CS 3214 Spring 2022 Test 2 Solution

April 4, 2022

Contents

1 On Multithreading (10 pts) **2**

2 Reasoning About Deadlock (6 pts) **2**

3 Twice as Safe (8 pts) **3**

4 Task Dependency Graphs (10 pts) **4**

5 `IllegalMonitorStateException` (12 pts) **10**

6 Threads, Threads, and More Threads (10 pts) **16**

7 A New Locking Algorithm (8 pts) **18**

8 Submission Requirements **21**

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. This includes sites such as chegg.com, which will be monitored. The open Internet stipulation does not apply to such sites.
- You are required to cite any sources you use, except for lecture material, source code provided as part of the class material, and the textbook. Failure to do so is an Honor Code violation.
- If you have a question about the exam, you may post it as a *private* question on Discourse addressing the instructors (account names: @Godmar_Back and @djwillia and @Liting_Hu). If your question is of interest to others, we will make it public as a clarification or hint.
- During the test period, you may not use Discourse to post other questions or reply to questions posted there.
- Any errata to this exam or hints given will be published prior to 12h before the deadline, please check Discourse under tag `test2`.

1 On Multithreading (10 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.

(a) In languages that support multi-threading, a thread's execution typically occurs inside some kind of "callable" construct such as a function or method.

True.

(b) The relationship between `fork()` and `wait()` is much like the relationship between `pthread_create()` and `pthread_join()`.

True.

(c) Even though Linux's implementation of POSIX Threads allows for the creation of multiple threads, only one of the these threads can be in the 'RUNNING' state at a time, even when the underlying machine uses a multiprocessor system.

False. *Linux provides kernel-support for threads which its NPTL POSIX library uses - if there are multiple cores, the scheduler can place different threads of a process in the 'RUNNING' state on different cores simultaneously, which provides for parallel speedup like you observed in project 2.*

(d) Context switches between 2 threads that are part of the same process are usually cheaper than context switches between 2 threads that belong to different processes.

True.

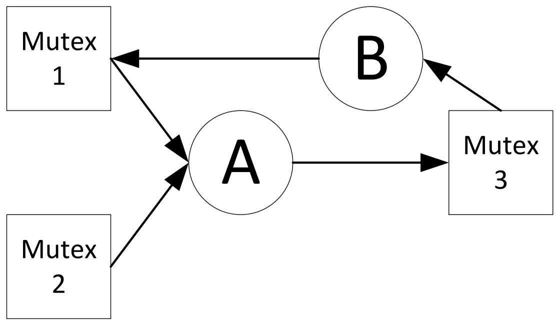
(e) Abstractions such as mutexes and semaphores may fail in situations where a system is under high load and thus needs to preempt threads more often, which increases the likelihood that they are preempted even while they are executing in a critical section.

False. *No, mutexes and semaphores are designed to work no matter what decisions are made by the scheduler. In particular, threads may be preempted while holding locks but this does not affect the mutual exclusion provided by these locks.*

2 Reasoning About Deadlock (6 pts)

Assume a process that has two threads (A, B) and three mutexes (mutex_lock_1, mutex_lock_2, mutex_lock_3). Thread A has acquired mutex_lock_1 and mutex_lock_2, and is now blocked trying to acquire mutex_lock_3. Thread B has acquired mutex_lock_3, and is now blocked trying to lock mutex_lock_1.

(a) Draw the resource allocation graph that describes this situation.



Resource allocation graph

(b) Based on the resource allocation graph, can you determine whether this system is deadlocked or not? Justify your answer.

Yes. *The 4 necessary conditions for a deadlock to happen are: 1) Exclusive Access; 2) Hold and Wait; 3) No Preemption; 4) Circular Wait. A mutex is a resource that is non-preemptive and has exclusive access. Threads A and B hold at least one mutex and wait for another. There is a circle in the resource allocation graph. Therefore, this system is deadlocked.*

3 Twice as Safe (8 pts)

When learning that locks could make their code safe from data races, a student decided to use two locks to be extra sure that a shared counter was protected.

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

static int counter;
static pthread_mutex_t l1 = PTHREAD_MUTEX_INITIALIZER; // protects counter
static pthread_mutex_t l2 = PTHREAD_MUTEX_INITIALIZER; // protects counter again

static void *thread(void * _tn) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&l1);
        pthread_mutex_lock(&l2);
        counter++;
        pthread_mutex_unlock(&l2);
        pthread_mutex_unlock(&l1);
    }
    return NULL;
}

int main() {
    int i, N = 10;
    pthread_t t[N];
    for (i = 0; i < N; i++)
        pthread_create(t + i, NULL, thread, NULL);
    for (i = 0; i < N; i++)
        pthread_join(t[i], NULL);
    printf("counter = %d\n", counter);
    return 0;
}
```

Questions:

(a) (4 pts) Is it possible that this code will deadlock? Why or why not?

Answer: *No, this code does not have a circular wait condition (l1 is always taken before l2) so it will not deadlock.*

(b) (4 pts) What downsides, if any, are there in using the above approach (with two locks) vs. a standard approach that uses only a single lock?

Answer: *The main downside is performance; even though l2 will always be uncontended there is a cost for the atomic exchange operation when acquiring l2, resulting in a larger critical section for l1 and a higher likelihood of contention on l1 and trips to the kernel to wait.*

4 Task Dependency Graphs (10 pts)

In project 2, you developed a task-parallel framework that could express certain task dependencies: for instance, a recursive task spawned one or more subtasks, then joined them before returning. This could be viewed as a dependency of the parent task on the subtask (or -tasks).

In some situations, the set of tasks that can be executed in parallel and their dependencies on each are statically known. In this problem, you are asked to prototype an example of such a framework in miniature. There will be 4 tasks A, B, C, and D numbered 0 through 3, inclusive. These tasks are represented by a function that simply prints `A`, `B`, `C`, or `D`, respectively. Each task is executed in a dedicated thread.

The dependencies between the tasks are given as a graph that is represented by an 4×4 adjacency matrix of values $a_{i,j}$. If $a_{i,j} = 1$ then task i depends on task j - that is, task j must have completed before task i can be started.

Complete the program shown below such that the tasks execute according to the dependency matrix given in the `dependencies` variable. Your implementation must work if this variable is replaced with any valid dependency matrix. Valid dependency matrices are those that do not contain cycles.

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

```c
#include <stdbool.h>
#include <semaphore.h>

#define N 4

/*
 * An example dependency graph.
 *
 * A -> B -> C and D -> C
 * B depends on A, C depends on B, and C depends on D
 *
 * Therefore, the only 3 possible outputs are
 *   ABDC
 *   ADBC
 *   DABC
 */
static int dependencies[N][N] = {
    { 0, 0, 0, 0 },
    { 1, 0, 0, 0 },
    { 0, 1, 0, 1 },
    { 0, 0, 0, 0 },
};

static void task_A(void) { printf("A"); }
static void task_B(void) { printf("B"); }
static void task_C(void) { printf("C"); }
static void task_D(void) { printf("D"); }

struct task_descriptor {
    int id;
    void (*my_task) (void);
};

static struct task_descriptor task_descriptor[] = {
    { .id = 0, .my_task = task_A },
    { .id = 1, .my_task = task_B },
    { .id = 2, .my_task = task_C },
    { .id = 3, .my_task = task_D },
};

/* A Fisher-Yates shuffle */
static void
fisher_yates(uint8_t *deck, uint8_t n)
{
    for (int i = n-1; i > 0; i--) {
        int j = random() % (i+1);
        uint8_t tmp = deck[j];
        deck[j] = deck[i];
        deck[i] = tmp;
    }
}

/***************************************************************
 * Begin of region you can change
 * Be sure to retain task_thread, however.
 */

static void *
task_thread(void *_td)
{
    struct task_descriptor *td = _td;
    /* Implement this function.
     * Each task needs to ensure that its dependent
     * tasks have executed, then execute `td->my_task`
     *
     * Different task_threads may need to coordinate
     * to achieve this.
     */

    return NULL;
}

int
main()
{

/* End of region you can change
***************************************************************/
    srand(getpid());

    // Note: shuffling the starting order of these threads
    // is not required to produced nondeterminism.  It
    // merely amplifies it.  Even with this shuffle, the
    // 4 task threads are under the control of the scheduler
    // which provides no guarantees about the order in which
    // they execute.
    uint8_t start[4] = {0, 1, 2, 3};
    fisher_yates(start, 4);

    pthread_t t[N];
    for (int i = 0; i < N; i++)
        pthread_create(&t[i], NULL, task_thread,
                        task_descriptor + start[i]);

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

    printf ("\n");
    return 0;
}
```

Notes

- Your implementation should ensure that tasks that do not have a direct or indirect dependency on each other can execute in parallel.
- Your program should not busy-wait.
- Your program should be free of data races. Note that it is not necessary to protect access to the `dependencies` array itself since it is not (and must not be) changed while the threads are running.
- You may define global variables as needed.
- You may use semaphores and/or condition variables.
- You should make changes only in the marked region of the code.

[Solution]

A solution can be implemented with $N \times N$ semaphores initialized with 0, each representing an edge in the dependency graph. Threads wait for the tasks on which they depend and signal the tasks that depend on them.

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

#define N 4

/*
 * An example dependency graph.
 *
 * A -> B -> C and D -> C
 * B depends on A, C depends on B, and C depends on D
 *
 * Therefore, the only 3 possible outputs are
 *   ABDC
 *   ADBC
 *   DABC
 */
static int dependencies[N][N] = {
    { 0, 0, 0, 0 },
    { 1, 0, 0, 0 },
    { 0, 1, 0, 1 },
    { 0, 0, 0, 0 },
};

static void task_A(void) { printf("A"); }
static void task_B(void) { printf("B"); }
static void task_C(void) { printf("C"); }
static void task_D(void) { printf("D"); }

struct task_descriptor {
    int id;
    void (*my_task) (void);
};

static struct task_descriptor task_descriptor[] = {
    { .id = 0, .my_task = task_A },
    { .id = 1, .my_task = task_B },
    { .id = 2, .my_task = task_C },
    { .id = 3, .my_task = task_D },
};

/* A Fisher-Yates shuffle */
static void
fisher_yates(uint8_t *deck, uint8_t n)
{
    for (int i = n-1; i > 0; i--) {
        int j = random() % (i+1);
        uint8_t tmp = deck[j];
        deck[j] = deck[i];
        deck[i] = tmp;
    }
}

/***************************************************************
 * Begin of region you can change
 * Be sure to retain task_thread, however.
 */
sem_t S[N][N];
```

```
static void *
task_thread(void *_td)
{
    struct task_descriptor *td = _td;
    /* Implement this function.
     * Each task needs to ensure that its dependent
     * tasks have executed, then execute `td->my_task`
     *
     * Different task_threads may need to coordinate
     * to achieve this.
     */
    for (int i = 0; i < N; i++)
        if (dependencies[td->id][i])
            sem_wait(&S[td->id][i]);

    td->my_task();

    for (int i = 0; i < N; i++)
        if (dependencies[i][td->id])
            sem_post(&S[i][td->id]);

    return NULL;
}

int
main()
{
    for (int i = 0; i < N; i++)
        sem_init(&S[i][i], 0, 0);

/* End of region you can change
**************************************************************/
    srand(getpid());

    // Note: shuffling the starting order of these threads
    // is not required to produced nondeterminism.  It
    // merely amplifies it.  Even with this shuffle, there
    // 4 task threads are under the control of the scheduler
    // which provides no guarantees about the order in which
    // they execute.
    uint8_t start[4] = {0, 1, 2, 3};
    fisher_yates(start, 4);

    pthread_t t[N];
    for (int i = 0; i < N; i++)
        pthread_create(&t[i], NULL, task_thread,
                          task_descriptor + start[i]);

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

    printf ("\n");
    return 0;
}
```

5 `IllegalMonitorStateException` (12 pts)

Java provides built-in support for the monitor pattern. In Java, every object can be used as a monitor. The `synchronized` keyword can be used to execute a block of code inside the monitor under the protection of a lock that is associated with the monitor. Each such monitor is provided with a single condition variable that is accessed via the methods `java.lang.Object.wait` and `java.lang.Object.notify` to wait for and signal this condition variable, respectively.

To use the monitor pattern correctly, `java.lang.Object.wait` and `java.lang.Object.signal` must be called while in the monitor. In fact, Java will throw a special type of exception called `IllegalMonitorStateException` if programmers disregard this rule.

Unfortunately, in Linux's implementation of the POSIX standard, condition variables lack any error checking in their default configuration. If a thread calls `pthread_cond_wait` without holding the associated mutex, results will be undefined.

In this question, you are asked to implement a Java-style error-checking monitor, which should provide 5 methods:

```
#include <pthread.h>
/* add header files as necessary */

struct monitor {
    /* implement this */
};

/* Initialize this monitor. */
static void
monitor_init(struct monitor *m) {
    /* implement this */
}

/* Enter this monitor. */
static void
monitor_enter(struct monitor *m) {
    /* implement this */
}

/* Exit this monitor. */
static void
```

```
monitor_exit(struct monitor *m) {
    /* implement this */
}

/* If calling thread is executing inside the
 * monitor, wait on this monitor's condition variable
 * and then return true.
 * Returns false if calling thread is not in the monitor.
 */
static bool
monitor_wait(struct monitor *m) {
    /* implement this */
}

/* Signal this monitor's condition variable if calling
 * thread is in the monitor and then return true.
 * Returns false if calling thread is not in the monitor.
 */
static bool
monitor_signal(struct monitor *m) {
    /* implement this */
}
```

An example program that uses this monitor is given below:

```
/*
 * Synchronization via the monitor pattern.
 * Unbounded buffer example.
 */
#include <pthread.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include "list.h"

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

struct item {
    int item;
    struct list_elem elem;
};

struct monitor monitor;
struct list queue;        // accessed only inside the monitor
```

```
/* Produce one item and place it into the queue. */
void
produce(int _item)
{
    struct item *item = malloc(sizeof *item);
    assert (item != NULL);  // ignore memory exhaustion
    item->item = _item;

    monitor_enter(&monitor);                // Enter monitor

    list_push_front(&queue, &item->elem);
    bool rc = monitor_signal(&monitor);     // wake up consumer (if any)
    assert (rc == true);
    monitor_exit(&monitor);                 // Leave monitor
}

/* Consume one item from the queue and return it. */
int
consume()
{
    monitor_enter(&monitor);                // Enter monitor
    while (list_empty(&queue))
        assert(monitor_wait(&monitor) == true);

    struct list_elem *e = list_pop_back(&queue);
    struct item *item = list_entry(e, struct item, elem);
    int _item = item->item;
    free (item);
    monitor_exit(&monitor);                 // Leave monitor
    return _item;
}

/* A buggy attempt at consuming an item - waits
 * without being inside the monitor.
 * This must be detected and fail.
 */
void
buggy_consume()
{
    assert(monitor_wait(&monitor) == false);
}

/* A buggy attempt at producing an item - signals
 * without being inside the monitor.
 * This must be detected and fail.
 */
void
buggy_produce()
{
    assert(monitor_signal(&monitor) == false);
}

/* Produce a number of items.
 * Occasionally invoke buggy_produce
 */
static void *
producer(void *_items_to_produce)
{
    int n = *(int *)_items_to_produce;
    for (int i = 0; i < n; ) {
        if (rand() % 5 == 0) {
            buggy_produce();
        } else {
            produce(i);
            i++;
        }
    }

    return NULL;
}

/* Consume a number of items, return their sum. */
static void *
consumer(void *_items_to_consume)
{
    int n = *(int *)_items_to_consume;
    uintptr_t sum = 0;
    for (int i = 0; i < n; ) {
        if (rand() % 5 == 0) {
            int item = consume();
            sum += item;
            i++;
        } else
            buggy_consume();
    }

    return (void *) sum;
}

int
main()
```

```
{
#define N 4
    int items [N] =             { 30000,    20000,    40000,    10000 };
    void * (*func [N])(void*) = { consumer, consumer, producer, producer };

    srand(getpid());
    monitor_init(&monitor);
    list_init(&queue);

    pthread_t t[N];
    for (int i = 0; i < N; i++)
        pthread_create(&t[i], NULL, func[i], items + i);

    long sum = 0;
    for (int i = 0; i < N; i++) {
        uintptr_t val;
        pthread_join(t[i], (void **) &val);
        sum += val;
    }
    printf("sum %ld\n", sum);
    assert (sum == items[2] * (items[2]-1) / 2
                 + items[3] * (items[3]-1) / 2);
    return 0;
}
```

The program relies on our list implementation, so be sure to link it with it.
Notes:

- You should use POSIX mutex(es) and condition variable(s) in your implementation.
- You may not use mutexes with the error checking feature turned on. While these mutexes would help with detecting errors in `monitor_wait`, they would not be able to detect the case where code tries to signal without being in the monitor, so overall they would not contribute to a complete solution of the problem given.
- Your solution should be data race free. You may use C11 atomic variables if this is necessary/useful. Remember that Helgrind will flag accesses to atomic variables as data races if they are not protected by a lock. If you use atomic variables, document in the code how they avoid data races.
- The sample program should run reliably, even when invoked many times.

[Solution]
To solve this problem, the monitor should include a mutex and a condition variable which is used in the respective functions according to the monitor pattern. To check ownership of the mutex, one or two additional fields can be used.

These fields must be protected. If the API is used correctly, the monitor's mutex will protect them. In the case where the API is used incorrectly, it is possible to make them atomic to check their state. Alternatively, an additional mutex may be used.

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

struct monitor {
    pthread_mutex_t lock;
    pthread_cond_t cond;
    // these are atomic to allow access without holding the
    // lock to check whether the lock is held
    _Atomic pthread_t holder;
    _Atomic bool monitorheld;
};

/* Initialize this monitor. */
static void
monitor_init(struct monitor *m) {
    m->monitorheld = false;
    pthread_mutex_init(&m->lock, NULL);
    pthread_cond_init(&m->cond, NULL);
}

/* Enter this monitor. */
static void
monitor_enter(struct monitor *m) {
    pthread_mutex_lock(&m->lock);
    m->holder = pthread_self();
    m->monitorheld = true;
}

/* Exit this monitor. */
static void
monitor_exit(struct monitor *m) {
    m->monitorheld = false;
    pthread_mutex_unlock(&m->lock);
}

/* If calling thread is executing inside the
 * monitor, wait on this monitor's condition variable
 * and then return true.
 * Returns false if calling thread is not in the monitor.
 */
static bool
monitor_wait(struct monitor *m) {
    if (m->monitorheld && pthread_equal(m->holder, pthread_self())) {
        m->monitorheld = false;
        pthread_cond_wait(&m->cond, &m->lock);
        m->holder = pthread_self();
        m->monitorheld = true;
        return true;
    } else {
        // fprintf(stderr, "IllegalMonitorStateException in monitor_wait\n");
        return false;
    }
}

