CS 3214 Summer 2020 Test 2 Solutions

July 21, 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.
- Any errata to this exam will be published prior to 12h before the deadline.

1 Managing Execution Order (12 pts)

Consider the following incomplete program:

```c
/* Complete the program below so that:
 *
 * - Threads A through E, and the main thread, each call
 *   function `output` zero or one times.
 *
 * - Possible outputs of the program are exactly these:
 *
 *      ACF
 *      ADF
 *      AEF
 *      BCF
 *      BDF
 *      BEF
 *
 * - The program does not contain data races
 * - All threads have finished by the time main returns
 * - The program exits upon return from main and not earlier
 *
 * You may:
 * - add any statements, variables, and functions
 * - use POSIX semaphores, mutexes, and/or C11 atomics
 *
 * You may not:
 * - use POSIX condition variables
 * - remove any of the given code or prevent it from running.
 */
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>

_Thread_local char *tag;

static void output() {
    printf("%s", tag);
}

static void *
thread_A(void *_data)
```

```
{
    tag = _data;
    return NULL;
}

static void *
thread_B(void *_data)
{
    tag = _data;
    return NULL;
}

static void *
thread_C(void *_data)
{
    tag = _data;
    return NULL;
}

static void *
thread_D(void *_data)
{
    tag = _data;
    return NULL;
}

static void *
thread_E(void *_data)
{
    tag = _data;
    return NULL;
}

int
main()
{
    /* Threads under a preemptive scheduling regime execute
     * non-deterministically.  To amplify this effect, we
     * start them in random order.
     *
     * However, the set of possible outputs of your program
     * must not depend on the order in which the threads are
     * started.
     */
    void * (*thread_func[])(void *) = { thread_A, thread_B,
                                        thread_C, thread_D,
```

```
                                        thread_E };
    char *tags[] = { "A", "B", "C", "D", "E" };
    tag = "F\n";
    const int N = sizeof(thread_func)/sizeof(thread_func[0]);
    srand(time(NULL));
    pthread_t threads[N];
    int started = 0;
    while (started != (1<<N)-1) {
        int next = rand() % N;
        if (started & (1<<next)) continue;
        started |= (1<<next);
        pthread_create(threads + next, NULL,
                       thread_func[next], tags[next]);
    }
}
```

Complete the program so that the resulting multi-threaded program outputs either ACF, ADF, AEF, BCF, BDF, or BEF, subject to the stipulations given in the comment at the beginning of the program.

You should use mutexes and semaphores, do not use condition variables. Please also take note of the stipulation that the main thread must exit last.

Solution: This problem could be solved with a semaphore to ensure the $(A|B) \rightarrow (C|D|E)$ constraint, a lock and a flag to ensure that either A or B prints (and the same for C, D, E. The $(C|D|E) \rightarrow F$ constraint could be accomplished by joining the spawned threads. Care must be taken that all threads exit.

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

_Thread_local char *tag;
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
sem_t ab_then_cde;
int ab_done;
int cde_done;

static void output() {
    printf("%s", tag);
}
```

```c
static void tryoutput(int *counter, pthread_mutex_t *lock)
{
    pthread_mutex_lock(lock);
    if (*counter == 0) {
        output();
        (*counter)++;
    }
    pthread_mutex_unlock(lock);
}

static void *
thread_A(void *_data)
{
    tag = _data;
    tryoutput(&ab_done, &lock1);
    for (int i = 0; i < 3; i++)
        sem_post(&ab_then_cde);
    return NULL;
}

static void *
thread_B(void *_data)
{
    tag = _data;
    tryoutput(&ab_done, &lock1);
    for (int i = 0; i < 3; i++)
        sem_post(&ab_then_cde);
    return NULL;
}

static void *
thread_C(void *_data)
{
    tag = _data;
    sem_wait(&ab_then_cde);
    tryoutput(&cde_done, &lock2);
    return NULL;
}

static void *
thread_D(void *_data)
{
    tag = _data;
    sem_wait(&ab_then_cde);
    tryoutput(&cde_done, &lock2);
```

```
    return NULL;
}

static void *
thread_E(void *_data)
{
    tag = _data;
    sem_wait(&ab_then_cde);
    tryoutput(&cde_done, &lock2);
    return NULL;
}

int
main()
{
    sem_init(&ab_then_cde, 0, 0);

