CSCI 4210 — Operating Systems
Exam 2 Prep and Sample Questions (document version 1.2)
SELECTED SOLUTIONS

- 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
- Exam 2 will be a 90-minute exam, but you will have a full **three-hour window of time** to complete and submit your exam solutions
- Submitty will start the three-hour "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 the full three-hour window**
- There will be a mix of auto-graded and free response questions on the exam; for auto-graded questions, 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 `upload.pdf`**
- Exam 2 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 2 covers everything through live lecture Thursday 4/8, including all assignments due through Friday 4/9; while Exam 2 is cumulative, approximately 75% of the exam questions will focus on topics covered since Exam 1, i.e., shared memory, multi-threaded programming, Java threads, threads in C using the Pthread library, mutual exclusion, synchronization, and semaphores
- 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

Use the previous sample questions provided to prepare for Exam 1. In addition, practice problems for Exam 2 are 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. What do the following system calls do? What are their input arguments and return values?

   - `shmget()`
   - `shmat()`
   - `shmdt()`
   - `shmctl()`

   SOLUTION: Refer to the `man` pages directly.

2. How can two processes share data via a shared memory segment, i.e., what are the series of system calls required? Since `shmat()` returns a generic pointer (i.e., `void *`), how can we be sure both processes interpret the shared data correctly?

   SOLUTION: Refer to the `count-shm.c`, `fork-octuplets-abcd-shm.c`, and corresponding `fork-octuplets-abcd-shm-separate-process.c` examples directly. In brief, processes must call `shmget()` and `shmat()` to obtain the generic pointer to the shared memory. See Question 4 below for details on `shmdt()`.

   It is up to the processes (through their design and implementation) to correctly interpret the shared data; in other words, the programmer(s) must define the structure of the shared memory and the protocol used to access it.

3. Without any synchronization between the two processes from Question 2 above, what could go wrong?

   SOLUTION: Refer to the coding examples for various examples of synchronization problems. In brief, lines of C source code could interleave between the multiple processes; similarly, assembly-level code could also interleave. This will likely cause data corruption (e.g., the `count-shm.c` example in which the sum is consistently smaller than the expected correct value).

   In addition to data corruption, the second type of synchronization errors involve sequences of instructions executing in the "wrong" order. As an example, a process writing data to a shared memory segment might not completely write all of its necessary data before another process reads the data. The reader then might end up with an incomplete set of data, so note that such sequencing errors often cause data corruption.

4. How is a shared memory segment deleted? And if a shared memory segment is deleted while other processes are attached to it, what happens?

   SOLUTION: From Question 2 above, once done, processes should call `shmdt()` to detach from the shared memory segment. Note that (from `man shmdt`) that "Upon `_exit(2)`, all attached shared memory segments are detached from the process."

   To delete a shared memory segment, a call to `shmctl()` is made with `IPC_RMID` as its second parameter. See the `man` page for `shmctl()`; from this `man` page, "The segment will actually be destroyed only after the last process detaches...."

   Also note from `man shmdt` that "After a `fork(2)`, the child inherits the attached shared memory segments." Further, "After an `execve(2)`, all attached shared memory segments are detached from the process."

5. Write a C program to create a shared memory segment with a known key and a size of 1024 bytes. Write a separate C program to attach to the shared memory segment and write the Fibonacci sequence starting with $F(0) = 0$ and $F(1) = 1$. Use an `unsigned long` data type for each value and either fill up the 1024 bytes or stop when overflow occurs.

   Finally, write a third C program to attach to the shared memory segment and simply display the Fibonacci sequence from the shared memory. As an optional command-line argument, if `-DELETE` is specified (as shown below), the shared memory segment is also deleted by this third program.

   ```
   bash$ ./a.out -DELETE
   ```

6. Write a C program to create a shared memory segment with a known key and a size of 128 bytes. Next, have this program prompt the user to repeatedly enter a line of text. Each line is written to the shared memory segment, truncating the line to fit, if necessary.

   Write a second C program to attach to the shared memory segment and wait for data to be added. When data is written by the first process, this second process downcases all data and displays it.

   Note that the first process must not display the prompt until the second process has downcased and displayed the data. Further, the two processes must be completely separate, i.e., do not use `fork()` to solve this problem.

   As a hint, use the first few bytes of the shared memory segment to relay synchronization information from one process to the other. Create a synchronization protocol to ensure there are no race conditions in your solution. (Do not use semaphores to solve this problem.)

   SOLUTION: Refer to the `Q6writer.c` and `Q6reader.c` source code files directly.

7. What do the following library functions do? What are their input arguments and return values? And how do they get compiled in to your code?

   - `pthread_create()`
   - `pthread_detach()`
   - `pthread_self()`
   - `pthread_join()`
   - `pthread_exit()`

   SOLUTION: Refer to the `man` pages directly. And compile with the `-pthread` flag.

8. In a multi-threaded program, describe at least two types of synchronization errors that can occur. How best can these situations be avoided?

   SOLUTION: Refer to the coding examples for various examples of synchronization problems at the thread level. In brief, lines of C source code (or Java code) could interleave between multiple threads; further, assembly-level code (or Java byte code?) could also interleave.

   To avoid these problems, synchronization is necessary. To synchronize threads, a `mutex` variable can act as a lock that ensures mutual exclusion to a set of critical sections across multiple threads.

9. What is a critical section? Why is it important in multi-threaded programming? Are there critical sections in multi-process programming, i.e., in support of inter-process communication (IPC)?

   SOLUTION: A critical section of code is a block of one or more statements for which mutual exclusion is required. More specifically, when one thread is running in its critical section, no other threads may be in their critical sections.

   Without identifying critical sections and implementing protective synchronization mechanisms, data corruption will likely occur.

   And it does also make sense to identify critical sections in multi-process programs, especially in support of IPC.

   Note that there groups of critical sections are typically associated per shared resource. For example, shared resource `A` might have three critical sections protected via a `mutex_A` variable, while shared resource `B` might have four critical sections protected via a `mutex_B` variable, etc.

   Here, if a critical section requires access to both `A` and `B`, we run the risk of having deadlock.

10. For semaphores, what are the definitions of the `P()` and `V()` operations? Further, how does a binary semaphore differ from a counting semaphore? How are each used to avoid synchronization problems?

    SOLUTION: The `P()` and `V()` operations are defined using the pseudo-code below. Of note, in the `P()` operation, there is never a context switch between exiting the loop and decrementing the semaphore variable. Note that `S` is a non-negative integer variable.

    ```
    P( semaphore S )     /* this P() operation MUST execute without    */
    {                    /*  any interruption, i.e., no context switch */
      while ( S == 0 )   /*   between exiting the while() loop and     */
      {                  /*    executing S--                           */
        /* busy wait */
      }
      S--;
    }

    V( semaphore S )
    {
      S++;
    }
    ```

    A binary semaphore is either 0 or 1; a value of 0 indicates the resource is not available. Binary semaphores are therefore used to implement `mutex` variables that act as a lock.

    A counting semaphore ranges between 0 and n, where n specifies the number of resources to control access to. As above, a value of 0 indicates no such resources are available.

    Note that we typically assume that when multiple processes/threads are blocked on `P()`, they are queued up in the order in which they called `P()` (so FCFS or FIFO).

11. Using `P()` and `V()` operations, write a pseudo-code solution to the Readers/Writers problem for a shared reservation system that governs an airplane with 20 rows of seats, each row consisting of four seats. More specifically, any number of readers can simultaneously read the seating chart, including which seats are available. And while there can be any number of writers, your solution must guarantee that each seat can only be reserved by one person.

    SOLUTION: See the lecture video and notes from April 6 as a reference.

    For the Readers/Writers problem, any number of readers can be reading the shared data at the same time. When a writer wishes to write, we must have mutually exclusive access to write and also must ensure no readers are reading. A pseudo-code solution is below, but it has the potential for starvation.

```
/* shared memory */
semaphore write = 1  /* controls whether a write can occur */
int readers = 0      /* number of active readers */
semaphore mutex = 1  /* mutex for the readers variable */
```

```
Writer( seat_index )                    Reader()
{                                       {
  P( write )                              P( mutex )
    reserve_seat( seat_index )              if ( readers == 0 ) P( write )
  V( write )                                readers++
}                                         V( mutex )
                                          read_seating_chart()
                                          P( mutex )
                                            readers--
                                            if ( readers == 0 ) V( write )
                                          V( mutex )
                                        }
```

In the above solution, if Readers are always trying to read, the solution starves any Writers. The next page has a revised solution to address this problem.

A revised solution is below, which includes an additional `me_next` semaphore. This semaphore ensures that when a Writer would like to write, no new Readers may read.

```
/* shared memory */
semaphore write = 1  /* controls whether a write can occur */
int readers = 0      /* number of active readers */
semaphore mutex = 1  /* mutex for the readers variable */

semaphore me_next = 1  /* queue of Readers and Writers */
```

```
Writer( seat_index )
{
  P( me_next )
    P( write )
      reserve_seat( seat_index )
    V( write )
  V( me_next )
}
```

```
Reader()
{
  P( me_next )
    P( mutex )
      if ( readers == 0 ) P( write )
      readers++
    V( mutex )
  V( me_next )
  read_seating_chart()
  P( mutex )
    readers--
    if ( readers == 0 ) V( write )
  V( mutex )
}
```

Here, all Readers and Writers are queued up waiting to acquire the `me_next` semaphore. When a Writer acquires the `me_next` semaphore, it blocks new Readers from acquiring the semaphore; in the meanwhile, active Readers terminate and eventually the last Reader relinquishes the `write` semaphore (which enables the Writer to call `reserve_seat()`).

Does this solution avoid deadlock?

12. Using `P()` and `V()` operations, write a solution to the Dining Philosophers problem.

    SOLUTION: See the lecture video and notes for April 6.

13. What conditions are required for deadlock to occur? How can you avoid deadlock? Describe at least two techniques for recovering from deadlock.

    SOLUTION: From the lecture video and notes for April 6, deadlock requires four conditions: mutual exclusion; hold and wait; no preemption; and circular wait.

    Avoiding deadlock requires a means of looking ahead to make sure that obtaining a lock on a resource (via `P()`) will not cause a deadlock. We can use a graph to track processes (or threads) and the resources they've acquired and requested.

    To recover from deadlock, all or some of the involved processes (or threads) can be killed. Alternatively, the operating systems can provide a means of rolling back to an earlier checkpoint at which there was no deadlock.

14. Given the following C program, what is the **exact** terminal output? Assume all system calls complete successfully.

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

int main()
{
  close( 0 );
  close( 1 );

  char * s = "ARE YOU READY FOR THE EXAM?";

  int p[2];
  pipe( p );

  pid_t pid = fork();

  if ( pid == 0 )
  {
    write( p[1], s, 20 );
    printf( "%s!\n", s );
  }
  else /* pid > 0 */
  {
    waitpid( -1, NULL, 0 );
    char buffer[1024];
    int rc = read( p[0], buffer, 1024 );
    printf( "(read %d bytes)\n", rc );
    buffer[14] = buffer[18] = buffer[rc] = '\0';
    fprintf( stderr, "%s%sNOT!\n", buffer + 8, buffer + 15 );
    fprintf( stderr, "%s\nHOPEFULLY.\n", buffer + 28 );
  }

