CSCI 4210 — Operating Systems
Exam 1 (June 29, 2021)

START TIME: _____ END TIME: _____
(Submitty will not show how much time you have remaining, so keep track...)

Overview

- 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
- This is a 120-minute exam, but you will have a full **three-hour window** of time to complete and submit your exam solutions; if you have extra-time accommodations, then you have either a 4.5-hour (50%) or six-hour (100%) window and can go beyond the 2:10AM EDT end time
- There are eight free response questions and one auto-graded question on this exam; for free response questions, please place all of your answers in **a single PDF file called** `exam1.pdf`; for the auto-graded question, submit your code in a single `Q9.c` file
- Exam 1 will be graded out of 100 points; 32 points are auto-graded; therefore, 68 points will be manually graded
- This exam 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 (but do **not** post any questions during the exam)
- **Please do not search the Web for answers;** follow the given instructions carefully and only use the techniques taught in this course
- 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 copy or communicate with anyone else about the exam both during and for at least three days after the exam**
- **Any copying or collaborating with others will result in a grade of zero on this exam; this includes posting in the Discussion Forum or other public or private forums**
- Once we have graded your exam, solutions will be posted; the grade inquiry window for this exam will then be one week

Submitting your exam answers

- Please combine all of your work on free response questions into **a single PDF file called** `exam1.pdf` that includes all pages of this exam
- Also submit the additional code file, i.e., your `Q9.c` source code file
- You **must** submit your two exam file(s) within the three-hour window on Submitty (or within the extended window if you have accommodations for additional time)
- You will have 20 penalty-free submissions, after which points will start to be deducted
- If you face any logistical problems during the exam, please email `goldschmidt@gmail.com` directly with details

Assumptions

1. Assume that all system calls complete successfully (unless otherwise noted)
2. Assume that all code runs on a 64-bit architecture (unless otherwise noted)
3. Assume that processes initially run with a process ID (`pid`) of 256; child process IDs are then numbered sequentially as they are created starting at 512 (i.e., 512, 513, 514, etc.)

Academic Integrity Confirmation

Please sign or indicate below to confirm that you will not copy and you will not cheat on this exam, which also means that you will not communicate with anyone under any circumstances about this exam:

Signature or Typed Name: _____

Failure to submit this page will result in a grade of 0 on the exam.

1. **(4 POINTS)** In the code below, where is the memory for the `summer` variable allocated? Circle the **best** answer.

```
int main()
{
  char * summer = "Pleeeez giMME a BREAk...";
  strcpy( summer + strlen( summer ), "c'mon pLeASE?" );

  /* ... */

}
```

(a) Memory for `summer` is allocated within the immutable data segment

(b) Memory for `summer` is allocated on the runtime stack

(c) Memory for `summer` is allocated within a shared memory segment

(d) Memory for `summer` is allocated on the runtime heap

(e) Memory for `summer` is allocated to an output file out on disk

(f) I have no memory of this

2. **(5 POINTS)** For the code below, assume the `data.txt` file exists and is readable; also assume the `output.txt` file does not initially exist. When you run this code, which file descriptor is the write end of the pipe? Circle the **best** answer.

```
int main()
{
  int p[2];
  close( 0 );
  open( "data.txt", O_RDONLY );
  open( "output.txt", O_WRONLY | O_CREAT, 0600 );
  pipe( p );

  /* which descriptor is the write end of the pipe? */

}
```

(a) `0`

(b) `1`

(c) `2`

(d) `3`

(e) `4`

(f) `5`

3. **(5 POINTS)** How many total bytes are allocated on the runtime heap at the **completion** of the code snippet shown below? Circle the **best** answer.

```
int main()
{
  char ** dcc = calloc( 17, sizeof( char * ) );
  *(dcc + 3) = malloc( 27 * sizeof( char ) );
  dcc = realloc( dcc, 33 * sizeof( char * ) );
  *(dcc + 6) = calloc( 27, sizeof( char ) );

  /* how many bytes are allocated on the heap here? */

}
```

