CS 3214 Fall 2021 Test 2 Solutions

November 15, 2021

Contents

1 On Multithreading (12 pts)

Find out if the following statements related to multithreading are true or false. If true, just write **true**. If false, write **false** and provide a corrected statement that includes a concise explanation of why the original statement was false.

1. In general purpose OS such as Linux, threads often migrate between processes when they make remote-procedure calls.

   [False] Threads are bound to the process in which they were created. (Thread migration is an experimental concept that is not available in general purpose OS at this time.)

2. If one thread opens a file descriptor, other threads within the same process can access this file descriptor.

   [True]

3. False sharing occurs when multiple threads access the same atomic variable or lock.

   [False] Accessing the same variable or lock represents true sharing. False sharing occurs when unrelated variables share a cache line.

4. When the OS schedules multiple threads onto different CPUs or cores, the hardware's cache coherency protocol ensures that all threads on all cores see updates in exactly the same order relative to other reads or updates they make.

   [False] The hardware may reorder reads and writes, resulting in different threads seeing updates in different order.

5. Condition variables can be used in combination with semaphores, mutexes, or sometimes by themselves for signaling tasks.

   [False] Condition variables must always be used with mutexes, never by themselves, since they represent one third of the monitor abstraction.

6. If a shared variable is written by one thread before a call to `sem_post` then the shared variable can be read by another thread after a matching call to `sem_wait` without requiring a lock. (Assume that the semaphore's initial value was 0 and that only a single call to `sem_wait` and `sem_post` occurs.)

   [True] The semaphore in this case creates a happens-before relationship, thus the 2 accesses are not concurrent. (Note that the question asked just about these accesses, not any potential other accesses. If there were later updates then a lock may be required.)

2 Fundraising Threads (10 pts)

Consider this scenario: multiple threads are trying to raise a predetermined amount of funds, measured in units. Some threads will contribute one or more units towards the shared goal. Other threads will not, but need to be certain that the shared goal has been met before proceeding.

Implement the following methods in a header-only library `shared_goal.h`.

```c
/*
 * A header-only library to implement the
 * fund raising logic.
 */
struct shared_goal {
    // define any fields you need
};

/* Create a new shared goal of `goal` units */
static struct shared_goal *
make_a_new_shared_goal(int goal)
{
    // implement this
}

/* Contribute `units` units to the shared goal. */
static void
contribute(struct shared_goal *goal, int units)
{
    // implement this
}

/* Wait until the goal has been reached, blocking if
 * necessary.  Threads must not busy-wait. */
static void
wait_for_success(struct shared_goal *goal)
{
    // implement this
}

/*
 * Return how much is left to achieve the goal.
 * Caution: the returned number may be outdated
 * by the time it is checked to due to concurrent
 * contributions.
 */
static int
how_much_is_left(struct shared_goal *goal)
{
    // implement this
}
```

For the purposes of this question,

- you do not need to implement freeing/reclamation of the shared_goal structure.
- you may use an `int` to represent units and ignore contributions that may lead to overflow or underflow.

- you must not use busy waiting.
- your implementation must be data race free.
- your implementation may not use any global variables of any kind (static, non-static, or thread local).

An example use is shown below:

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

// a header-only library
#include "shared_goal.h"

// ----------------------------------------------------------
// A contributor thread. This thread contributes 1 unit
// and then waits for the goal to be reached.
static void *
contributor(void *_goal)
{
    struct shared_goal *goal = _goal;
    contribute(goal, 1);
    wait_for_success(goal);
    assert (how_much_is_left(goal) <= 0);
    return NULL;
}

// A bystander thread. This thread doesn't contribute, it
// just waits for the goal to be achieved.
static void *
bystander(void *_goal)
{
    struct shared_goal *goal = _goal;
    wait_for_success(goal);
    assert (how_much_is_left(goal) <= 0);
    return NULL;
}

#define N  20
int
main()
{
    pthread_t t[2*N];
    srand(time(NULL));
    struct shared_goal * goal = make_a_new_shared_goal(N);

    printf("starting %d bystanders and %d contributors\n", N, N);
    int bystanders = 0;
    for (int i = 0; i < 2*N; i++) {
        if ((rand() % 2 == 0 && bystanders < N)
            || i-bystanders >= N) {
            pthread_create(&t[i], NULL, bystander, goal);
            bystanders++;
        } else {
            pthread_create(&t[i], NULL, contributor, goal);
        }
    }

    for (int i = 0; i < 2*N; i++)
        pthread_join(t[i], NULL);

    printf("joined %d bystanders and %d contributors, left %d\n",
        bystanders, 2*N-bystanders, how_much_is_left(goal));
    return 0;
}
```

[Solution:]

The “fundraising threads” primitive is more commonly known as a countdown latch, such as `java.util.concurrent.CountdownLatch` or C++'s `std::latch`. It can be implemented as follows:

```
/*
 * A header-only library to implement the
 * fund raising logic.
 */
struct shared_goal {
    int left;
    pthread_mutex_t lock;
    pthread_cond_t cond;
};

/* Create a new shared goal of `goal` units */
static struct shared_goal *
make_a_new_shared_goal(int goal)
{
    struct shared_goal * rc = malloc(sizeof (*rc));
    pthread_mutex_init(&rc->lock, NULL);
    pthread_cond_init(&rc->cond, NULL);
    rc->left = goal;
    return rc;
}
```

```c
/* Contribute `units` units to the shared goal. */
static void
contribute(struct shared_goal *goal, int units)
{
    pthread_mutex_lock(&goal->lock);
    goal->left -= units;
    if (goal->left <= 0)
        pthread_cond_broadcast(&goal->cond);
    pthread_mutex_unlock(&goal->lock);
}

/* Wait until the goal has been reached, blocking if
 * necessary.  Threads must not busy-wait. */
static void
wait_for_success(struct shared_goal *goal)
{
    pthread_mutex_lock(&goal->lock);
    while (goal->left > 0)
        pthread_cond_wait(&goal->cond, &goal->lock);
    pthread_mutex_unlock(&goal->lock);
}

/*
 * Return how much is left to achieve the goal.
 * Caution: the returned number may be outdated
 * by the time it is checked to due to concurrent
 * contributions.
 */
static int
how_much_is_left(struct shared_goal *goal)
{
    pthread_mutex_lock(&goal->lock);
    int left = goal->left;
    pthread_mutex_unlock(&goal->lock);
    return left;
}
```

3 Unraveling a Stuck Program (12 pts)

When debugging a multithreaded program on our rlogin cluster, a programmer observed that the program “got stuck.” They then took the following steps to find out what was going on:

```
$ gcc -pthread -ggdb3 unravelingdeadlock.c -o unravelingdeadlock
$ gdb ./unravelingdeadlock
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-16.el8
```

```
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-redhat-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 ./unravelingdeadlock...done.
(gdb) run
Starting program: /home/courses/cs3214/test2-fall2021/unravelingdeadlock
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-167.el8.x86_64
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffff77ea700 (LWP 1135789)]
[New Thread 0x7ffff6fe9700 (LWP 1135790)]
[New Thread 0x7ffff67e8700 (LWP 1135791)]
^C
Thread 1 "unravelingdeadl" received signal SIGINT, Interrupt.
0x00007ffff7bb966d in __pthread_timedjoin_ex () from /lib64/libpthread.so.0
(gdb) info threads
  Id   Target Id                                         Frame
* 1    Thread 0x7ffff7fe3740 (LWP 1135785) "unravelingdeadl" 0x00007ffff7bb966d
                         in __pthread_timedjoin_ex () from /lib64/libpthread.so.0
  2    Thread 0x7ffff77ea700 (LWP 1135789) "unravelingdeadl" thread3 (_arg=0x0)
                         at unravelingdeadlock.c:31
  3    Thread 0x7ffff6fe9700 (LWP 1135790) "unravelingdeadl" 0x00007ffff7bc0cd6
                         in do_futex_wait.constprop () from /lib64/libpthread.so.0
  4    Thread 0x7ffff67e8700 (LWP 1135791) "unravelingdeadl" 0x00007ffff7bc175d
                         in __lll_lock_wait () from /lib64/libpthread.so.0
(gdb) thread apply all backtrace

Thread 4 (Thread 0x7ffff67e8700 (LWP 1135791)):
#0  0x00007ffff7bc175d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007ffff7bbaa79 in pthread_mutex_lock () from /lib64/libpthread.so.0
#2  0x0000000000400873 in thread2 (_arg=0x0) at unravelingdeadlock.c:25
#3  0x00007ffff7bb817a in start_thread () from /lib64/libpthread.so.0
#4  0x00007ffff78e7ee3 in clone () from /lib64/libc.so.6

