CSCI 4210 — Operating Systems
Exam 2 — April 14, 2021 (v1.2)
SOLUTIONS

Overview

- Exam 2 is scheduled for the extended window that starts at 6:55PM ET on Wednesday 4/14 and ends at 2:00AM ET on Thursday 4/15
- 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 4/14** to have your full window of time
- There are nine free response questions and one auto-graded question on this exam; for the auto-graded question, you will submit code file `Q10.c`; for free response questions, please place all of your answers in **a single PDF file called** `upload.pdf` (this is worth **1 point**)
- Exam 2 will be graded out of 100 points; 36 of these points are auto-graded, and the remaining 64 points will be manually graded; also, you have 20 submissions before points are deducted
- 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 please 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 0 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 the one additional code file, i.e., your `Q10.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)
- As with the homeworks, you will have 20 submissions before points start to be deducted
- If you face any logistical problems during the exam, please email `goldschmidt@gmail.com` directly with details

Assumptions for free response questions

1. Assume that all system calls and library functions 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.)
4. Assume that child threads are numbered sequentially as they are created starting at 400 (i.e., 400, 401, 402, 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. Alternatively, you may copy-and-paste this statement and type your name into a text file that you submit on Submitty.

Signature or Typed Name: ______________________________

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

1. **(4 POINTS)** When you call `pthread_exit()`, what happens? Circle the **best** answer.

   (a) The thread is joined back into the corresponding parent thread
   (b) The thread detaches by terminating and returning `NULL`
   (c) The thread is acknowledged as being terminated
   (d) The thread exits via the `_exit()` system call
   (e) The thread terminates and returns a specified value `<<<<<<<<<<<< CORRECT (4pts)`
   (f) The thread disconnects from its parent thread (i.e., it exits the thread pool)

2. **(4 POINTS)** When you call `shmat()`, what happens? Circle the **best** answer.

   (a) The process is attached to a shared memory segment with the given shared memory ID
   `^^^^^^^^^^^^^^^^ CORRECT (4pts)`
   (b) Given a shared memory key, the shared memory ID is determined (and returned)
   (c) A shared memory segment is created and attached to the process
   (d) The process detaches from the already attached shared memory segment
   (e) A shared memory segment is attached with a specified size (in bytes) to the process
   (f) Given a shared memory ID, the size of the shared memory segment is returned

3. **(4 POINTS)** Which of these "solutions" solves the Dining Philosopher's problem, i.e., is fair and prevents starvation? Circle the **best** answer.

   (a) If there is an argument about who gets to eat, everyone except the philosopher with the longest name puts down their chopstick(s)
   (b) If each philosopher is holding exactly one chopstick, the spaghetti gets taken away
   (c) Everyone who is holding exactly one chopstick must put it down and try to pick up the other chopstick
   (d) If each philosopher is holding exactly one chopstick, each philosopher counts backwards from 1000, then puts their chopstick down and attempts to pick it up again
   (e) Only one philosopher is allowed to eat at the table at any given time `<<<< CORRECT (4pts)`
   (f) The philosophers put down all chopsticks, take a vote on who does not get to eat, and the one with the most votes is no longer allowed to pick up any chopsticks for 10 seconds

4. **(5 POINTS)** How many total bytes remain allocated on the runtime heap at the **completion** of the code snippet shown below (i.e., directly before the `return EXIT_SUCCESS` statement)? Circle the **best** answer.

```
int main()
{
  char *** low = calloc( 375, sizeof( char ** ) );
  *(low + 4) = calloc( 7, sizeof( char * ) );
  *( *(low + 4) ) = calloc( 5, sizeof( char ) );
  *( *(low + 4) + 1 ) = calloc( 7, sizeof( char ) );
  *(low + 4) = realloc( *(low + 4), 4 * sizeof( char * ) );
  *( *(low + 4) ) = calloc( 7, sizeof( char ) );

  /* ... */

  return EXIT_SUCCESS;
}
```

(a) 3039

(b) 3045

(c) 3051 <<<<<<<<<<<<<<<< CORRECT (5pts)

(d) 3112

(e) 3116

(f) 3130

5. **(5 POINTS)** How many **total** bytes are allocated on the runtime heap at the **completion** of the code shown below (i.e., after the call to `free()` in `main()` but before the main thread terminates)? Circle the **best** answer.

```
void * q5( void * x )
{
  char * five = calloc( *((int *)x) + 202, sizeof( char ) );
  for ( int i = 1 ; i <= sizeof( five ) ; i++ )
  {
    sprintf( five, "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
    five = realloc( five, 90 + 135 * i );
  }
  return NULL;
}

int main()
{
  int * sage = calloc( 3101, sizeof( int ) );
  pthread_t t1, t2, t3;
  pthread_create( &t1, NULL, q5, sage );
  pthread_create( &t2, NULL, q5, sage );
  pthread_create( &t3, NULL, q5, sage );
  pthread_join( t3, NULL );
  pthread_join( t2, NULL );
  pthread_join( t1, NULL );
  free( sage );
  return EXIT_SUCCESS;
}
```

(a) 3101

(b) 3303

(c) 3510 `<<<<<<<<<<<<<<<< CORRECT (5pts)`

(d) 3704

(e) 4101

(f) 5101

6. **(5 POINTS)** Consider the C code below. Assume that initially there is no shared memory segment with key 4000. If you run the given code three times in a row, what is the **exact** terminal output of the third and final program execution? Circle the **best** answer.

```
int main()
{
  key_t key = 4000;
  int shmid = shmget( key, sizeof( int ), IPC_CREAT | 0660 );
  int * six = shmat( shmid, NULL, 0 );
  pid_t pid = fork();

  if ( pid > 0 ) waitpid( pid, NULL, 0 );

  int i, stop = key / 1000;
  for ( i = 1 ; i <= stop ; i++ ) *six += i;

  if ( pid > 0 ) printf( "Sum is %d\n", *six );

  shmdt( six );
  return EXIT_SUCCESS;
}
```

(a) `Sum is 0`

(b) `Sum is 10`

(c) `Sum is 20`

(d) `Sum is 30`

(e) `Sum is 40`

(f) `Sum is 50`

(g) `Sum is 60 <<<<<<<<<<<<<<<< CORRECT (5pts)`

(h) `Sum is 70`

(i) `Sum is 80`

(j) `Sum is 90`

(k) `Sum is 100`

7. **(13 POINTS)** Assume the process running the code snippet below successfully attaches to a shared memory segment (of size 777 bytes) in the `seven` variable via the `shmat()` call.

```
char * seven = shmat( shmid, NULL, 0 );
sprintf( seven, "RENT" );
sprintf( seven + 3, "SSSZ" );
sprintf( seven + 5, "ELSE" );
sprintf( seven + 7, "AER1" );
strcat( seven + 8, "ERRRRRRR" );
do_something_else( 7 );
memcpy( seven + 4, "ULLL", 4 );
strcat( seven + 9, "RRR!" );  /* (v1.1) removed extraneous third argument */
```

(a) During the `do_something_else()` call, a separate process attaches to the same shared memory segment and runs the code snippet shown below. Note that `stdout` is set to output to the terminal.

```
setvbuf( stdout, NULL, _IONBF, 0 );
char * huh = shmat( shmid, NULL, 0 );
while ( *huh && !isdigit( *huh ) ) printf( "%c", *huh++ );
```

What is the **exact** output of this second process? If there are multiple possibilities, show all possible outputs.

SOLUTION: Correct output is "`RENSSELAER`"; each character in "`RENSSELAER`" is worth 1 point. Subtract 1 point for each incorrect character, including a newline if specified.

Also give credit if "`RENSULLLER`" is noted as being possible. (If the given code snippet is executing and the `do_something_else()` function call returns, the `memcpy()` could interleave with the output.)

(b) Again during the `do_something_else()` call above, assume that four separate processes simultaneously run the code snippet from part (a). What happens to the output of each of these four processes? Circle the **best** answer.

(i) The first of the four processes to run displays the output from part (a), while the other three processes do not display anything

(ii) All four processes display the same output from part (a) with individual characters potentially interleaved `<<<<<<<<<<<<<<<< CORRECT (4pts)`

(iii) No output is produced by any of the four processes

(iv) All four processes display the same output from part (a) without any interleaving of individual characters

(v) At least one of the four processes crashes due to a segmentation fault

(vi) The four processes each display a portion of the entire shared string, with different characters displayed depending on the order the processes happen to run in

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

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void * q8( void * arg )
{
  int * w = (int *)arg;
  pthread_mutex_lock( &mutex );
  *w += 8;
  pthread_mutex_unlock( &mutex );
  return NULL;
}

int main()
{
  pthread_t tid1, tid2;
  int eight = 4;
  pthread_create( &tid1, NULL, q8, &eight );
  eight = 5;
  pthread_create( &tid2, NULL, q8, &eight );
  eight = 6;
  pthread_join( tid1, NULL );
  pthread_join( tid2, NULL );
  return EXIT_SUCCESS;
}
```

SOLUTION: See the next page for a diagram.

(a) How many distinct values could variable `eight` be set to at the **completion** of the `main()` function in the given code (i.e., directly before the `return EXIT_SUCCESS` statement)? Circle the **best** answer.

- (i) Only one possible value for `eight`
- (ii) Exactly two possible values for `eight`
- (iii) Exactly four possible values for `eight`
- (iv) Exactly five possible values for `eight`
- (v) Exactly three possible values for `eight`
- (vi) Exactly six possible values for `eight` `<<<<<<<<<<<<<<<< CORRECT (2pts)`

(b) What are the distinct value(s) variable `eight` could be set to at the **completion** of the `main()` function in the given code (i.e., directly before the `return EXIT_SUCCESS` statement)?

SOLUTION: Correct values are 6, 14, and 22, **(v1.2)** as well as 13, 20, and 21. Each correct value is worth 2 points. Subtract 2 points for each incorrect value, but stay within 0 to 6 points.

DIAGRAM FOR QUESTION 8:

```
            main()
              |
           eight=4
              |
  tid1 <------|
   |          |
eight+=8   eight=5
   |          |------> tid2
   |          |         |
   |       eight=6   eight+=8
   |          |         |
   |--------->|<--------|
              |
              v
         what is eight?
```

(v1.2) EVERYONE GETS CREDIT FOR THIS QUESTION DUE TO THE GRADING ERRORS FROM v1.1. See the next page for solution details.

(v1.2) The two child threads will not interleave with one another when they attempt to add eight to the `eight` variable through pointer `w`; however, interleaving could occur between a child thread and the main thread, yielding 13, 20, or 21, as shown further below.

```
[ORDER THAT YIELDS eight == 6]
  eight += 8;    <==>  eight = 5;
  eight += 8;    <==>  ^^^ doesn't matter where this interleaves
  eight = 6;

[ORDER THAT YIELDS eight == 14]
  eight += 8;    <==>  eight = 5;
  eight = 6;           ^^^ doesn't matter where this interleaves
  eight += 8;

[ORDER THAT YIELDS eight == 22]
  eight = 5;
  eight = 6;
  eight += 8;
  eight += 8;

[ORDER THAT YIELDS eight == 13]
  eight += 8;
  eight = 5;
  eight = 6;     <==>  eight += 8;
                       ^^^ this update overwrites main()'s update,
                            so eight is 5 + 8

[ORDER THAT YIELDS eight == 20]
  eight = 4;
  eight = 5;     <==>  eight += 8;
                       ^^^ this update overwrites main()'s update,
                            so eight is 4 + 8
  eight = 6;     <==>  eight += 8;
                       ^^^ this update overwrites main()'s update,
                            so eight is 12 + 8

[ORDER THAT YIELDS eight == 21]
  eight = 5;
  eight = 6;     <==>  eight += 8;
                       ^^^ this update overwrites main()'s update,
                            so eight is 5 + 8
  eight += 8;
```

9. **(16 POINTS)** Consider the C code below.

```
char * nine;

void * q9( void * y )
{
  char n = *(char *)y;
  for ( int m = n + n ; m > 0 ; m-- )
  {
    usleep( 10 );  /* <== forces a thread context switch... */
    strncat( nine, &n, 1 );
  }
  pthread_exit( NULL );
  return NULL;
}

int main()
{
  nine = malloc( 9999 );
  nine[0] = '\0';
  pthread_t * t = calloc( 9, sizeof( pthread_t ) );

  for ( int i = 0 ; i < 9 ; i++ )
  {
    char * j = malloc( 1 );
    *j = 'q' + i;
    pthread_create( t + i, NULL, q9, j );
  }

