CS 3214 Fall 2020 Test 2 Solutions

November 11, 2020

Contents

Rules

- This exam is open book, open notes, and open Internet, but in a read-only way.
- You are not allowed to post or otherwise communicate with anyone else about these problems.
- You are required to cite any sources you use, except for lecture material, source code provided as part of the class materials, and the textbook.
- If you have a question about the exam, you may post it as a *private* question on Piazza. If it is of interest to others, I will make it public.
- During the test period, you may not use Piazza to post other questions or reply to questions posted there.
- Any errata to this exam will be published prior to 12h before the deadline.

1 Pass on Threads? (12 pts)

Consider the following program:

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

#define LIST_SIZE 5

struct bagOStuff {
  atomic_int threadID;
  long * shouldBeHidden;
  atomic_int * listOStuff;
};

void * thread_start1(void * arg) {
  struct bagOStuff * myBag = (struct bagOStuff *)arg;

  for(int i = 0; i < LIST_SIZE; ++i) {
    myBag->listOStuff[i] = myBag->threadID;
  }

  printf("Thread: %u\n", myBag->threadID);

  return NULL;
}

void * thread_start2(void * arg) {
  struct bagOStuff * myBag = (struct bagOStuff *)arg;
  printf("Thread: %u\n", myBag->threadID);
  printf("Thread's hidden: %lu\n", *myBag->shouldBeHidden);

  for(int i = 0; i < LIST_SIZE; ++i) {
    printf("%u ", myBag->listOStuff[i]);
  }
  printf("\n");

  *myBag->shouldBeHidden = 9035768;

  return NULL;
}

```

```
43 int main(void) {
44   long hidden = 8675309;
45   pthread_t thread1, thread2;
46   atomic_int * intList = (atomic_int *)calloc(LIST_SIZE, sizeof(atomic_int));
47   struct bagOStuff toPass = {1, &hidden, intList};
48
49   pthread_create(&thread1, NULL, thread_start1, (void *)(&toPass));
50
51   toPass.threadID = 2;
52
53   pthread_create(&thread2, NULL, thread_start2, (void *)(&toPass));
54
55   pthread_join(thread1, NULL);
56   pthread_join(thread2, NULL);
57
58   printf("Main's hidden: %lu\n", hidden);
59 }
```

Using the atomic data types provided by `stdatomic.h` ensures that even in the presence of data races, the resulting operations on the protected variable are sequentially consistent (i.e., behaves how programmers would expect). While the atomics library ensures that the low-level variable accesses are well-behaved, thread inter-leavings may still impact the result of execution (i.e., sequential consistency is not the same as having one possible total ordering of variable accesses). Keep that in mind while answering the following questions:

1. (2 pts) What is/are the possible ID values printed by the first thread created in `main()`? Why? Saying "Because that's what happened when I ran it" is insufficient.

   [Solution] The possible ID values will be 1 or 2, depending on the ordering of main() preparing the struct for thread 2 and thread 1's execution.

2. (2 pts) What is/are the possible ID values printed by the second thread created in `main()`? Why? Saying "Because that's what happened when I ran it" is insufficient.

   [Solution] The only possible ID printed is 2, since the change of ID happens before thread 2 is created. Thus the ID value update always happens-before the ID print by thread 2.

3. (2 pts) Both threads take an argument `arg` that they store in the variable `myBag`. Assume that instead of a `struct` the parameter was a scalar (e.g., `int`). See line 104 of Problem 3 for an example of how to do this with threads. Does such a variable need to be protected from data races across threads? Why or why not?

   [Solution] Threads have their own stack and registers and scalars, such as `int`s, are passed by value. When that value is assigned to a thread's local variable (`myBag` in this case), it resides in that thread's stack or registers. Thus, all accesses are private and not subject to data races.

4. (2 pts) What does the thread see as the value of the variable `shouldBeHidden` on line 31 and why (especially with respect to the previous question/answer)?

[Solution] 8675309 Because `main()`'s creation of thread 2 happens-before the printing by thread 2. Even though you threads have their own stack (i.e., the previous answer) and `hidden` is allocated on `main()`'s stack, threads share the same, single, virtual address space. Thus, if you have a variable's address, you can access that variable.

5. (2 pts) How many possible list strings does the second thread created print out (e.g., "0 0 0 0 0", "0 0 0 0 1", and "1 0 0 0 0" count as three different list strings)?

   [Solution] There is a happens-before relationship that creates two distinct thread scheduling races that we must account for (I use the term scheduling race to indicate when the results of execution depend on thread scheduling. When this is the case, a program has multiple sequentially-consistent outcomes.): by the time thread 2 starts printing, `main()` updated the thread ID in the struct passed to the threads during creation, which thread 1 uses for list updates (as you know from the first question). This means that thread 1 will set the list to some, potentially zero, number of `1`'s, followed by enough `2`'s to fill the five-item list. This results in six possible prefixes. For each of these prefixes, the second thread scheduling race determines whether the value of 0 from `main()` is printed or thread 1's ID (which must now be 2, due to the previously-described happens-before relationship). The key idea behind this scheduling race is that the list value updates by thread 1 can be preempted—at any time—by thread 2's printing. Likewise, thread 2's printing can be preempted—at any time—by thread 1's list updates. This makes each index of the array (as printed by thread 2) the subject of a independent scheduling race between the `0` (as set by `main()`) or thread 1's ID. Thus, for the `N` bits remaining after the leading-1's, there are 2^N possible values. To determine the total number of possible string values, we sum the result for each of the six possible leading-1 prefixes. This results in a total of 63 possible strings. As a "fun" extension to this problem, consider what happens if thread 1 and thread 2 traverse the list from opposite directions.

6. (2 pts) What does `main()` see as the value of its variable `hidden` on line 58 and why?

   [Solution] 9035768, because a happens-before relationship is created by the `pthread_join()`. Again shared virtual address space (i.e., no copy-on-write as seen with multiprocessing).

2 A River Crossing Puzzle (12 pts)

The wolf, goat, and cabbage problem is a river crossing puzzle that, according to Wikipedia [URL], dates back centuries.

> Once upon a time a farmer went to a market and purchased a wolf, a goat, and a cabbage. On his way home, the farmer came to the bank of a river and rented a boat. But crossing the river by boat, the farmer could carry only himself and a single one of his purchases: the wolf, the goat, or the cabbage.
>
> If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage.
>
> The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact. How did he do it?

If the man, the wolf, the goat, and the cabbage were threads, a possible solution that denotes their actions could be described by the following program:

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>

// you may define your condition variable(s) here
// if you need mutexes too, define them here
// as well as any other global variables (if needed)
// you may use helper functions.
//
// You may not use semaphores.
//

// you may change the following four functions, but you may not change
// the order of the printf statements, nor the thread that makes those
// statements, and you may not add additional printf statements.
static void *
man(void *_arg)
{
    printf("man enters boat\n");
    printf("man returns\n");
    printf("man returns once more\n");
    return NULL;
}

static void *
wolf(void *_arg)
{
    printf("wolf and man cross left to right\n");
    return NULL;
}

static void *
cabbage(void *_arg)
{
    printf("cabbage and man cross left to right\n");
    return NULL;
}

static void *
```

```c
goat(void *_arg)
{
    printf("goat and man cross left to right\n");
    printf("goat and man cross right to left\n");
    printf("goat arrives on other side for good\n");
    return NULL;
}

int
main(int ac, char *av[])
{
    pthread_t t[4];

    // you may add code here
    pthread_create(&t[0], NULL, man, NULL);
    pthread_create(&t[1], NULL, wolf, NULL);
    pthread_create(&t[2], NULL, cabbage, NULL);
    pthread_create(&t[3], NULL, goat, NULL);

    for (int i = 0; i < 4; i++)
        pthread_join(t[i], NULL);
    // you may add code here
}
```

Unfortunately, due to the unpredictable nature of thread scheduling on a preemptive multicore system, this does not output a solution that would preserve all 3 items.

Your task is to use condition variables (not semaphores!) to ensure that the program always outputs the correct solution, which is:

```
man enters boat
goat and man cross left to right
man returns
cabbage and man cross left to right
goat and man cross right to left
wolf and man cross left to right
man returns once more
goat arrives on other side for good
```

- Your program must not use semaphores or other POSIX constructs besides POSIX condition variables and, if necessary mutexes. You may define global variables if that is needed.
- You cannot use functions such `sleep()` or `nanosleep()` because they do not guarantee that a thread will be scheduled right after being woken up.
- Your program should not contain data races and should not busy wait, which you can test with the usual tools.

[Solution]

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>

/*
 *      Wolf Cabbage Goat Man   --
                --> Goat Man
 *      Wolf Cabbage            -- Goat Man
                <-- Man
 *      Wolf Cabbage Man        -- Goat
                --> Cabbage Man
 *      Wolf                    -- Goat Cabbage Man
                <-- Goat Man
 *      Wolf Goat Man           -- Cabbage
                --> Wolf Man
 *      Goat                    -- Wolf Cabbage Man
                <-- Man
 *      Goat Man                -- Wolf Cabbage
                --> Goat Man
 *                              -- Wolf Cabbage Goat Man
 */
pthread_mutex_t statelock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int state;  // 0, 1, 2, 3, ...

static void signal()
{
    state++;
    pthread_cond_broadcast(&cond);
}

static void waitfor(int wantstate)
{
    while (state != wantstate)
        pthread_cond_wait(&cond, &statelock);
}

static void *
man(void *_arg)
```

```
{
    pthread_mutex_lock(&statelock);
    printf("man enters boat\n");
    signal();
    waitfor(2);
    printf("man returns\n");
    signal();
    waitfor(6);
    printf("man returns once more\n");
    signal();
    waitfor(7);
    pthread_mutex_unlock(&statelock);
    return NULL;
}

static void *
wolf(void *_arg)
{
    pthread_mutex_lock(&statelock);
    waitfor(5);
    printf("wolf and man cross left to right\n");
    signal();
    pthread_mutex_unlock(&statelock);
    return NULL;
}

static void *
cabbage(void *_arg)
{
    pthread_mutex_lock(&statelock);
    waitfor(3);
    printf("cabbage and man cross left to right\n");
    signal();
    pthread_mutex_unlock(&statelock);
    return NULL;
}

static void *
goat(void *_arg)
{
    pthread_mutex_lock(&statelock);
    waitfor(1);
    printf("goat and man cross left to right\n");
    signal();
    waitfor(4);
    printf("goat and man cross right to left\n");
```

```c
    signal();
    waitfor(7);
    printf("goat arrives on other side for good\n");
    pthread_mutex_unlock(&statelock);
    return NULL;
}

int
main(int ac, char *av[])
{
    pthread_t t[4];

    pthread_create(&t[0], NULL, man, NULL);
    pthread_create(&t[1], NULL, wolf, NULL);
    pthread_create(&t[2], NULL, cabbage, NULL);
    pthread_create(&t[3], NULL, goat, NULL);

    for (int i = 0; i < 4; i++)
        pthread_join(t[i], NULL);
}
```

3 Making Random Permutations (15 pts)

The goal of the following program is to output 10 permutations of the string `abcdefghij`, for example:

```
cebdihgjaf
ghfejacibd
djcehigbaf
begafjchdi
ibaegjhdcf
jacfebgihd
ecjaifgdbh
jibegfhcad
idfhjcaegb
ijgahcedfb
```

The program is shown below:

```c
// making random permutations
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
```

```c
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>

#define NTHREADS 10

static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
static sem_t permutation_done;
static sem_t permutation_extra;     // not currently used

// you may change these 2 lines below if necessary
const int PERMUTATION_DONE_INITIAL_VALUE = 0;
const int PERMUTATION_EXTRA_INITIAL_VALUE = 0;

static bool use_slow_threads;
typedef void (*func_t)(void);

/*
 * This function is called by exactly n threads, repeatedly, as
 * shown below in thread_fun.
 * Each time it's called by these threads it should output a
 * permutation of the string 'abcdefghij', following by a newline.
 * Thread 1 must output 'a', thread 2 must output 'b', and so on
 * Thread i must output 'a' + i.
 *
 * Exactly one thread should invoke the `when_permutation_finished`
 * function when the permutation has been output and before the
 * output of the next permutation has been started.
 */
static void
make_a_permutation(int n, int id, func_t when_permutation_finished)
{
    assert (0 <= id && id < n);
    // you may not change this line
    printf("%c", 'a' + id);

