CS 3214 Fall 2025 Test 2 Solutions

October 30, 2025

Contents

1 Loading and Linking (32 pts)

1.1 Understanding Separate Compilation (24 pts)

Consider the following header file `header.h`:

```
int x = 0;
extern int y;
static double z;
extern void second_fun(int);
```

and the C source files `lmain.c` and `lsecond.c`:

```
// lmain.c
#include "header.h"

int main()
{
    z = 1.0;
    y = (int)z;
    second_fun(y);
}
```

```
// lsecond.c
#include "header.h"
#include <stdio.h>

void second_fun(int y)
{
    printf("Second function called"
           " x = %d, y = %d z = %f\n",
            x, y, z);
}
```

a) (2 pts) Provide a command or commands to compile, but not link, both `lmain.c` and `lsecond.c` on our rlogin machines.

`gcc -c lmain.c lsecond.c`

Also accepted was `gcc -S lmain.c lsecond.c` or separate invocations of `gcc -c lmain.c` followed by `gcc -c lsecond.c`

b) (2 pts) What error messages, if any, will the compiler output?

The compiler will not output any error messages.

c) (2 pts) Which files will result from compiling these two source files based on the command you provided?

The file `lmain.o` and `lsecond.o` were created.

(If you used `gcc -S` above, your answer here should be `lmain.s` and `lsecond.s`.

d) (4 pts) Which of the following represents the correct symbol table for the object module resulting from compiling `lmain.c`? (Addresses were omitted.) Assume that the compiler will not emit common symbols as per recent conventions. Check one of A, B, C, or D.

☐ A	☐ B	☐ C	☑ D
`T main`	`T main`	`T main`	`T main`
`U second_fun`	`U second_fun`	`T second_fun`	`U second_fun`
`U y`	`C x`	`B x`	`B x`
`U z`	`U y`	`B y`	`U y`
	`b z`	`b z`	`b z`

The program defines `main` as a function, `x` as a global symbol, `z` as a local symbol (both BSS), and it declares and uses `second_fun` and `y`, but defines neither, hence the unresolved `U` entry.

e) (4 pts) Which of the following represents the correct symbol table for the object module resulting from compiling `lsecond.c`? (Addresses were omitted.) Check one of A, B, C, or D.

☐ A	☑ B	☐ C	☐ D

`U printf`
`T second_fun`
`U y`
`B x`
`b z`
	`U printf`
	`T second_fun`
	`B x`
	`b z`
		`T second_fun`
		`B x`
		`b z`
			`U printf`
			`T second_fun`
			`B x`
			`U y`

The program defines `second_fun` as a function, it defines `x` as a global symbol, `z` as a local symbol (both BSS). It does not, however, reference `y` (note that the symbol `y` in `second_fun` is a parameter, not a reference to a global variable.) Therefore, there is no `U y` entry.

f) (2 pts) Provide the command to invoke the linker to produce a final executable which shall be called `lmain`.

The command is `gcc lmain.o lsecond.o -o lmain` or `gcc lmain.c lsecond.c -o lmain`
If you answered `gcc -S` above, you should answer `gcc lmain.c lsecond.c -o lmain` here.

g) (8 pts) Will invoking the linker command succeed in creating an executable? If you answer no, describe which error message(s) the linker will output. If you answer yes, provide the output of the program when it is run.

☐ yes / ☑ no

You will get two errors:

- multiple definition of ‘x’
- undefined reference to ‘y‘

since ‘x‘ was defined in a header file that was included in two translation units, and ‘y‘ was never defined in any translation unit.

1.2 Understanding Linking (8 pts)

Check if the following statements about compilation and linking are true or false.

a) Linking is a process that combines object and archive files in order to create executables or shared object files (aka dynamic libraries).

☑ true / ