CSCI 4210 — Operating Systems
Exam 1 (March 10, 2021)
SOLUTIONS (v1.1)

Overview

- Exam 1 is scheduled for the extended window that starts at 6:55PM ET on Wednesday 3/10 and ends at 2:00AM ET on Thursday 3/11
- This is a 90-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 2:00AM ET end time
- Submitty will start the "clock" for you when you **first download** the exam, so please plan accordingly by avoiding any distractions or interruptions; **and note that you must start your exam by 11:00PM ET on Wednesday 3/10 to have your full window of time**
- There are eight free response questions and two auto-graded questions on this exam; for auto-graded questions, you will submit code files `Q9.c` and `Q10.c`; for free response questions, please place all of your answers in **a single PDF file called** `upload.pdf`
- Exam 1 will be graded out of 100 points; 41 points are auto-graded; 59 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 concise 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 48 hours after the exam**
- **Any 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**
- Once we have graded your exam, solutions will be posted; the grade inquiry window for this exam will be one week

Submitting your exam answers

- Please combine all of your work on free response questions into **a single PDF file called** `upload.pdf` that includes all pages of this exam
- Also submit two additional files, i.e., your `Q9.c` and `Q10.c` source code files
- You **must** submit your three exam file(s) within the three-hour window on Submitty (or within the extended window if you have accommodations for additional time)
- 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 128; child process IDs are then numbered sequentially as they are created starting at 200 (i.e., 200, 201, 202, 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. **(3 POINTS)** In the code below, where is the memory for the `spring` variable allocated? Circle the **best** answer.

```
int main()
{
  char spring[2021] = "GiMME a BREAk";
  strcpy( spring + 100, "PLeASE" );

  /* ... */

}
```

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

(b) Memory for `spring` is allocated on the runtime stack `<<<<<<<<<<<<<<<< CORRECT (3pts)`

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

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

(e) I have no memory of this

2. **(3 POINTS)** For the code below, assume the `data.txt` file exists and is readable; also assume the `output.txt` file does not 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( 1 );
  close( 2 );
  open( "data.txt", O_RDONLY );
  open( "output.txt", O_WRONLY | O_CREAT, 0600 );
  pipe( p );

  /* ... */

}
```

(a) `0`

(b) `1`

(c) `2`

(d) `3`

(e) `4 <<<<<<<<<<<<<<<< CORRECT (3pts)`

(f) `5`

3. **(3 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( 11, sizeof( char * ) );
  *(dcc + 3) = calloc( 22, sizeof( char ) );
  dcc = realloc( dcc, 33 * sizeof( char * ) );
  *(dcc + 6) = calloc( 22, sizeof( char ) );

  /* ... */

}
```

(a) 308 <<<<<<<<<<<<<<< CORRECT (3pts)
(b) 318
(c) 324
(d) 330
(e) 337
(f) 396

4. **(3 POINTS)** Assume that local file `alpha.txt` contains the 26 lowercase letters of the alphabet followed by a single 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, 1 );
  read( fd, buffer + 2, 2 );
  buffer++;
  printf( ">> %c\n", *buffer );

  /* ... */

}
```

(a) >> a\n
(b) >> b\n
(c) >> c\n
(d) >> d\n <<<<<<<<<<<<<<< CORRECT (3pts)
(e) >> e\n
(f) >> f\n

5. **(8 POINTS)** Consider the C code below.

```
void f1( int s ) { fprintf( stderr, "-I-LIKE-" ); }

void f2( int s ) { printf( "-MOUSE-" ); fflush( stdout ); }

int main()
{
  int p[2];
  pipe( p );
  dup2( p[1], 1 );
  signal( SIGUSR1, f1 );
  signal( SIGUSR2, f2 );
  printf( "COFFEE" );
  fflush( stdout );
  sleep( 1 );
  printf( "K-A" );
  fflush( stdout );
  sleep( 1 );
  char * buffer = calloc( 3, sizeof( char ) );
  close( p[1] );
  char * ptr = buffer;
  read( p[0], buffer, 2 );
  fprintf( stderr, "%s%sA-PU", buffer, ptr++ );
  read( p[0], buffer, 2 );
  fprintf( stderr, "%sS\n", buffer );
  free( buffer );
  close( p[0] );
  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>`.

SOLUTION: This code will output the following line to the terminal:

```
-I-LIKE-COCOA-PUFFS
[2pts]  -I-LIKE-       (or only 1pt if not exact match)
[2pts]  COCOA-PUFFS    (or only 1pt if not exact match)
```

(b) Show the exact contents of the pipe when this process terminates. If the pipe is empty, just write `<empty>`.