    /* Threads under a preemptive scheduling regime execute
     * non-deterministically.  To amplify this effect, we
     * start them in random order.
     *
     * However, the set of possible outputs of your program
     * must not depend on the order in which the threads are
     * started.
     */
    void * (*thread_func[])(void *) = { thread_A, thread_B,
                                        thread_C, thread_D,
                                        thread_E };
    char *tags[] = { "A", "B", "C", "D", "E" };
    tag = "F\n";
    const int N = sizeof(thread_func)/sizeof(thread_func[0]);
    srand(time(NULL));
    pthread_t threads[N];
    int started = 0;
    while (started != (1<<N)-1) {
        int next = rand() % N;
        if (started & (1<<next)) continue;
        started |= (1<<next);
        pthread_create(threads + next, NULL,
                       thread_func[next], tags[next]);
    }

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

2 Locking Code Review (12 pts)

In this question, you are given examples of uses of the POSIX Threads Mutex API taken from excerpts of multi-threaded code that may bear some resemblance to projects you have done in this class. The examples are separate and not connected to each other. You should assume that any code that is elided is correct - base your answers on the code shown here.

Your task is to discuss these uses as you might in a critical code review. Explain if the use is necessary and correct, and explain the purpose of the use if so. Otherwise, explain why it is not correct and suggest ways of correcting the code. Justify your statements.

2.1 Creating a new future (4 pts)

Discuss the calls on lines 6 and 9.

```
1  struct future *
2  threadpool_submit(fork_join_task_t task, void *data)
3  {
4      struct future * newFut = malloc(sizeof(*newFut));
5      pthread_mutex_init(&newFut->lock, NULL);
6      pthread_mutex_lock(&newFut->lock);
7      newFut->task = task;
8      newFut->data = data;
9      pthread_mutex_unlock(&newFut->lock);
10     return newFut;
11 }
```

Solution: These calls are redundant and should be removed. The newly created future is not (yet) shared been any threads, so taking this lock during initialization has no effect.

2.2 Worker Threads, Take 1 (4 pts)

Discuss the calls on lines 10 and 16. Assume that `N >= nthreads`.

```
1  static struct list workerqueue[N];
2
3  static void *
4  worker(void *_data)
5  {
6      intptr_t i = (intptr_t)_data;
7
8      pthread_mutex_t worker_lock = PTHREAD_MUTEX_INITIALIZER;
9
10     pthread_mutex_lock(&worker_lock);
11     list_init(workerqueue+i);
12
```

```
13      //
14      // worker loop is here
15      //
16      pthread_mutex_unlock(&worker_lock);
17  }
18
19  void
20  threadpool_new(int nthreads)
21  {
22      for (intptr_t i = 0; i < nthreads; i++) {
23          pthread_create(workers + i, NULL,
24                         worker, (void *)i);
25      }
26  }
```

Solution: These calls acquire and release a lock that is defined as a local variable. Every worker thread here would have its own lock, not shared with anyone else. Such locks are ineffective - they are seen by only this thread, thus do not provide useful mutual exclusion. Remedy is to move those locks into a place that is safely accessible to all threads, such as the thread pool struct or a location linked from it.

(Note that you could assume that a pointer to this lock is somehow passed to other threads; however, this is very difficult to do since the lock will be deallocated should this thread exit.)

2.3 Worker Threads, Take 2 (4 pts)

Discuss the calls on lines 8, 11, 17, and 24. Assume that `N >= nthreads > 1`.

```
1   pthread_mutex_t lock;
2   pthread_t workers[N];
3   struct list workerqueue[N];
4
5   static void *
6   worker(void *_data)
7   {
8       pthread_mutex_lock(&lock);
9       // worker loop - threads may examine workerqueues in a work stealing
10      // regime to check for tasks to be stolen
11      pthread_mutex_unlock(&lock);
12  }
13
14  void
15  threadpool_new(int nthreads)
16  {
17      pthread_mutex_lock(&lock);
18      for (int i = 0; i < nthreads; i++) {
```

```
19          list_init(workerqueue+i);
20          pthread_create(workers + i, NULL,
21                         worker, NULL);
22      }
23
24      pthread_mutex_unlock(&lock);
25  }
```

Solution: Here, the lock is used appropriately. Holding the lock during thread creation ensures that no worker thread will enter their worker loop unless all worker queues have been initialized. (Therefore, the stealing code will not encounter a situation where it is accessing another worker's list and that list has not yet been initialized.)