    // you may make changes below in the remainder of this function
    static int permutation_index = 0;

    pthread_mutex_lock(&lock);
    int pi = ++permutation_index;
    pthread_mutex_unlock(&lock);

    if (pi == n) {
        when_permutation_finished();
        sem_post(&permutation_done);
    }

    sem_wait(&permutation_done);
    sem_post(&permutation_done);

    pthread_mutex_lock(&lock);
    pi = --permutation_index;
    pthread_mutex_unlock(&lock);

    if (pi == 0)
        sem_wait(&permutation_done);
    // you may not make changes below this line
}

static void
newline(void)
{
    printf("\n");
    fflush(stdout);
}

static void *
thread_fun(void *_arg)
{
    int id = (uintptr_t) _arg;
    for (int i = 0; i < 10; i++) {
        if (use_slow_threads) {
            // slow threads sleep a random amount of time
            // drawn from [.5ms, 1.5ms]
            struct timespec ts = {
                .tv_sec = 0,
                .tv_nsec = 500000 + rand() % 1000000
            };
            nanosleep(&ts, 0);
        }
        make_a_permutation(NTHREADS, id, newline);
    }
    return NULL;
}

int
main(int ac, char *av[])
{
    pthread_t threads[NTHREADS];
    sem_init(&permutation_done, 0, PERMUTATION_DONE_INITIAL_VALUE);
```

```
 99     sem_init(&permutation_extra, 0, PERMUTATION_EXTRA_INITIAL_VALUE);
100
101     use_slow_threads = !(ac > 1 && !strcmp(av[1], "-f"));
102
103     for (int i = 0; i < NTHREADS; i++)
104         pthread_create(threads + i, NULL, thread_fun, (void *) (uintptr_t) i);
105
106     for (int i = 0; i < NTHREADS; i++)
107         pthread_join(threads[i], NULL);
108 }
```

Consider this program, and answer the following questions.

1. (3 pts) If you compiled and ran this program on a multiprocessor machine such as our rlogin cluster without specifying any command line switches, it would appear to "work" probably most of the time and indeed output 10 permutations. Analyze the program and briefly explain why this is the case.

   [Solution] Under benign conditions (imagine the threads running very slowly, or one at a time), the first $n-1$ threads to arrive in the `make_a_permutation` function will wait on line 56 until the last thread arrives, which then calls `when_permutation_finished` on line 52, then posts to the semaphore. As a result, one thread will make it past line 56. This thread will post this semaphore, allowing another thread to make it past line 56. The semaphore is thus signaled $n+1$ times overall; one of the threads (the thread that sees that `pi` has reached 0) will wait on line 64 to consume the extra signal. Thus under benign conditions, each thread waits on a "turnstile" semaphore that will not be signaled until the last thread arrives. This ensures that `when_permutation_finished` is not called until all threads have arrived.

2. (3 pts) Unfortunately, the program ran under conditions that have a tendency to hide concurrency related bugs in it – simulated here by letting each thread sleep for a small amount of time in its main loop (`use_slow_threads` is true by default). The `nanosleep` function moves a thread (and only this thread) into the `BLOCKED` state for some amount of (wallclock) time. When you run the program instead with the `-f` switch so that each thread will no longer have this sleep period, the program suddenly starts to fail.

   Analyze the program for data races. Data races are defined in lecture; or more formally in the C11 memory model [URL] as follows:

   > When an evaluation of an expression writes to a memory location and another evaluation reads or modifies the same memory location, the expressions are said to conflict. A program that has two conflicting evaluations has a data race unless either
   >
   > - both conflicting evaluations are atomic operations
   > - one of the conflicting evaluations *happens-before*[1] another

   Which data races does this program contain, if any? Justify your answer, referring to line numbers in the code if this is necessary.

[1]Here, *happens-before* refers to the cross-thread relationship between events, see Slide 11.

[Solution] Interestingly, this program does not contain data races because all accesses to shared variables are protected with a lock.

3. (3 pts) Discuss one way in which program fails and explain why it fails. If the failure is due to a data race, explain how the data race contributed to the failure. If the failure is due to another kind of concurrency related problem, describe the problem and how it contributed to the failure.

   [Solution] Although the program does not contain data races, it contains other concurrency bugs. Here, there is an ordering violation: the programmer assumed that no thread will invoke `make_a_permutation` again until all threads have left this function (and the `permutation_index` variable has reached 0). If a thread that has returned from this function calls it again before this variable reaches zero, then this thread will increment it again, and the variable may reach n again even though not all threads have had a chance to output their character for the second permutation. Conversely, it is possible for threads to leave this function after a thread has looped around, decrementing the variable, which in turn prevents any thread from executing the if-branch on line 51, leading to a deadlock of all threads being blocked on line 56. You can observe this effect when running the program under gdb. Naming one of these 2 scenarios is sufficient.

4. (6 pts) Now fix the program, subject to the following constraints:

   - You may make changes only to the marked areas in the `make_a_permutation` function, but you may also change (if necessary) the initial values of the 2 semaphores defined in the program. The section of code to which you may make changes are marked. Only 1 semaphore is currently used, you will need to use the second (marked "extra") semaphore to make the program work.
   - You may not introduce additional mutexes, semaphores, or condition variables, or use other POSIX APIs not related to mutexes and semaphores.
   - A correct program should output 10 permutations, separated by a newline, not only when run under `-f`, but under any permissible scheduling of the 10 threads.

   [Solution]

   It is possible to use the second semaphore in a similar manner as the first one to ensure that no thread leaves this function until `permutation_index` has reached zero. This "extra" semaphore acts as a second turnstile. It is signaled when pi reaches 0, then each waiting thread executes a wait/post sequence, letting out thread after thread. Since there will be an extra signal when the last thread leaves the function, this extra signal must be consumed for the next round; this can be done by the last thread to arrive (the thread that starts the process of letting all others threads through the first turnstile. To ensure this works the first time around, this semaphore can be initialized to 1.