  return EXIT_SUCCESS;
}
```

From the shell, how would you redirect terminal output to go to an output file?

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

How would the output change if the two `close()` calls were removed?

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

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <time.h>
#include <string.h>
#include <ctype.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SHARED_KEY 8000

int main( int argc, char * argv[] )
{
  int shmid = shmget( SHM_SHARED_KEY, sizeof( int * ), IPC_CREAT | 0660 );
  int * data = shmat( shmid, NULL, 0 );

  int pid = fork();

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

  int i, stop = 6;

  for ( i = 1 ; i <= stop ; i++ )
  {
    data[i%2] += i;
  }

  printf( "%s: data[%d] is %d\n", pid > 0 ? "PARENT" : "CHILD", 0, data[0] );
  printf( "%s: data[%d] is %d\n", pid > 0 ? "PARENT" : "CHILD", 1, data[1] );

  shmdt( data );

  return EXIT_SUCCESS;
}
```

Add code to delete the shared memory segment such that only one process actually deletes the segment. For this, do not assume that system calls will always return successfully.

What is the **exact** terminal output if the `waitpid()` call is removed? Clearly show all output possibilities.

SOLUTION: To have the parent process delete the shared memory segment, the code below is added after the call to `shmdt()`:

```
if ( pid > 0 )
{
  /* remove the shared memory segment */
  if ( shmctl( shmid, IPC_RMID, 0 ) == -1 )
  {
    perror( "PARENT: shmctl() failed" );
    exit( EXIT_FAILURE );
  }
}
```

With `waitpid()` included, the child process will always write to the shared memory segment and then produce its output before the parent process proceeds. Output is as follows:

```
CHILD: data[0] is 12
CHILD: data[1] is 9
PARENT: data[0] is 24
PARENT: data[1] is 18
```

If the `waitpid()` call is removed, both the updates to the shared memory segment and the output between the parent and child processes are interleaved. Without synchronization, some updates may overwrite one another. Output occurs as follows:

```
                          int pid = fork()
                         /                \
                        /                  \
        PARENT: data[0] is <v1>             CHILD: data[0] is <v3>
        PARENT: data[1] is <v2>             CHILD: data[1] is <v4>
```

Here, `<v1>` and `<v2>` are typically going to be 12 and 9, respectively. Further, `<v3>` and `<v4>` are typically going to be 24 and 18, respectively.

(v1.2) This was previously incorrect—from the chat during the review session, yes, the minimum value of `<v1>` or `<v3>` is 1, which can occur if the first process to access the global `data[0]` reads 0, adds 1 in a register, then gets context-switched out; during this context switch, the other process makes numerous updates and context-switches out before it displays its two lines of output; when we return to the first process, all updates are overwritten since it writes 1 back to `data[0]`; finally, switching back to the second process, it displays 1. Similarly, the minimum value of `<v2>` or `<v4>` is 2.

The corresponding maximum values of `<v1>` and `<v3>` are 24. And the maximum values of `<v2>` and `<v4>` are 18. If the update statements interleave within the `for` loop, though, the maximum values might not be reached.

Note that two synchronization issues can occur here. First, the two pairs of `printf()` calls may occur while the other process is still updating the global `data` variable. Second, the update statement (i.e., `data[i%2] += i`) itself might interleave at the assembly level.

16. Given the following C program, what is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully. Also assume that the main thread ID is 256, while child thread IDs are assigned sequentially starting at 512.

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

void * go( void * arg )
{
  char * s = calloc( strlen( arg ) + 1, sizeof( char ) );
  strcpy( s, arg );
  s[6] = '\0';
  fprintf( stderr, "%s", s );
  return NULL;
}

int main()
{
  int a = 1;
  pthread_t tid;
  char * q = "READY FOR THE EXAM";
  pthread_create( &tid, NULL, go, q );
  if ( a == 1 ) pthread_join( tid, NULL );
  fprintf( stderr, "%s", q );
  if ( a == 2 ) pthread_join( tid, NULL );
  fprintf( stderr, "!\n" );
  return EXIT_SUCCESS;
}
```

How does the output change if variable a is initialized to 2?

How does the output change if variable a is initialized to 3?

SOLUTION: When variable `a` is initialized to 1, no interleaving of instructions is possible. The output is as follows:

```
READY READY FOR THE EXAM!
```

When variable `a` is initialized to 2, interleaving may occur. There are two possible outputs, as shown below, with the first output much more likely to occur. (Try adding a `usleep(5)` after the `pthread_create()` call to see some interleaving.)