(a) `308`

(b) `318`

(c) `324`

(d) `330`

(e) `337`

(f) `454`

4. **(5 POINTS)** Assume that local file `alpha.txt` contains the 26 uppercase letters of the alphabet followed by a newline character (i.e., `"ABCDEFGHIJKLMNOPQRSTUVWXYZ\n"`). What would the code snippet below show as terminal output? Circle the **best** answer.

```
int main()
{
  int fd = open( "alpha.txt", O_RDONLY );
  lseek( fd, 2, SEEK_CUR );
  char * buffer = calloc( 10, sizeof( char ) );
  read( fd, buffer, 1 );
  read( fd, buffer + 1, 2 );
  read( fd, buffer + 2, 4 );
  buffer++;
  fprintf( stderr, ">> %c", *(buffer+2) );

  /* ... */

}
```

(a) `>> D`

(b) `>> E`

(c) `>> F`

(d) `>> G`

(e) `>> H`

(f) `>> I`

5. **(14 POINTS)** Consider the C code below.

```
void f1( int s ) { printf( "4!3!2!1!" ); fflush( stdout ); }
void f2( int s ) { fprintf( stderr, "==SU" ); }
int main()
{
  signal( SIGUSR1, f1 );
  signal( SIGUSR2, f2 );
  close( 0 );
  close( 1 );
  int * p = calloc( 2, sizeof( int ) );
  pipe( p );
  printf( "NS--" ); fflush( stdout );
  sleep( 1 );
  fprintf( stderr, "<MEME>" );
  sleep( 1 );
  char * buffer = calloc( 8, sizeof( char ) );
  read( p[0], buffer, 4 );
  fprintf( stderr, "%s", buffer );
  read( p[0], buffer, 2 );
  fprintf( stderr, "in%s", buffer );
  /* ====== see part (b) below ====== */
  close( p[0] );
  close( p[1] );
  free( buffer ); free( p );
  return EXIT_SUCCESS;
}
```

(a) What is the **exact** terminal output of this code if a `SIGUSR1` signal is sent to this process during the first `sleep()` call, then a `SIGUSR2` signal is sent to this process during the second `sleep()` call? If no terminal output occurs, write `<empty>`.

(b) Continuing from part (a) above, show the exact contents of the pipe directly before the pipe descriptors are closed (i.e., where the comment appears in the given code). If the pipe is empty, write `*empty*`.

6. **(5 POINTS)** What is the **exact** terminal output of the code below if a `SIGTERM` signal is sent to this process during the `sleep()` call? Circle the **best** answer.

```
void summer( int s )
{
  fprintf( stderr, "-SUMMER-" );
}

int main()
{
  signal( SIGUSR1, SIG_IGN );
  signal( SIGTERM, summer );
  printf( "NO-MORE" );
  sleep( 1 );
  signal( SIGTERM, SIG_DFL );
  printf( "ARCH" );
  return EXIT_SUCCESS;
}
```

(a) `NO-MORE-SUMMER-ARCH`

(b) `-SUMMER-NO-MOREARCH`

(c) `NO-MOREARCH-SUMMER-`

(d) `-SUMMER-ARCH`

(e) `NO MOREARCH`

(f) No output (due to the `SIGTERM` signal)

7. **(5 POINTS)** In the code below, what specific type of error occurs? If multiple types of errors occur, which type of error occurs first? Circle the **best** answer.

```
int main()
{
  int i = 0;
  pid_t buffer1[256];
  int buffer2[256];

  while ( 1 )
  {
    int j = 0;
    pid_t p7 = fork();
    pid_t * careful = malloc( sizeof( pid_t ) );
    *careful = getpid();
    buffer1[i] = getpid();
    buffer2[j] = i;

    if ( p7 > 0 && waitpid( p7, NULL, 0 ) )
    {
      printf( "PARENT: %d %d %d\n", getpid(), *careful, buffer1[i++] );
    }
    else if ( p7 == 0 )
    {
      printf( "CHILD: %d %d\n", *careful, buffer2[j] );
      free( careful );
      break;
    }

    free( careful );
    sleep( 1 );
  }

  return EXIT_SUCCESS;
}
```