3 Simplified Thread Pool (10 pts)

In project 2, you implemented a fork-join thread pool that can execute tasks using a pool consisting of a fixed number of n threads. Consider a simplified version of project 2 in which the `future_get()` and `future_free()` functionality is removed. As before, tasks are submitted to the thread pool. If a task wishes to know if a subtask it has spawned has finished, it would be the programmer's responsibility to add this functionality (this is consistent with the idea of keeping APIs small.) If the task returned a result, the programmer will need to dynamically allocate memory to hold said result. To avoid memory leaks, the pool will free the internal data structure associated with each submitted task as soon as the task is complete.

3.1 Client Changes (5 pts)

Consider this excerpt of the mergesort test you used in project 2:

```
1   /* msort_task describes a unit of parallel work */
2   struct msort_task {
3       int *array;
4       int *tmp;
5       int left, right;
6   };
7
8   /* Parallel mergesort */
9   static void
10  mergesort_internal_parallel(struct thread_pool * threadpool, struct msort_task * s)
11  {
12      int * array = s->array;
13      int * tmp = s->tmp;
14      int left = s->left;
15      int right = s->right;
16
17      if (right - left <= min_task_size) {
```

```
18          mergesort_internal(array, tmp + left, left, right);
19      } else {
20          int m = (left + right) / 2;
21
22          struct msort_task mleft = {
23              .left = left,
24              .right = m,
25              .array = array,
26              .tmp = tmp
27          };
28          struct future * lhalf = thread_pool_submit(threadpool,
29                                      (fork_join_task_t) mergesort_internal_parallel,
30                                      &mleft);
31
32          struct msort_task mright = {
33              .left = m + 1,
34              .right = right,
35              .array = array,
36              .tmp = tmp
37          };
38          mergesort_internal_parallel(threadpool, &mright);
39          future_get(lhalf);
40          future_free(lhalf);
41          merge(array, tmp, left, left, m, right);
42      }
43  }
```

Change this code so that the calls to `future_free()` and `future_get()` are removed and add replacement code that ensures that the code continues to work as before, that is, that the merge step on 41 is performed only after the task representing the left half has been completed.

Hint: use a semaphore or condition variable.

Make sure your code will work even in the presence of recursion (line 38).

Solution: A possible approach is shown below.

```
1   /* msort_task describes a unit of parallel work */
2   struct msort_task {
3       int *array;
4       int *tmp;
5       int left, right;
6       sem_t finished;     // added
7   };
8
9   /* Parallel mergesort */
10  static void
```

```c
mergesort_internal_parallel(struct thread_pool * threadpool, struct msort_task * s)
{
    int * array = s->array;
    int * tmp = s->tmp;
    int left = s->left;
    int right = s->right;

    if (right - left <= min_task_size) {
        mergesort_internal(array, tmp + left, left, right);
    } else {
        int m = (left + right) / 2;

        struct msort_task mleft = {
            .left = left,
            .right = m,
            .array = array,
            .tmp = tmp
        };
        sem_init(&mleft.finished, 0, 0);   // added
        /*struct future * lhalf = */thread_pool_submit(threadpool,
                                    (fork_join_task_t) mergesort_internal_parallel,
                                    &mleft);

        struct msort_task mright = {
            .left = m + 1,
            .right = right,
            .array = array,
            .tmp = tmp
        };
        sem_init(&mright.finished, 0, 0);   // added
        mergesort_internal_parallel(threadpool, &mright);
        /* Removed
        future_get(lhalf);
        future_free(lhalf);
        */
        sem_wait(&mleft.finished);  // added: wait for subtask to finish
        merge(array, tmp, left, left, m, right);
        sem_post(&s->finished);  // added: signal parent that this task has finished
    }
}
```

3.2 Impact (5 pts)

Discuss the overall impact of this design change on correctness, usability, and performance. After the necessary changes to remove calls to `future_get` were made to the tests to compensate for the lack of `future_get`, will the existing thread pool implementation continue to work? If so, explain

why. If not, explain why not and outline any changes that may be necessary. You may refer to your own implementation or state your assumptions as necessary.