SOLUTION: The pipe contains the following at process termination:

```
[4pts]  EEK-A              (v1.1) --or-- [3pts]  <empty>
          --or--
[4pts]  EEK-A-MOUSE-
```

6. **(21 POINTS)** Consider the C code below.

```
 1:  int main()
 2:  {
 3:    int t = sizeof( int * );
 4:    int * r = malloc( sizeof( int ) );
 5:    float * i = malloc( sizeof( float ) );
 6:    double ** x = calloc( 20, sizeof( double * ) );
 7:
 8:    *i = 8.88;
 9:    *r = sizeof( short );
10:    pid_t p = fork();
11:    printf( "{%d}{%d}{%.3f}{%d}{", t, *r, *i, p );
12:
13:    if ( p == 0 )
14:    {
15:      x[t] = calloc( 20, sizeof( double ) );
16:      *r = t - *r;
17:      printf( "you're a %d}\n", *r );
18:      x[t][*r] = 3.10;
19:    }
20:    else if ( p > 0 )
21:    {
22:      x[t] = calloc( 40, sizeof( double ) );
23:      *r = t + *r;
24:      printf( "square %d}\n", t );
25:      waitpid( p, NULL, 0 );
26:    }
27:
28:    printf( "{%0.2lf}\n", x[t][t-2] );
29:    return EXIT_SUCCESS;
30:  }
```

(a) How many bytes are allocated on the runtime heap in the parent process?
SOLUTION: (2pts) 488 bytes (no partial credit)

(b) How many bytes are allocated on the runtime heap in the child process?
SOLUTION: (2pts) 328 bytes (no partial credit)

(c) The given code has multiple memory leaks. Without changing the code already shown, add the necessary calls to `free()` to remove all memory leaks. Be sure to show (via arrows) where the `free()` calls would be added, each of which must be added at the **earliest possible point in the code**.
SOLUTION: (no partial credit)

```
[1pt]   free( i );     /* line 11; after first printf() */
[1pt]   free( r );     /* line 18; after assigning 3.10 in child */
[1pt]   free( r );     /* line 23; after assignment statement in parent */
[1pt]   free( x[t] );  /* line 28; after last printf() */
[1pt]   free( x );     /* line 28; after last printf() and free( x[t] ) */
```

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

```
--<parent>--------------------         --<child>----------------------
{8}{2}{8.880}{200}{square 8}   <=====> {8}{2}{8.880}{0}{you're a 6}
                               <=====> {3.10}
--waitpid()-------------------         ------------------------------
   |
{0.00}


[2pts]  {8}{2}{8.880}{200}{square 8}
[2pts]  {8}{2}{8.880}{0}{you're a 6}
[1pt]   {3.10}
[1pt]   Interleaving of the two sets of outputs is clear
[2pts]  Last line is always {0.00}
```

(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
```

SOLUTION: Given the `waitpid()` call, the child process will always terminate before the parent process; therefore, all child output is written to the file before any parent output is written to the file.

Output is always:

```
{8}{2}{8.880}{0}{you're a 6}
{3.10}
{8}{2}{8.880}{200}{square 8}
{0.00}

[2pts]  Interleaving does not occur
[2pts]  Output is shown above (deduct points only if incorrect order)
```

7. **(6 POINTS)** The code below is an example of what? Circle the **best** answer.

```
int main()
{
  pid_t buffer[8];

  while ( 1 )
  {
    int i = 0;
    pid_t p = fork();
    pid_t * careful = malloc( sizeof( pid_t ) );
    *careful = getpid();
    buffer[i] = getpid();

    if ( p > 0 && waitpid( p, NULL, 0 ) )
    {
      printf( "PARENT: %d %d\n", *careful, buffer[i++] );
    }
    else if ( p == 0 )
    {
      printf( "CHILD: %d\n", *careful );
    }