Thread 3 (Thread 0x7ffff6fe9700 (LWP 1135790)):
```

```
#0  0x00007ffff7bc0cd6 in do_futex_wait.constprop () from /lib64/libpthread.so.0
#1  0x00007ffff7bc0dc8 in __new_sem_wait_slow.constprop.0 () from /lib64/libpthread.so.0
#2  0x0000000000400850 in thread1 (_arg=0x0) at unravelingdeadlock.c:18
#3  0x00007ffff7bb817a in start_thread () from /lib64/libpthread.so.0
#4  0x00007ffff78e7ee3 in clone () from /lib64/libc.so.6

Thread 2 (Thread 0x7ffff77ea700 (LWP 1135789)):
#0  thread3 (_arg=0x0) at unravelingdeadlock.c:31
#1  0x00007ffff7bb817a in start_thread () from /lib64/libpthread.so.0
#2  0x00007ffff78e7ee3 in clone () from /lib64/libc.so.6

Thread 1 (Thread 0x7ffff7fe3740 (LWP 1135785)):
#0  0x00007ffff7bb966d in __pthread_timedjoin_ex () from /lib64/libpthread.so.0
#1  0x0000000000400a1d in main () at unravelingdeadlock.c:59
(gdb) thread 4
[Switching to thread 4 (Thread 0x7ffff67e8700 (LWP 1135791))]
#0  0x00007ffff7bc175d in __lll_lock_wait () from /lib64/libpthread.so.0
(gdb) frame 2
#2  0x0000000000400873 in thread2 (_arg=0x0) at unravelingdeadlock.c:25
25            pthread_mutex_lock(&lock);
(gdb) p lock
$1 = {__data = {__lock = 2, __count = 0, __owner = 1135790, __nusers = 1, __kind = 0,
    __spins = 0, __elision = 0, __list = {__prev = 0x0, __next = 0x0}},
  __size = "\002\000\000\000\000\000\000\000\256T\021\000\001", '\000' <repeats 26 times>,
  __align = 2}
(gdb) quit
A debugging session is active.

        Inferior 1 [process 1135785] will be killed.

Quit anyway? (y or n) y
$
```

In addition, they took the following steps after quitting gdb:

```
$ ./unravelingdeadlock &
[1] 1143376
$ ps --pid 1143376 -o cmd,stat,pcpu,tid -L
CMD                         STAT %CPU     TID
./unravelingdeadlock        Sl    0.0 1143376
./unravelingdeadlock        Sl    0.0 1143377
./unravelingdeadlock        Sl    0.0 1143378
./unravelingdeadlock        Rl   99.5 1143379
$ kill 1143376
$ [1]+  Terminated              ./unravelingdeadlock
```

Part (a) (2 pts): Describe what led to the program getting stuck. Be concise.

[Solution:] The backtraces show 4 threads: the main thread blocked in `pthread_join`, `thread1` blocked in `sem_wait`, and `thread2` blocked in `pthread_mutex_lock`. Furthermore, we can see that `thread1` holds the mutex upon which `thread2` waits. The fourth thread (`thread3`) however is not blocked and in the running state, which is confirmed by the ps command.

Part (b) (2 pts): Is this scenario an example of deadlock, livelock, or does it not fit neatly into one of these categories? Justify your answer.

[Solution:] It does not fit neatly into these categories because with the information given, we can't easily tell what `thread3` is doing. For instance, if it simply performed some computation that would take a very long time, but then signaled the semaphore upon which `thread1` is waiting, it's possibly for the program to make progress as planned in the future. Or `thread3` could exit the process, terminating all threads in it. On the other hand, if `thread3` will never signal this semaphore or exit the process before it completes, the program will deadlock as the other threads are unable to make progress.

Part (c) (8 pts): Reconstruct the multithreaded program that would (always) get stuck in this way.

Your reconstructed program needs to include 3 threads in addition to the main thread. Here is a skeleton you should start from, with the areas of the code in which you can make changes marked.

```c
/* Unraveling a deadlock mystery. */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <semaphore.h>

/* you may add definitions here */

static void *
thread1(void *_arg)
{
    /* you may add code here */
}

static void *
thread2(void *_arg)
{
    /* you may add code here */
}

static void *
thread3(void *_arg)
{
    /* you may add code here */
}

typedef void *(*thread_fun)(void *);
```

```c
int
main()
{
    const int n = 3;
    pthread_t t[n];

    srand(time(NULL));
    /* you may add code here */
    thread_fun funcs[] = { thread1, thread2, thread3 };

    // Fisher-Yates
    for (int i = n-1; i >= 1; i--) {
        int j = random() % (i+1);
        thread_fun temp = funcs[i];
        funcs[i] = funcs[j];
        funcs[j] = temp;
    }
    for (int i = 0; i < n; i++)
        pthread_create(&t[i], NULL, funcs[i], NULL);