```
1  // making random permutations
2  #include <pthread.h>
3  #include <semaphore.h>
4  #include <stdlib.h>
5  #include <stdio.h>
```

```c
#include <assert.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>

#define NTHREADS 10

static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
static sem_t permutation_done;
static sem_t permutation_extra;     // not currently used

// you may change these 2 lines below if necessary
const int PERMUTATION_DONE_INITIAL_VALUE = 0;
const int PERMUTATION_EXTRA_INITIAL_VALUE = 1;

static bool use_slow_threads;
typedef void (*func_t)(void);

/*
 * This function is called by exactly n threads, repeatedly, as
 * shown below in thread_fun.
 * Each time it's called by these threads it should output a
 * permutation of the string 'abcdefghij', following by a newline.
 * Thread 1 must output 'a', thread 2 must output 'b', and so on
 * Thread i must output 'a' + i.
 *
 * Exactly one thread should invoke the `when_permutation_finished`
 * function when the permutation has been output and before the
 * output of the next permutation has been started.
 */
static void
make_a_permutation(int n, int id, func_t when_permutation_finished)
{
    assert (0 <= id && id < n);
    // you may not change this line
    printf("%c", 'a' + id);

    // you may make changes below in the remainder of this function
    static int permutation_index = 0;

    pthread_mutex_lock(&lock);
    int pi = ++permutation_index;
    pthread_mutex_unlock(&lock);

    if (pi == n) {
        // last thread to arrive
        when_permutation_finished();
        sem_wait(&permutation_extra); // consume extra signal
        sem_post(&permutation_done); // let in next thread
    }

    sem_wait(&permutation_done); // wait for first signal
    sem_post(&permutation_done); // let in next thread

    pthread_mutex_lock(&lock);
    pi = --permutation_index;
    pthread_mutex_unlock(&lock);

    if (pi == 0) {
        // last thread to leave
        sem_wait(&permutation_done);  // consume extra signal
        sem_post(&permutation_extra); // let out first thread
    }

    sem_wait(&permutation_extra); // wait to leave function
    sem_post(&permutation_extra); // let out next thread
    // you may not make changes below this line
}

static void
newline(void)
{
    printf("\n");
    fflush(stdout);
}

static void *
thread_fun(void *_arg)
{
    int id = (uintptr_t) _arg;
    for (int i = 0; i < 10; i++) {
        if (use_slow_threads) {
            // slow threads sleep a random amount of time
            // drawn from [.5ms, 1.5ms]
            struct timespec ts = {
                .tv_sec = 0,
                .tv_nsec = 500000 + rand() % 1000000
            };
            nanosleep(&ts, 0);
        }
        make_a_permutation(NTHREADS, id, newline);
```