    free( careful );
    sleep( 1 );
  }

  return EXIT_SUCCESS;
}
```

(a) A buffer overflow

(b) A segmentation fault

(c) A fork-bomb <<<<<<<<<<<<<<<< CORRECT (6pts)

(d) A memory leak

(e) Submitty source code

8. **(12 POINTS)** For the processes described below, apply the 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., if `P14` and `P16` are "tied," then `P14` is placed in the queue before `P16`).

Process ID	CPU burst time	Arrival time
P1	5 ms	0
P2	11 ms	1 ms
P3	5 ms	0
P4	2 ms	2 ms
P5	7 ms	2 ms
P6	3 ms	22 ms

Fill in the two tables below. We will assign **1 point per row**; no partial credit.

SJF:

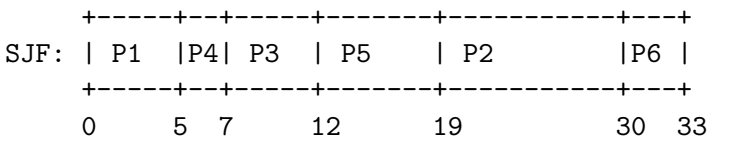
Process ID	Turnaround Time	Wait Time	# of Preemptions
P1	5 ms	0 ms	0
P2	29 ms	18 ms	0
P3	12 ms	7 ms	0
P4	5 ms	3 ms	0
P5	17 ms	10 ms	0
P6	11 ms	8 ms	0

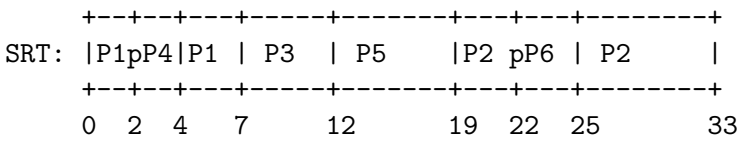
<<<<<<<<< (v1.1) see next page for SJF and SRT diagrams >>>>>>>

SRT:

Process ID	Turnaround Time	Wait Time	# of Preemptions
P1	7 ms	2 ms	1
P2	32 ms	21 ms	1
P3	12 ms	7 ms	0
P4	2 ms	0 ms	0
P5	17 ms	10 ms	0
P6	3 ms	0 ms	0

(v1.1) see below for SJF and SRT diagrams





9. **(20 POINTS)** Place all code for this problem in a `Q9.c` source file.

   Write a program to read at most m bytes from a given `data.txt` file, where m is given as the first command-line argument (i.e., `argv[1]`). All bytes must be sent via a pipe to a child process.

   In the child process, bytes read from the pipe are filtered such that only digits are displayed to `stdout`, all on one line. The child process must also display a second line of output that states how many *total* bytes were read from the pipe, including newlines and any hidden (unprintable) characters, as well as how many digits were filtered out.

   As an example, assume the `data.txt` file contains the following two lines, each ending in a newline character:

```
abcd1932efghijk4291lmnopqrstuv2582wxyz!abcdefg3223hijklmnop2033qrstuvwxyz
ab2199cdefghijklmnopqrstu2322v5558wxyz#abc1234defghijk2234lmnopqrstuvwxyz
```

   The child process would receive all of these characters on the pipe and display two lines of output on `stdout`, as shown below:

```
bash$ ./a.out 200
193242912582322320332199232255581234 2234
(filtered 40 digits from 148 bytes)
```

   As another example:

```
bash$ ./a.out 8
1932
(filtered 4 digits from 8 bytes)
```

   And another example:

```
bash$ ./a.out 1

(filtered 0 digits from 1 byte)
```

   The parent process must call `waitpid()` before exiting.

   Perform basic error-checking, returning `EXIT_SUCCESS` or `EXIT_FAILURE` accordingly in both the parent and child processes; however, note that the hidden test cases are all successful program executions.

10. **(20 POINTS)** Place all code for this problem in a `Q10.c` source file.

    Write a program to parallelize the work of counting the number of occurrences of a specific character in a given `data.txt` input file.

    More specifically, your program creates n child processes, where n is given as the first command-line argument (i.e., `argv[1]`). Each child process is responsible for counting the number of times character c occurs within a specific part of the file. Here, c is given as the second command-line argument (i.e., `argv[2]`).

    To distribute the work to n child processes, you must first determine the size of the given file by using `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.

    Each child process outputs its findings, as shown in the sample program execution below:

```
bash$ ./a.out 5 G
Counted 47 occurrences of 'G' in byte range 0-199
Counted 1 occurrence of 'G' in byte range 200-399
Counted 14 occurrences of 'G' in byte range 400-599
Counted 0 occurrences of 'G' in byte range 600-799
Counted 19 occurrences of 'G' in byte range 800-999
```

    As another example:

```
bash$ ./a.out 3 G
Counted 48 occurrences of 'G' in byte range 0-332
Counted 14 occurrences of 'G' in byte range 333-665
Counted 19 occurrences of 'G' in byte range 666-999
```

    Parallelize the above output; therefore, lines could be interleaved in any order 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()`; it must also call `waitpid()` for each of its child processes before exiting.

    Perform basic error-checking, returning `EXIT_SUCCESS` or `EXIT_FAILURE` accordingly in both the parent and child processes; however, note that the hidden test cases are all successful program executions.