/* Signal this monitor's condition variable if calling
 * thread is in the monitor and then return true.
 * Returns false if calling thread is not in the monitor.
 */
static bool
monitor_signal(struct monitor *m) {
    if (m->monitorheld && pthread_equal(m->holder, pthread_self())) {
        pthread_cond_signal(&m->cond);
        return true;
    } else {
        // fprintf(stderr, "IllegalMonitorStateException in monitor_signal\n");
        return false;
    }
}
```

6 Threads, Threads, and More Threads (10 pts)

In a blog post dated Mar 21, 2022, the Windows expert Pavel Yosifovich discusses their observations about multithreading in Windows [1].

(a) The author writes:

> Looking at a typical Windows system shows thousands of threads, with process numbers in the hundreds, even though the total CPU consumption is low, meaning most of these threads are doing nothing most of the time. I typically rant about it in my Windows Internals classes. Why so many threads?

(2 pts) Assuming the author's observation is correct, why do "most of these threads do nothing most of the time?"

[Answer] Because these threads are in what we termed the 'BLOCKED' state - waiting for some kind of event to occur. In this state, they do not consume CPU time.

(b) After examining some of the information provided by Windows' diagnostic tools, the author opines:

[1] See https://scorpiosoftware.net/2022/03/21/threads-threads-and-more-threads/

> In an ideal world, the number of threads in a system would be the same as the number of logical processors – any more and threads might fight over processors, any less and you're not using the full power of the machine.

(3 pts) Do you agree with the author's statement? Justify your answer.

[Answer] This statement, as written, is difficult to defend. It is an oversimplification at best.

The statement would make sense if the author referred to runnable threads (that are in the 'READY' or 'RUNNING' state) - it's a common optimization goal to match the number of ready/running threads to the number of available processors (like we did in our p2 benchmarks).

Applied to all threads in a system, no matter their state, and no matter their purpose, the statement is harder to defend, because the threads envisioned in the author's "ideal world" would then have to perform and multiplex all functionality in the system, which would be a radical departure from existing systems design.

(c) Finally, the author concludes with this paragraph:

> Many applications today depend on various libraries and frameworks, some of which don't seem to care too much about using threads economically – examples include Node.js, the Electron framework, even Java and .NET. Threads are not free – there is the ETHREAD and related data structures in the kernel, stack in kernel space, and stack in user space. Context switches and code run by the kernel scheduler when threads change states from Running to Waiting, and from Waiting to Ready are not free, either.
>
> Many desktop/laptop systems today are very powerful and it might seem everything is fine. I don't think so. Developers use so many layers of abstraction these days, that we sometimes forget there are actual processors that execute the code, and need to use memory and other resources. None of that is free.

(5 pts) Discuss this passage. Is the author factually correct, and do these facts support the author's conclusion that "(not) everything is fine?"

Notes: For this problem, as you may encounter a number of opinions others have voiced about this blog post, it is important that you provide your own arguments based on the facts that you have learned in class and/or observed in the projects and exercises. To the extent that you use others' arguments, you must (a) properly cite them and (b) integrate and discuss them in your statement, as opposed to merely repeating and citing them.

[Answer] Let's first discuss whether the facts listed by the author are correct: frameworks such as node.js do spawn multiple threads (node.js, for instance, needs to use a threadpool to offload I/O). The Java Virtual Machine uses additional threads for garbage collection and just-in-time compilation even before threads created by the user.

Threads do impose a cost: there is a TCB (thread control block) inside the kernel, which is called the ETHREAD (executive thread) structure in Windows, and threads also have their stack - actually, 2 - one while executing in user mode and a second one that's reserved for when they make system calls. And there is an overhead when threads switch between states (Ready, Running, and Blocked (which is called Waiting in Windows).

However, the author's argument is clearly vulnerable in that the author fails to quantify either cost.

Aside from the per-thread data structures, the primary memory cost of a thread is in its stacks. The amount of memory for a thread's kernel stack in Windows is 3 pages[2] (12 KB, Linux: 16KB), which is typically physical memory.

The amount of memory on the user side is virtual memory, with physical memory being allocated on demand. It is reasonable to assume that most of the threads existing in the system have relatively small user stacks. If we conservatively estimate 64KB in physical memory on average per stack the total cost would come to 446 MB, which would be 10.9% of the memory available on a 4GB computer, or 2.7% on a 16GB computer.

However, the cost of context switching does not directly depend on the number of threads in a system - it depends on how often these threads switch between states, and that in turn depends on their purpose and what the system is used for. For instance, in the frameworks the author discusses such as node.js, threads become active only when needed - for instance, in node.js's case, when blocking I/O needs to be performed (or other tasks are to be offloaded for performance gain). More threads thus does not equal more context switches by themselves. Most of the thousands of threads in the system are in the blocked state and will likely very infrequently cause context switches unless the user activates a workload for which they were designed.

7 A New Locking Algorithm (8 pts)

A student came up with an idea of how to implement mutual exclusion and solve the problem of critical sections without requiring OS support, by just using logic and a shared variable. To that end, they implemented two functions `enter()` and `leave()` which are part of the program shown below.

They weren't quite sure what modifier, if any, they needed before their shared variables, so they placed a to-be-#defined placeholder `MODIFIER` in front of them.

```c
#include <pthread.h>
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdint.h>
#include <stdatomic.h>

static MODIFIER int  whose_turn_is_it = 0;
_Thread_local int threadid;

static void
enter()
{
    int me = threadid;

    while (whose_turn_is_it != me)
```

[2]Source: microsoft.com

```
        continue;
}

static void
leave()
{
    int me = threadid;
    int other = 1 - me;

    whose_turn_is_it = other;
}

static long count;

static void *
thread_function (void *_id)
{
    threadid = (uintptr_t) _id;

    for (int i = 0; i < 100000; i++) {
        enter();
        count++;
        leave();
    }
}

int
main()
{
    pthread_t t[2];
    for (uintptr_t i = 0; i < 2; i++)
        pthread_create(&t[i], NULL, thread_function, (void *) i);

    for (int i = 0; i < 2; i++) {
        pthread_join(t[i], NULL);
    }
    printf("count %ld\n", count);
    return 0;
}
```

To test the implementation, they used two threads trying to increment a counter inside the critical section.

(a) (3 pts) First, they compile this code like so:

```
gcc -pthread -DMODIFIER= -O2 newmutex.c -o newmutex
```

When they run this code, it appears to get stuck until they hit Ctrl-C:

```
$ ./newmutex
^C
```

Explain why this program gets stuck when compiled with optimization level 2.

Answer: *The compiler does not recognize that* `whose_turn_is_it` *can be modified by another thread, so it pulls the reading of* `whose_turn_is_it` *from memory out of the loop as an optimization, resulting in an infinite loop.*

(b) (3 pts) Second, they compile their code like so:

```
gcc -pthread -DMODIFIER=_Atomic -O2 newmutex.c -o newmutex
```

it now reliably produces the answer they expected:

```
$ ./newmutex
count 200000
```

After using the `_Atomic` keyword as modifier, did they in fact provide a correct solution to the general mutual exclusion problem for 2 threads, or did their solution just appear to work for the test they had chosen? A general solution includes these two requirements:

- only one thread can be in the critical section at the same time.
- if there are threads trying to enter the critical section and no threads currently in the critical section, one must eventually succeed.

Briefly justify your answer.

Answer: *This does not provide a solution for the general mutual exclusion problem for 2 threads because a thread cannot enter the critical section without the other thread exiting, meaning that it will only work if threads can/need to access the critical section the same number of times in a lock-step fashion. For example, if thread A only needs to enter the critical section 10 times and thread B needs to enter 100 times, a situation will occur where thread B is waiting but will never succeed.*

(c) (2 pts) For the specific 2 thread counter problem above, what drawback would this approach of protecting access to the counter have when compared to, say, traditional POSIX mutexes?

Answer: *A drawback of this approach is that it relies on busy waiting, wasting CPU cycles while checking the* `whose_turn_is_it` *variable. POSIX mutexes do not busy wait and waste CPU cycles in this way.*

8 Submission Requirements

Submit a tar file that contains the following files:

- `multithreading.txt` with the answers related to question 1.
- `depgraph.c` to answer Question 4. This program should compile and run and may also be tested on other cases with different dependency graphs.
- `monitor.h` to answer Question 5. This header file, when included into `monitor-sample.c`, should compile and run.
- A PDF or PNG file with your answer to 2 (a). This can be a photo, scan, or drawing.
- `answers.txt` with answers to Questions 2 (b), 3, 6, and 7.

Please verify that the autograder accepted your submission and was able to extract all required files. The autograder will not run any code, but rather it will only check that all required files were untarred. We will apply the grade penalty specified in the resubmission policy for submissions that were flagged by the autograder as being incomplete or not meeting the required format.