              THREE-800
--<child>---          --<parent>--
ONE-777               ONE-777
TWO-800               TWO-777
THREE-800             FOUR-777
------------          ------------
```

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.

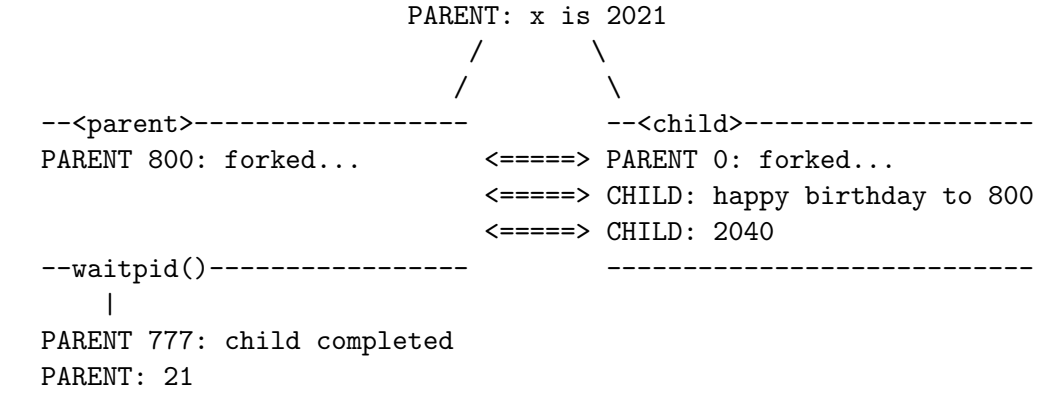
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

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
```

SOLUTIONS: There are four possible outputs, as shown below. In the diagram, the bidirectional arrows indicate lines of output that could be interleaved.




```
PARENT: x is 2021
PARENT 800: forked...
PARENT 0: forked...
CHILD: happy birthday to 800
CHILD: 2040
PARENT 777: child completed
PARENT: 21
```

```
PARENT: x is 2021
PARENT 0: forked...
CHILD: happy birthday to 800
PARENT 800: forked...
CHILD: 2040
PARENT 777: child completed
PARENT: 21
```

```
PARENT: x is 2021
PARENT 0: forked...
PARENT 800: forked...
CHILD: happy birthday to 800
CHILD: 2040
PARENT 777: child completed
PARENT: 21
```

```
PARENT: x is 2021
PARENT 0: forked...
CHILD: happy birthday to 800
CHILD: 2040
PARENT 800: forked...
PARENT 777: child completed
PARENT: 21
```

If the `waitpid()` system call is removed from the code, the last two lines of the parent process would move up in the diagram and become part of the block of output that could interleave with the child output. There would therefore be 20 possible outputs, so showing the modified diagram would be best here.

SOLUTIONS (CONTINUED): When output is redirected to a file, `stdout` switches to being block-buffered; therefore, given the `waitpid()` in the parent process, we have only one possible output, as shown below. Also note that the first line is duplicated (i.e., printed twice) since the buffer is copied to the child process.

```
PARENT: x is 2021
PARENT 0: forked...
CHILD: happy birthday to 800
CHILD: 2040
PARENT: x is 2021
PARENT 800: forked...
PARENT 777: child completed
PARENT: 21
```

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()`?

SOLUTIONS: There is only one possible output, as shown below (though think about how this would differ using a 32-bit architecture or if we added one or more `fork()` calls to this code).

```
(C)(IC)()
(ECHNIC)(TECHNIC)(IC)
(0.00)(0.00)(2.718)
```

To properly deallocate memory, we need to first add the following calls to `free()` after the second call to `printf()`:

```
free( d[7] );
free( d[6] );  /* or free( c ) */
free( d );
```

Next, we need to add the following call to `free()` after the last call to `printf()`:

```
free( f );  /* or free( g ) */
```

To use only pointer arithmetic, replace `d[7]` with `*(d+7)`, etc.

And finally, if we called `malloc()` instead of `calloc()`, we might see "garbage" uninitialized data, as shown by the question marks below. Likely these will be empty strings or zeroes.

```
(C)(IC)(???)
(ECHNIC)(TECHNIC)(IC)
(?.??)(?.??)(2.718)
```

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
```

SOLUTIONS: There is only one possible output, as shown below.

```
ONE
ERROR: ONE
==> 0
TWO
==> 2
THREE
ERROR: THREE
```

If we redirect `stdout` to a file, we see the following:

```
bash$ ./a.out > output.txt
ERROR: ONE
bash$ cat output.txt
ERROR: THREE
ONE
==> 0
TWO
==> 2
THREE
```

The above behavior occurs because the `dup2()` call copies the file descriptor table entry for descriptor 1 to descriptor 2, so from that point forward, all output sent to `stderr` also goes to the output file.

Further, since `stderr` remains non-buffered (while file output is block-buffered), the first line of the output file is `ERROR: THREE`.

If we instead redirect `stderr` to a file, we see the following:

```
bash$ ./a.out 2> output.txt
ONE
==> 0
TWO
==> 2
THREE
ERROR: THREE
bash$ cat output.txt
ERROR: ONE
```

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?

SOLUTIONS: There is only one possible output and one possibility for what the `output.txt` file will contain, as shown below.