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

    return 0;
}
```

Your reconstructed program:

- should be *deterministic* in the sense that if the user runs the program under gdb and waits for a sufficient amount before hitting Ctrl-C, they will always observe what is shown in the gdb transcript above[1].
- must have 4 threads at the time it gets stuck and not fewer.
- may have a different naming for the threads, i.e., the threads named "Thread 2," "Thread 3," and "Thread 4" may run any permutation of `thread1`, `thread2`, and `thread3` since they are started in random order.
- may have different line numbers than the ones shown, but otherwise each thread's backtrace should be identical (for instance, `thread1` must be inside `do_futex_wait.constprop`, `thread2` must be inside `__lll_lock_wait`, and `thread3` must be on some line number in `unravelingdeadlock.c`.

Important: please make sure that you do not leave running processes on any rlogin machines if you decide to test your code. Be sure to kill any processes you create, as shown in the sample outputs.

[1] "sufficient amount" here means that we ignore the theoretical possibility that the scheduler may not allow the started threads to make progress for a long time.

[Solution:] For a correct solution it's important to use a reliable mechanism to ensure that `thread1` holds the lock before `thread2` attempt to acquire it. The sample solution uses the semaphore `sem_1_before_2` to that end.[2]

```
/* Solution to unraveling a deadlock mystery. */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <semaphore.h>

sem_t sem_deadlock;
sem_t sem_1_before_2;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

static void *
thread1(void *_arg)
{
    pthread_mutex_lock(&lock);
    sem_post(&sem_1_before_2);
    sem_wait(&sem_deadlock);
}

static void *
thread2(void *_arg)
{
    sem_wait(&sem_1_before_2);
    pthread_mutex_lock(&lock);
}

static void *
thread3(void *_arg)
{
    for (;;)
        ;
}

typedef void *(*thread_fun)(void *);

int
main()
{
    const int n = 3;
```

[2]Note that using `sleep()` here is not a reliable way of accomplishing ordering: no matter how long one threads sleeps, there is no guarantee that the other thread will have reached the point where it acquired the mutex.

```
    pthread_t t[n];

    srand(time(NULL));
    sem_init(&sem_deadlock, 0, 0);
    sem_init(&sem_1_before_2, 0, 0);
    thread_fun funcs[] = { thread1, thread2, thread3 };

    // Fisher-Yates
    for (int i = n-1; i >= 1; i--) {
        int j = random() % (i+1);
        thread_fun temp = funcs[i];
        funcs[i] = funcs[j];
        funcs[j] = temp;
    }
    for (int i = 0; i < n; i++)
        pthread_create(&t[i], NULL, funcs[i], NULL);

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

    return 0;
}
```

4 Parallel Algebra (10 pts)

Your math professor wants to use parallelism to speed up the execution of a common function in linear algebra. She's prepared a test case, which is shown below:

```
/*
 * Parallel Algebra
 */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/sysinfo.h>

#include "algebra.h"

int
main()
{
    srand(time(NULL));

#define N 100000007
```

```
    double *a = malloc(N * sizeof(double));
    double *b = malloc(N * sizeof(double));
    double serial_result = 0;
    for (int i = 0; i < N; i++) {
        a[i] = random() % 10000;
        b[i] = random() % 10000;
        serial_result += a[i] * b[i];
    }

    double parallel_res = parallel_algebra(a, b, N);
    printf("serial %f parallel %f\n",
        serial_result, parallel_res);

    return 0;
}
```

Also, one of her students started writing a parallel implementation, but unfortunately it's uncommented and incomplete:

```
struct unit_of_work {
    double *a, *b;
    size_t n;
    double dp;
};

static void *
algebra_helper(void *arg)
{
    struct unit_of_work *u = arg;
    double s = 0;
    double *a = u->a;
    double *b = u->b;
    for (size_t i = 0; i < u->n; i++)
        s += *a++ * *b++;

    u->dp = s;
    return NULL;
}

static double
parallel_algebra(double *a, double *b, size_t N)
{
    int n = get_nprocs();

    // complete this
}
```

Complete `algebra.h` to provide a parallel implementation of the algebraic function tested. You should use n threads where n is the number of processors reported by the `get_nprocs` function.

[Solution:]