(a) A segmentation fault

(b) A buffer overflow

(c) A fork-bomb

(d) A memory leak

(e) Submitty `/tmp` error

(f) Summer Arch

8. **(25 POINTS)** Consider the C code below.

```
int main()
{
  int a = sizeof( int * );
  int * r = malloc( sizeof( int ) );
  float * c = malloc( sizeof( float ) );
  double ** h = calloc( 32, sizeof( double * ) );

  *c = 8.88;
  *r = sizeof( short );
  pid_t p3 = fork();
  printf( "{%d}{%d}{%.4f}{%d}{", a, *r, *c, p3 );

  if ( p3 > 0 )
  {
    h[a] = calloc( 16, sizeof( double ) );
    *r = a + *r;
    printf( "summer %d}\n", a );
    waitpid( p3, NULL, 0 );
  }
  else if ( p3 == 0 )
  {
    h[a] = calloc( 8, sizeof( double ) );
    *r = a - *r;
    printf( "square %d}\n", *r );
    h[a][*r] = 20.21;
  }

  printf( "{%0.1lf}\n", h[a][a-2] );
  return EXIT_SUCCESS;
}
```

(a) After the last `printf()` call in the parent process, how many bytes are allocated on the runtime heap?

(b) After the last `printf()` call in the child process, how many bytes are allocated on the runtime heap?

(c) The given code has multiple memory leaks. Without changing the given code, add calls to `free()` to remove all memory leaks. Show (via arrows) where the `free()` calls would be added, each of which must be added at the **earliest possible point in the code**.

(d) For the code on the previous page, show all possible terminal outputs.

(e) Again for the code on the previous page, if executed as follows, show all possible file contents for the `output.txt` file.

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

9. **(31 POINTS)** Place all code for this problem in a `Q9.c` source file. You are **not** allowed to use square brackets anywhere in your code. If a `'['` or `']'` character is detected, including within comments, that line of code will be removed before running `gcc`.

   In `Q9.c`, write code that attempts to parallelize the work of counting the number of occurrences of a given substring in the `data.txt` input file. Your program creates n child processes, where n is given as the first command-line argument.

   Each child process is responsible for counting the number of times substring s occurs within a specific part of the file. Here, s is given as the second command-line argument. And as an example, the substring `"MMM"` occurs three times in the string `"MMMMxyzMMxyzMMM"`; use a simple brute force algorithm here.

   To distribute the work to n child processes, you must first determine the size of the given file by calling `stat()`. Divide this size by n (using integer division) and allocate that number of adjacent bytes to each child process. The last child process created might therefore work on a slightly smaller or larger byte range.

   As an example, if the `data.txt` file is 1000 bytes in size and n is 5, then the first child process will work on the first 200 bytes of the file, the second child process will work on the second 200 bytes of the file, etc. If instead n is 3, then the first two child processes will work on adjacent 333-byte chunks of the file, while the last child process will work on the remaining 334 bytes of the file.

   Substrings that span two different chunks of memory should not be counted! As such, do not implement inter-process communication beyond the `fork()` and `waitpid()` calls.

   Each child process outputs its findings using the format shown below:

```
bash$ ./a.out 5 th
Found 2 occurrences of "th" in byte range 0-199
Found 5 occurrences of "th" in byte range 200-399
Found 0 occurrences of "th" in byte range 400-599
Found 0 occurrences of "th" in byte range 600-799
Found 4 occurrences of "th" in byte range 800-999
```

   As another example:

```
bash$ ./a.out 3 th
Found 6 occurrences of "th" in byte range 0-332
Found 1 occurrence of "th" in byte range 333-665
Found 4 occurrences of "th" in byte range 666-999
```

   Be sure to call `waitpid()` only after all child processes are created. As a result, output lines could be interleaved since all child processes are running independently of one another.

   Each child process must call `open()`, `read()`, `lseek()`, and `close()` on the `data.txt` input file. The parent process must call `stat()` and `fork()`; it must also call `waitpid()` for each of its child processes before exiting.

   Note that the hidden test cases are all successful program executions.