```
READY FOR THE EXAMREADY !

READY READY FOR THE EXAM!
```

When variable `a` is initialized to 3, interleaving may occur. **(v1.2)** There are four possible outputs, as shown below, with the first output much more likely to occur. (Try adding a `usleep(5)` after the `pthread_create()` call and elsewhere to see some of the interleaving.)

```
"READY FOR THE EXAM!\n"        <== child thread does not display anything

"READY READY FOR THE EXAM!\n"  <== child thread displays its output first

"READY FOR THE EXAMREADY !\n"  <== child thread displays its output before
                                    final fprintf() in main()

"READY FOR THE EXAM!\nREADY "  <== chlid thread displays its output after
                                    final fprintf() in main()
```

17. Given the following C program, what is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully. Also assume that the main thread ID is `256`, while child thread IDs are assigned sequentially starting at `512`.

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

void * charms( void * arg )
{
  int * s = (int *)arg;
  printf( "%ld lucky %d\n", pthread_self(), *s );
  return NULL;
}

int main()
{
  pthread_t tid1, tid2;
  int rc = -1, x = 7;
  rc = pthread_create( &tid1, NULL, charms, &x );
  rc = pthread_create( &tid2, NULL, charms, &x );
  x = 13;
  rc = pthread_join( tid1, NULL );
  printf( "%d unlucky %d\n", rc, x );
  rc = pthread_join( tid2, NULL );
  return EXIT_SUCCESS;
}
```

SOLUTION: The first child thread is guaranteed to display its output and terminate before the main thread displays its output. Interleaving and output occurs as shown below. (Try adding `usleep(5)` calls after the `pthread_create()` calls to see some interleaving.)

```
<first-child-and-main-thread>          <second-child-thread>
512 lucky <7 or 13>           <====>  513 lucky <7 or 13>
0 unlucky 13                  <====>
```

Note that we see the above output possibilities because the following code is interleaved:

```
x = 7;
pthread_create() calls...
  |                        |                        |
<main>                 <first-child-thread>     <second-child-thread>
x = 13;                *s = arg;                *s = arg;
pthread_join(); <----- printf( ... );           printf( ... );
printf( ... );                                      |
pthread_join(); <-----------------------------------
```

18. Given the following C program, what is the **exact** terminal output? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully. Also assume that the main thread ID is `256`, while child thread IDs are assigned sequentially starting at `512`.

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

void * qwerty( void * arg )
{
  int * q = (int *)arg;
  *q += 6;
  printf( "%ld lucky %d\n", pthread_self(), *q );
  return NULL;
}

int main()
{
  pthread_t tid1, tid2;
  int x = 7;
  int rc = pthread_create( &tid1, NULL, qwerty, &x );
  rc = pthread_create( &tid2, NULL, qwerty, &x );
  x = 13;
  rc = pthread_join( tid1, NULL );
  printf( "%d unlucky %d\n", rc, x );
  rc = pthread_join( tid2, NULL );
  return EXIT_SUCCESS;
}
```

Identify critical section(s) in the code above and use at most one `mutex` variable to implement a fully synchronized solution that avoids corruption with variable `x`. More specifically, add to the code above without changing it or deleting any of the existing code; use arrows to show where the additional code would be added.

Next, write a C program that makes use of a shared memory segment to implement the same functionality and behavior of the above code. In other words, convert the multi-threaded code above into a multi-process version by using `fork()` and `waitpid()` instead of `pthread_create()` and `pthread_join()`.

SOLUTION: Unlike Question 17, in this question, the child threads change the variable `x` back in `main()` via the local pointers `q`.

Still, the first child thread is guaranteed to display its output and terminate before the main thread displays its output. Interleaving and output occurs as shown below. (Try adding `usleep(5)` calls after the `pthread_create()` calls and after the `*q += 6` statement.)