```
bash$ ./a.out
HELLO
==> 2
WHAT?
bash$ cat output.txt
ERROR
```

The above output occurs primarily because the `open()` call uses the lowest-numbered file descriptor available, which is 2 in this case, causing output sent to `stderr` to redirect to the file.

The mode or permissions of the `output.txt` file will be set to `-rw-r--r--` unless the file already exists. Also, the current `umask` in the terminal may limit the permissions if the file does not already exist.

If the second `close()` call is uncommented, we have the following behavior (since the opened file uses file descriptor 1 this time):

```
bash$ ./a.out
HELLO
bash$ cat output.txt
==> 1
WHAT?
```

SOLUTIONS (CONTINUED): Finally, to redirect all `stdout`/`stderr` output to the output file, make use of `dup2()`. One possible solution is shown below. If we assume we cannot change the given code, we first change the buffering behavior of `stdout` to be block-buffered via `setvbuf()`. Next, we add a `dup2()` call and, since we changed `stdout` to block-buffered, we must include a call to `fflush()`.

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

  int main()
  {
    close( 2 );
    setvbuf( stdout, NULL, _IOFBF, 0 );  /* <=== */
    printf( "HELLO\n" );

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

    int fd = open( "output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644 );

    int rc = dup2( 1, 2 );  /* <=== */

    if ( rc == -1 )
    {
      perror( "dup2() failed" );
      return EXIT_FAILURE;
    }

    printf( "==> %d\n", fd );
    printf( "WHAT?\n" );
    fprintf( stderr, "ERROR\n" );

    fflush( stdout );  /* <=== */

    close( fd );

    return EXIT_SUCCESS;
  }
```

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?

SOLUTIONS: (See the solutions to the previous question for more details.)

Given that there is no output after the `fork()` call through to the `waitpid()` call in the parent process, there is still only one possible output and one possibility for what the `output.txt` file will contain, as shown below.

```
bash$ ./a.out
HELLO
==> 2
WHAT?
SUNS 800?
BYE 800
800
BYE 777
bash$ cat output.txt
ERROR
ERROR 800 ERROR
HELLO
HELLO
```

And if the second `close()` call is uncommented, we have the following behavior (since the opened file uses file descriptor 1 instead of descriptor 2):

```
bash$ ./a.out
HELLO
bash$ cat output.txt
==> 1
WHAT?
SUNS 800?
BYE 800
800
BYE 777
```

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

   SOLUTIONS: When a child process terminates, it becomes a defunct (or zombie) process. It will remain in the process table until its parent process acknowledges its termination via a call to `waitpid()` (or `wait()`). And if a process that has child processes terminates, the child processes become orphan processes that are inherited by the top-level `systemd` process (pid 1) or a `systemd` running for the given user (e.g., `goldsd`). The `systemd` process acts as a "zombie killer" process.

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
   ```

   SOLUTIONS: In brief, the shell process creates a pipe, then creates two child processes. These two child processes attach the pipe to `stdin` and `stdout` such that the first process pipes its output (`stdout`) to the input (`stdin`) of the second process. See the `fork-ps-grep.c` example for a hard-coded implementation.

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 `|`.

   SOLUTIONS: We can pipe together the `ls` and `wc` as follows:

   ```
   bash$ ls -1 *.[ch] | wc -l
   ```

   Read the `man` page for `ls` to see what the `-1` option does.

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?

    SOLUTIONS: Check the `man` page for `fork()` to see all of the specific reasons `fork()` could fail. Two key reasons are insufficient memory and reaching the `RLIMIT_NPROC` soft limit. Therefore, a "fork-bomb" essentially keeps calling `fork()` until no memory remains (and the system grinds to a halt); this can generally be avoided by lowering the soft limit of `RLIMIT_NPROC`.

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

    SOLUTIONS: When we are done with dynamically allocated memory, we should call `free()` at the earliest possible point in our code. If we do not do this, a memory leak occurs, which may cause us later to run out of memory or, more generally, might contribute to buffer overflow errors or other memory-related errors.

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

    SOLUTIONS: Segmentation faults occur when we access invalid memory or memory we do not have permission to access. One example is dereferencing a `NULL` pointer. Another example is invalid pointer arithmetic that leads us far beyond accessible memory. A third example is attempting to modify an immutable character string that is neither on the runtime stack or the runtime heap.

    Try writing code snippets or short code examples for all of these cases.

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

    SOLUTIONS: Memory leaks occur when we dynamically allocate memory and that memory is not freed up or is no longer accessible without being properly freed. One example is simply not calling `free()` when we are done using dynamically allocated memory. Another example is not freeing up a multi-layered structure in the proper order, i.e., in reverse. A third example is when a pointer to dynamically allocated memory goes out of scope or is overwritten to point to something else.

    Try writing code snippets or short code examples for all of these cases.

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

    SOLUTIONS: Buffer overflows occur when we access an array or buffer beyond its defined size. One example is attempting to read or write to a specific array index beyond the end of the array. Another example is using an unsafe library function, such as `scanf()` with `"%s"` as its format string, which will read data without bound. A third example (same as the first example) often occurs when we do not allocate space for the end-of-string termination character `'\0'` and overwrite this byte in a neighboring variable.

    Try writing code snippets or short code examples for all of these cases.

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

    SOLUTIONS: The key reason is efficiency. It is inefficient to repeatedly perform I/O write operations. Further, for `stdout`, the assumption generally is that a human user is at a terminal, so seeing lines of output makes sense here—i.e., line-buffered output. This is why when we redirect to an output file, we switch to block-buffered output.

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.