Solution: Not implementing future_get means that the threadpool cannot help with tasks that have been submitted, but have not yet been executed. Changing the client code to use semaphores or condition variables means that the worker thread executing them will be blocked. The pool will thus run out of worker threads and deadlock. (This effect was the motivation for having future_get in the first place.)

Fixing this would require an approach that ensures that there's always a worker available when a new task is spawned. Such a dynamically growing thread pool, however, would negate the goal of concurrency management (limiting the number of threads to a reasonable number that can execute without undue contention on the existing number of CPUs), because some workloads may spawn large numbers of tasks.

4 Optimizing Parallel Performance (8 pts)

Using a new visualization tool, you have obtained a timeline of the execution of a multi-threaded program you are currently optimizing, which can be seen Figure 1. Time moves from the top to the bottom, and different colors are used to denote phases of execution, as shown in the caption.

What step or steps would you recommend be taken next to optimize the parallel performance of this code? Justify your answer.

Solution: The timeline shows that the red lock is held only for short periods of time, and infrequently, causing hardly any reduction in CPU utilization.

The blue lock, however, is held for longer periods of time, and more frequently.

Therefore, optimization should be directed towards reducing the length and frequency of the time periods for which the blue lock is held, if possible. If the blue lock protects more than one data structure, breaking it up should be considered.

5 Deadlock (8 pts)

When using the Helgrind thread error detection tool, the following warnings appeared when running the executable `deadlock3`:

```
1  ==866307== Helgrind, a thread error detector
2  ==866307== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
3  ==866307== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
4  ==866307== Command: ./deadlock3
5  ==866307==
6  ==866307== ---Thread-Announcement------------------------------------------
7  ==866307==
8  ==866307== Thread #4 was created
9  ==866307==    at 0x515EE72: clone (in /usr/lib64/libc-2.28.so)
10 ==866307==    by 0x4E4A0CE: create_thread (in /usr/lib64/libpthread-2.28.so)
```