```c
    }
    return NULL;
}

int
main(int ac, char *av[])
{
    pthread_t threads[NTHREADS];
    sem_init(&permutation_done, 0, PERMUTATION_DONE_INITIAL_VALUE);
    sem_init(&permutation_extra, 0, PERMUTATION_EXTRA_INITIAL_VALUE);

    use_slow_threads = !(ac > 1 && !strcmp(av[1], "-f"));

    for (int i = 0; i < NTHREADS; i++)
        pthread_create(threads + i, NULL, thread_fun, (void *) (uintptr_t) i);

    for (int i = 0; i < NTHREADS; i++)
        pthread_join(threads[i], NULL);
}
```

4 A Deadlock Mystery (8 pts)

When Dr. Back tried to come up with a question for this exam, he experimented with a "mystery" program that he hoped would deadlock at some point. Eventually, he caught the program deadlocking in gdb. You can see the resulting debugging session below, with linebreaks inserted for readability.

```
GNU gdb (Ubuntu 8.2-0ubuntu1~18.04) 8.2
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./mystery...done.
(gdb) run
Starting program: /home/gback/cs3214/exams/mystery
[Thread debugging using libthread_db enabled]
```

```
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff73b9700 (LWP 6087)]
[New Thread 0x7ffff6bb8700 (LWP 6088)]
[New Thread 0x7ffff63b7700 (LWP 6089)]
[New Thread 0x7ffff5bb6700 (LWP 6090)]
^C
Thread 1 "mystery" received signal SIGINT, Interrupt.
0x00007ffff79b7d2d in __GI___pthread_timedjoin_ex (threadid=140737341265664,
       thread_return=0x0, abstime=0x0, block=<optimized out>)
    at pthread_join_common.c:89
89        pthread_join_common.c: No such file or directory.
(gdb) thread apply all backtrace

Thread 5 (Thread 0x7ffff5bb6700 (LWP 6090)):
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1  0x00007ffff79b9023 in __GI___pthread_mutex_lock (mutex=0x555555755080 <NW>)
                                                     at ../nptl/pthread_mutex_lock.c:78
#2  0x0000555555554973 in car (_info=0x3) at mystery.c:47
#3  0x00007ffff79b66db in start_thread (arg=0x7ffff5bb6700) at pthread_create.c:463
#4  0x00007ffff76dfa3f in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95

Thread 4 (Thread 0x7ffff63b7700 (LWP 6089)):
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1  0x00007ffff79b9023 in __GI___pthread_mutex_lock (mutex=0x5555557550c0 <SE>)
                                                     at ../nptl/pthread_mutex_lock.c:78
#2  0x0000555555554930 in car (_info=0x2) at mystery.c:40
#3  0x00007ffff79b66db in start_thread (arg=0x7ffff63b7700) at pthread_create.c:463
#4  0x00007ffff76dfa3f in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95

Thread 3 (Thread 0x7ffff6bb8700 (LWP 6088)):
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1  0x00007ffff79b9023 in __GI___pthread_mutex_lock (mutex=0x555555755100 <SW>)
                                                     at ../nptl/pthread_mutex_lock.c:78
#2  0x00005555555548ea in car (_info=0x1) at mystery.c:33
#3  0x00007ffff79b66db in start_thread (arg=0x7ffff6bb8700) at pthread_create.c:463
#4  0x00007ffff76dfa3f in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95