```c
struct unit_of_work {
    double *a, *b;
    size_t n;
    double dp;
};

static void *
algebra_helper(void *arg)
{
    struct unit_of_work *u = arg;
    double s = 0;
    double *a = u->a;
    double *b = u->b;
    for (size_t i = 0; i < u->n; i++)
        s += *a++ * *b++;

    u->dp = s;
    return NULL;
}

static double
parallel_algebra(double *a, double *b, size_t N)
{
    int n = get_nprocs();

    // divide work
    pthread_t t[n];
    struct unit_of_work u[n];
    size_t off = 0;
    size_t chunk = (N + n - 1)/n;
    for (int i = 0; i < n; i++) {
        u[i].n = (chunk < N - off) ? chunk : (N - off);
        u[i].a = a + off;
        u[i].b = b + off;
        off += u[i].n;
    }

    // start n threads
    for (int i = 0; i < n; i++)
        pthread_create(&t[i], NULL, algebra_helper, &u[i]);

    // join n threads and aggregate subtotals
    double s = 0;
    for (int i = 0; i < n; i++) {
        pthread_join(t[i], NULL);
        s += u[i].dp;
    }
    return s;
}
```

5 On Parallel Performance (8 pts)

Using a visualization tool, you have obtained a timeline of the execution of 2 multi-threaded workloads running in 2 separate processes, as seen in Figure 1. Time moves from the top to the bottom, and different colors are used to denote phases of execution, as described in the caption. Process 1 contains threads 1 and 2 (the 2 leftmost columns), and process 2 contains thread 3 through 6 (the 4 rightmost columns).

Answer the following questions:

1. (2 pts) Estimate the CPU utilization of process 1 in percent, where 100% would be full utilization of 2 cores.

   [Solution:] 50% - each thread makes process only while holding the blue lock.

2. (2 pts) Estimate the CPU utilization of process 2 in percent, where 100% would be full utilization of 4 cores.

   [Solution:] Slightly less than 100% - the threads are hardly ever blocked on locks.

3. (2 pts) Give a real-world example of a workload that could cause the pattern observed for process 1.

   [Solution:] The pattern displayed is evident in any environment that uses a global "big" lock a thread must hold before executing - for instance, a Python interpreter or tools like valgrind whose core execution engine is not thread-safe.

4. (2 pts) Give a real-world example of a workload that could cause the pattern observed for process 2.

   [Solution:] Among many possible examples is a well-tuned task-parallel framework processing coarse-grained tasks, such as your p2 threadpool running mergesort, that could display the pattern shown - threads only occasionally acquire locks, and when they do, they are not unusually contended.

Briefly justify each answer.

6 Millions of threads, and counting! (12 pts)

In modern large scale computing there is often a need to run thousands or even millions of threads, e.g., when supporting deep learning and mathematical modeling of complex physical phenomena such as hurricane path prediction. However, creating and managing such large numbers of threads is not easy.

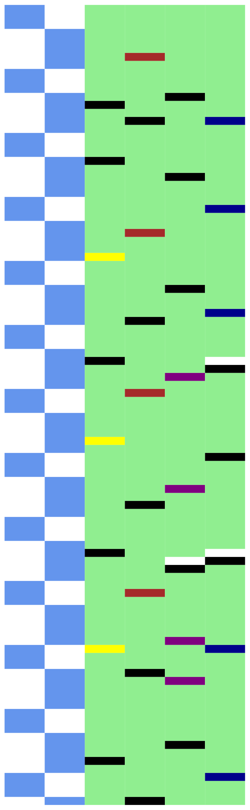

Figure 1: Timeline showing the execution of 6 threads on 6 CPU cores. Each vertical stripe (or column) represents one thread. Time that is spent executing while holding no lock is shown in green. Time during which the CPU is idle is shown in white. Time spent holding a lock is shown in a color corresponding to the lock, either black, blue, yellow, darkblue, brown, or purple. None of the threads engage in I/O or move into the BLOCKED state for any reason other than waiting for a lock.

1. (4 pts) What are the two main challenges in creating a very large number of threads in a commodity OS such as Linux? Provide fundamental reasons for this and not policy limits, e.g., maximum number of threads allowed per user, etc.

2. (4 pts) One can argue that given the number of cores in commodity processors is limited to tens of units, creating a very large number of threads is not needed to make the most of the available cores. Do you agree with this statement? Why or why not?

3. (4 pts) An alternate mechanism to extract more parallelism is to employ an event-based programming model instead of the thread-based mode. Use Internet search to determine the pros and cons of the two models, and then provide one (and only one) reason for why you would prefer one over the other.

Note: This is an open-ended question. Please provide brief answers with well-formed relevant arguments. Answers should be brief and under 100 words total.