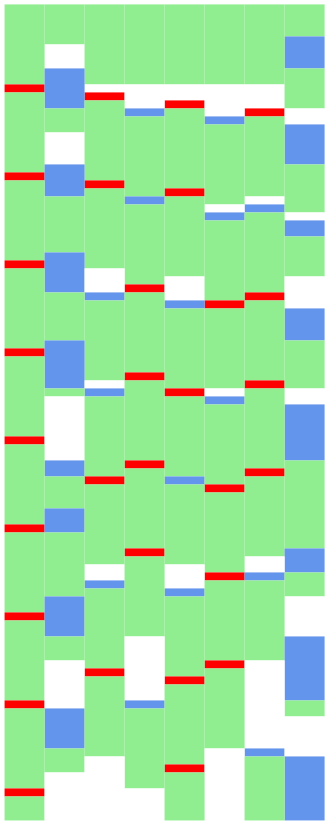

Figure 1: Timeline showing the execution of 8 threads on 8 CPUs. Each vertical stripe (or column) represents one thread. Time that is spent executing while holding no lock is shown in green. Time during which the CPU is idle is shown in white. Time spent holding a lock is shown in a color corresponding to the lock, either red 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.

```
==866307==    by 0x4E4BB55: pthread_create@@GLIBC_2.2.5 (in /usr/lib64/libpthread-2.28.so)
==866307==    by 0x4C3800B: pthread_create_WRK (hg_intercepts.c:427)
==866307==    by 0x4C39105: pthread_create@* (hg_intercepts.c:460)
==866307==    by 0x400804: main (deadlock3.c:51)
==866307==
==866307== ----------------------------------------------------------------
==866307==
==866307== Thread #4: lock order "0x601060 before 0x6010E0" violated
==866307==
==866307== Observed (incorrect) order is: acquisition of lock at 0x6010E0
==866307==    at 0x4C35638: mutex_lock_WRK (hg_intercepts.c:909)
==866307==    by 0x4C39504: pthread_mutex_lock (hg_intercepts.c:925)
```

```
==866307==    by 0x40076F: thread3 (deadlock3.c:36)
==866307==    by 0x4C38203: mythread_wrapper (hg_intercepts.c:389)
==866307==    by 0x4E4B2DD: start_thread (in /usr/lib64/libpthread-2.28.so)
==866307==    by 0x515EE82: clone (in /usr/lib64/libc-2.28.so)
==866307==
==866307==  followed by a later acquisition of lock at 0x601060
==866307==    at 0x4C35638: mutex_lock_WRK (hg_intercepts.c:909)
==866307==    by 0x4C39504: pthread_mutex_lock (hg_intercepts.c:925)
==866307==    by 0x400779: thread3 (deadlock3.c:37)
==866307==    by 0x4C38203: mythread_wrapper (hg_intercepts.c:389)
==866307==    by 0x4E4B2DD: start_thread (in /usr/lib64/libpthread-2.28.so)
==866307==    by 0x515EE82: clone (in /usr/lib64/libc-2.28.so)
==866307==
==866307==  Lock at 0x601060 was first observed
==866307==    at 0x4C35638: mutex_lock_WRK (hg_intercepts.c:909)
==866307==    by 0x4C39504: pthread_mutex_lock (hg_intercepts.c:925)
==866307==    by 0x4006DB: thread1 (deadlock3.c:14)
==866307==    by 0x4C38203: mythread_wrapper (hg_intercepts.c:389)
==866307==    by 0x4E4B2DD: start_thread (in /usr/lib64/libpthread-2.28.so)
==866307==    by 0x515EE82: clone (in /usr/lib64/libc-2.28.so)
==866307==  Address 0x601060 is 0 bytes inside data symbol "lock1"
==866307==
==866307==  Lock at 0x6010E0 was first observed
==866307==    at 0x4C35638: mutex_lock_WRK (hg_intercepts.c:909)
==866307==    by 0x4C39504: pthread_mutex_lock (hg_intercepts.c:925)
==866307==    by 0x40072F: thread2 (deadlock3.c:26)
==866307==    by 0x4C38203: mythread_wrapper (hg_intercepts.c:389)
==866307==    by 0x4E4B2DD: start_thread (in /usr/lib64/libpthread-2.28.so)
==866307==    by 0x515EE82: clone (in /usr/lib64/libc-2.28.so)
==866307==  Address 0x6010e0 is 0 bytes inside data symbol "lock3"
==866307==
```

Examine the output carefully, and then based on your knowledge about deadlocks reconstruct `deadlock3.c`.

You do not need to match the line numbers exactly, but you should match the places that say `thread1`, `thread2`, and `thread3` (those are the names of functions executing in different threads), as well as the lines that refer to a "data symbol" `lock1` and `lock3`.

Solution: The program is shown below. Note that the potential deadlock here involved 3 locks and 3 threads, with the obvious circularity $L_1 \to L_2$, $L_2 \to L_3$, $L_3 \to L_1$.

Helgrind observed locking orders $L_1 \to L_2$ and $L_2 \to L_3$ in threads 1 and 2, and it is able to transitively conclude that this implies $L_1 \to L_3$, though it doesn't explicitly say that it did that. Instead, it merely reports that it encountered `lock3` first in `thread2`.

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

```
3
4   #define NTHREADS 3
5   #define N 10
6   static int counter;
7   pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
8   pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
9   pthread_mutex_t lock3 = PTHREAD_MUTEX_INITIALIZER;
10
11  static void *
12  thread1(void * _tn)
13  {
14      pthread_mutex_lock(&lock1);
15      pthread_mutex_lock(&lock2);
16      counter++;
17      pthread_mutex_unlock(&lock2);
18      pthread_mutex_unlock(&lock1);
19      return NULL;
20  }
21
22  static void *
23  thread2(void * _tn)
24  {
25      pthread_mutex_lock(&lock2);
26      pthread_mutex_lock(&lock3);
27      counter++;
28      pthread_mutex_unlock(&lock3);
29      pthread_mutex_unlock(&lock2);
30      return NULL;
31  }
32
33  static void *
34  thread3(void * _tn)
35  {
36      pthread_mutex_lock(&lock3);
37      pthread_mutex_lock(&lock1);
38      counter++;
39      pthread_mutex_unlock(&lock3);
40      pthread_mutex_unlock(&lock1);
41      return NULL;
42  }
43
44
45  int
46  main()
47  {
48      pthread_t t[NTHREADS];
```

```
49      pthread_create(t + 0, NULL, thread1, NULL);
50      pthread_create(t + 1, NULL, thread2, NULL);
51      pthread_create(t + 2, NULL, thread3, NULL);
52
53      for (int i = 0; i < NTHREADS; i++)
54          pthread_join(t[i], NULL);
55
56      return 0;
57  }
```