Thread 2 (Thread 0x7ffff73b9700 (LWP 6087)):
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1  0x00007ffff79b9023 in __GI___pthread_mutex_lock (mutex=0x555555755040 <NE>)
                                                     at ../nptl/pthread_mutex_lock.c:78
#2  0x00005555555548a4 in car (_info=0x0) at mystery.c:26
#3  0x00007ffff79b66db in start_thread (arg=0x7ffff73b9700) at pthread_create.c:463
#4  0x00007ffff76dfa3f in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95

Thread 1 (Thread 0x7ffff7fbfb80 (LWP 6082)):
#0  0x00007ffff79b7d2d in __GI___pthread_timedjoin_ex (threadid=140737341265664,
                                  thread_return=0x0, abstime=0x0, block=<optimized out>)
    at pthread_join_common.c:89
#1  0x0000555555554a7c in main () at mystery.c:66
```

```
(gdb) info threads
  Id   Target Id                                  Frame
* 1    Thread 0x7ffff7fbfb80 (LWP 6082) "mystery" 0x00007ffff79b7d2d
            in __GI___pthread_timedjoin_ex (threadid=140737341265664, thread_return=0x0,
                            abstime=0x0, block=<optimized out>) at pthread_join_common.c:89
  2    Thread 0x7ffff73b9700 (LWP 6087) "mystery" __lll_lock_wait ()
                                        at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
  3    Thread 0x7ffff6bb8700 (LWP 6088) "mystery" __lll_lock_wait ()
                                        at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
  4    Thread 0x7ffff63b7700 (LWP 6089) "mystery" __lll_lock_wait ()
                                        at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
  5    Thread 0x7ffff5bb6700 (LWP 6090) "mystery" __lll_lock_wait ()
                                        at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
(gdb) p NE
$1 = pthread_mutex_t = {Type = Normal, Status = Acquired, possibly with waiters,
                        Owner ID = 6090, Robust = No, Shared = No, Protocol = None}
(gdb) p NW
$2 = pthread_mutex_t = {Type = Normal, Status = Acquired, possibly with waiters,
                        Owner ID = 6088, Robust = No, Shared = No, Protocol = None}
(gdb) p SW
$3 = pthread_mutex_t = {Type = Normal, Status = Acquired, possibly with waiters,
                        Owner ID = 6089, Robust = No, Shared = No, Protocol = None}
(gdb) p SE
$4 = pthread_mutex_t = {Type = Normal, Status = Acquired, possibly with waiters,
                        Owner ID = 6087, Robust = No, Shared = No, Protocol = None}
```

They say a picture says a thousand words, so to make your job easier, we ask that you draw a sketch that illustrates the cause of this deadlock.

[Solution]

The sketch we were looking for is the classic illustration of gridlock in an intersection, shown in Figure 1.

4 cars (threads) are in the blocked state. For instance, thread 6090 holds lock NE ("North East") and is blocked in an attempt to acquire lock NW ("North West"), similarly for the others. The program used to produce this gdb output is shown below:

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

enum Direction { NORTH, SOUTH, EAST, WEST };
pthread_mutex_t NE = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t NW = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t SE = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t SW = PTHREAD_MUTEX_INITIALIZER;

static void *
```