  for ( int i = 0 ; i < 9 ; i++ ) pthread_join( *(t + i), NULL );
  printf( "%s\n", nine );
  return EXIT_SUCCESS;
}
```

(a) How many bytes of output does this program produce in the `printf()` statement?
SOLUTION: See the next page (worth 3 points).

(b) Add code above to free up all memory that is dynamically allocated in the given code. Be sure to place the `free()` calls at the **earliest possible point** in the code. Do not change or remove any of the given code.
SOLUTION: See extended code on the next page (worth 6 points).

(c) This program runs with no synchronization. The global `nine` variable may contain something like `"wxyvutsrqwxyvutsrq..."` and will vary each time we run this code. We instead want to ensure that this program keeps all like characters together (e.g., keep all `'y'` characters together). To accomplish this, identify all critical sections by drawing a rectangle around them in the code. Next, add code to synchronize the child threads. Do not change or remove any of the given code.
SOLUTION: See extended code on the next page (worth 7 points).

SOLUTION TO QUESTION 9(a): The `printf()` statement typically produces 2107 bytes (2 points); however, it is possible for the number of bytes to be less than 2107 if the `strncat()` calls interleave and interfere with one another (1 point).

SOLUTIONS TO QUESTION 9(b) AND 9(c): The additional code shown below must be added exactly where shown except for the declaration of `mutex`, which could also be above the declaration of `nine`. No partial credit for each line of code.

```
char * nine;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* <== 1pt for 9(c) == */

void * q9( void * y )
{
  char n = *(char *)y;
  free( y ); /* <======================================= 2pts for 9(b) == */
  pthread_mutex_lock( &mutex ); /* <=================== 3pts for 9(c) == */
  for ( int m = n + n ; m > 0 ; m-- )
  {
    usleep( 10 );  /* <== forces a thread context switch... */
    strncat( nine, &n, 1 );
  }
  pthread_mutex_unlock( &mutex ); /* <================= 3pts for 9(c) == */
  pthread_exit( NULL );
  return NULL;
}

int main()
{
  nine = malloc( 9999 );
  nine[0] = '\0';
  pthread_t * t = calloc( 9, sizeof( pthread_t ) );

  for ( int i = 0 ; i < 9 ; i++ )
  {
    char * j = malloc( 1 );
    *j = 'q' + i;
    pthread_create( t + i, NULL, q9, j );
  }

  for ( int i = 0 ; i < 9 ; i++ ) pthread_join( *(t + i), NULL );
  free( t ); /* <======================================= 2pts for 9(b) == */
  printf( "%s\n", nine );
  free( nine ); /* <==================================== 2pts for 9(b) == */
  return EXIT_SUCCESS;
}
```

10. **(35 POINTS)** Place all code for this problem in a `Q10.c` source file.

For this problem, you are to write a multi-threaded program in C that gets its instructions from a shared memory segment. The shared memory key is given as the first command-line argument; assume the size of the shared memory to be 32,768 bytes.

In your solution, each child thread reads from a given input file, concatenating valid characters to the "data" portion of the shared memory segment. A valid character is an alphanumeric character as identified via the `isalnum()` library function.

The shared memory segment is structured as follows:

```
      +---+---------------------------+-----------+
 SHM: | G | filename (ends with '\0') | N threads | ...
      +---+---------------------------+-----------+

               +-----------+-----------+-----+-----------+
           ... |  offset1  |  offset2  | ... |  offsetN  | ...
               +-----------+-----------+-----+-----------+

                      +-----------------------------------------+
                  ... | data (concatenate valid characters here) |
                      +-----------------------------------------+
```

The first byte of the shared memory segment is not used until all child threads have completed their work. Once your main thread knows all child threads are done, set the first byte of the shared memory segment to `'G'`. This will let the Submitty test code to evaluate your shared memory segment.

The `filename` is a variable-length string ending with `'\0'`, e.g., `"inputfile.txt"`.

The `N threads` field is a four-byte `int` variable indicating how many child threads you must create. It also tells you how many offset values are present (i.e., `offset1`, `offset2`, etc.). Each child thread is assigned an offset value, which indicates the byte offset to start reading from in the given file.

The `data` portion of the shared memory segment is initially empty (i.e., set to `'\0'` or zero bytes). Each child thread reads from its given offset in the given file, copying exactly 111 valid characters from the file into the shared memory segment. If there are less than 111 valid characters through to the end of the file, copy only the valid characters encountered.

The order in which these characters are added to the shared memory segment does not matter. Submitty will sort this data before validating it.

Be sure to parallelize the child threads above to the extent possible.

You are required to use `pthread_create()`, `pthread_join()`, and `pthread_exit()`. Further, each child thread must at least call `lseek()` and `read()`.

Perform basic error-checking, returning `EXIT_SUCCESS` or `EXIT_FAILURE` from the main thread accordingly; however, note that the hidden test cases are all successful program executions.