```
<first-child-and-main-thread>          <second-child-thread>
512 lucky <v1>                  <====> 513 lucky <v3>
0 unlucky <v2>                  <====>
```

Here, the final value `<v2>` in `main()` could be 13, 19, or 25 (depending on when the child threads add 6 to `x` in relation to the `x=13` assignment statement in `main()`).

Similarly, the values of `<v1>` and `<v3>` could then be 13, 19, or 25. (**(v1.1)** Corrected typo in this line.)

Note that we see the above output possibilities because the following code is interleaved:

```
x = 7;
pthread_create() calls...
  |                        |                        |
<main>                 <first-child-thread>     <second-child-thread>
x = 13;                *q = arg; *q += 6;       *q = arg; *q += 6;
pthread_join(); <----- printf( ... );           printf( ... );
printf( ... );                                      |
pthread_join(); <-----------------------------------
```

Critical sections are shown below in relation to variable `x` in `main()`, which is shared via pointer `q` in the two child threads. Further, a `mutex` variable is added to synchronize the three threads.

Note that at minimum, a thread obtains the `mutex` lock when it wants to change the shared variable `x`. We do not include read access to `x` within our critical sections unless the problem requires this behavior (i.e., a specific sequence of multi-threaded instructions is required). Further, we assume that the assembly-level update of `x` consists of one indivisible assembly instruction.

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void * qwerty( void * arg )
{
  int * q = (int *)arg;
  pthread_mutex_lock( &mutex );   /*----<start critical section>----*/
  {
    *q += 6;
  }
  pthread_mutex_unlock( &mutex ); /*-----<end critical section>-----*/
  printf( "%ld lucky %d\n", pthread_self(), *q );
  return NULL;
}

int main()
{
  pthread_t tid1, tid2;
  int x = 7;
  int rc = pthread_create( &tid1, NULL, qwerty, &x );
  rc = pthread_create( &tid2, NULL, qwerty, &x );
  pthread_mutex_lock( &mutex );   /*----<start critical section>----*/
  {
    x = 13;
  }
  pthread_mutex_unlock( &mutex ); /*-----<end critical section>-----*/
  rc = pthread_join( tid1, NULL );
  printf( "%d unlucky %d\n", rc, x );
  rc = pthread_join( tid2, NULL );
  return EXIT_SUCCESS;
}
```

19. Given the following C program, what is the **exact** terminal output? Further, what is the **exact** contents of the `E.txt` file? If multiple outputs are possible, succinctly describe all possibilities. Assume that all system calls complete successfully. Also assume that the main thread ID is `256`, while child thread IDs are assigned sequentially starting at `512`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define SONNY 0

void * wtf( void * arg )
{
  int * f = (int *)arg;
  printf( "%ldA%d\n", pthread_self(), *f );
  fprintf( stderr, "%ldB\n", pthread_self() );
  return NULL;
}

int main()
{
  close( SONNY );
  printf( "%ldC\n", pthread_self() );
  fprintf( stderr, "%ldD\n", pthread_self() );

  int fd = open( "E.txt", O_WRONLY | O_CREAT | O_TRUNC, 0660 );
  printf( "%ldF\n", pthread_self() );
  fprintf( stderr, "%ldG%d\n", pthread_self(), fd );

  pthread_t tid1, tid2;
  int rc = pthread_create( &tid1, NULL, wtf, &fd );
  rc = pthread_create( &tid2, NULL, wtf, &rc );

  pthread_join( tid1, NULL );
  pthread_join( tid2, NULL );
  printf( "%ldH\n", pthread_self() );
  fprintf( stderr, "%ldI\n", pthread_self() );

  fflush( NULL );
  close( fd );
  return EXIT_SUCCESS;
}
```

How do the output and file contents change if `SONNY` is defined as `1` instead of `0`?