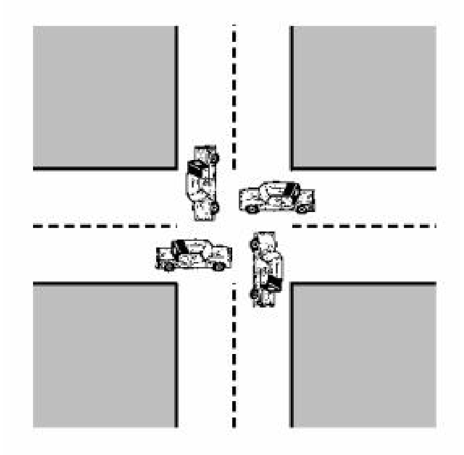

Figure 1: Source: Stallings, *Operating Systems: Internals and Design Principles*

```c
car(void * _info)
{
    struct timespec req = {
        .tv_sec = 0,
        .tv_nsec = 1000
    };
    enum Direction dir = (enum Direction) _info;
    for (int i = 0; i < 100000; i++) {
        switch (dir) {
        case NORTH:
            pthread_mutex_lock(&SE);
            pthread_mutex_lock(&NE);
            nanosleep(&req, NULL);
            pthread_mutex_unlock(&NE);
            pthread_mutex_unlock(&SE);
            break;
        case SOUTH:
            pthread_mutex_lock(&NW);
            pthread_mutex_lock(&SW);
            nanosleep(&req, NULL);
            pthread_mutex_unlock(&SW);
            pthread_mutex_unlock(&NW);
            break;
        case EAST:
            pthread_mutex_lock(&SW);
            pthread_mutex_lock(&SE);
            nanosleep(&req, NULL);
            pthread_mutex_unlock(&SE);
            pthread_mutex_unlock(&SW);
            break;
        case WEST:
            pthread_mutex_lock(&NE);
            pthread_mutex_lock(&NW);
            nanosleep(&req, NULL);
            pthread_mutex_unlock(&NW);
            pthread_mutex_unlock(&NE);
            break;
        }
    }
}

int
main()
{
    pthread_t t[4];
    pthread_create(&t[0], NULL, car, (void *) NORTH);
    pthread_create(&t[1], NULL, car, (void *) SOUTH);
    pthread_create(&t[2], NULL, car, (void *) EAST);
    pthread_create(&t[3], NULL, car, (void *) WEST);

    for (int i = 0; i < 4; i++)
        pthread_join(t[i], NULL);
}
```

5 Optimizing Parallel Performance (8 pts)

Using a visualization tool, you have obtained a timeline of the execution of a multi-threaded program you are currently optimizing, which can be seen Figure 2. Time moves from the top to the bottom, and different colors are used to denote phases of execution, as described in the caption.

Answer the following questions:

1. (2 pts) Estimate the overall CPU utilization for this workload in percent, where 100% would be full utilization of all 6 cores.

   [Solution] The CPU utilization here is 82%, corresponding to the ratio of non-white areas to the area of the entire rectangle.

2. (2 pts) Identify and describe the effect that most affects CPU utilization for this example.

   [Solution] This workload shows a so-called "convoy effect" - one thread (thread 5) holds a lock (the black lock) for long periods of time (relative to the time others are holding this lock), causing a convoy of other threads that also wish to acquire this lock to form behind it. When those threads try to acquire this lock they are blocked and the corresponding CPU cannot be used.

3. (2 pts) Which thread and/or lock should be optimized to improve performance, and how?

   [Solution] A main optimization goal should be to reduce the time thread 5 holds the black lock (or alternatively, find a way that avoids that threads 2, 3, 4 also compete for this lock).

4. (2 pts) Could this program have deadlocked under a different scheduler?

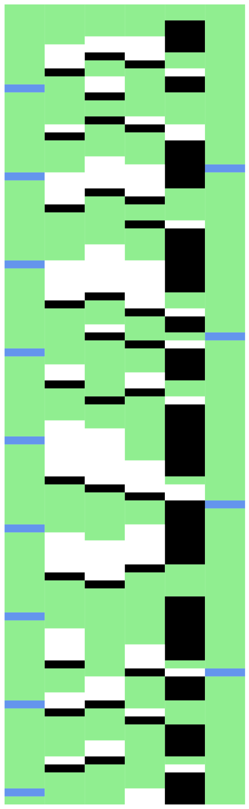

Figure 2: Timeline showing the execution of 6 threads on 6 CPU cores. Each vertical stripe (or column) represents one thread. Time that is spent executing while holding no lock is shown in green. Time during which the CPU is idle is shown in white. Time spent holding a lock is shown in a color corresponding to the lock, either black or blue. None of the threads engage in I/O or move into the BLOCKED state for any reason other than waiting for a lock.

> **[Solution]** From the information shown, no thread ever holds 2 locks, removing at least one of the necessary conditions for deadlock. Thus, deadlock cannot occur.

Briefly justify each answer.

6 Submission Requirements

Submit a tar file that contains the following files:

- `random_permutation.c` to answer Question 3 part 4. This program should compile.
- `mystery.pdf` to answer Question 4.
- `rivercrossing.c` to answer Question 2. This program should also compile.

- `answers.txt` with answers to Question 1, Questions 3 (parts 1 to 3), and Question 5.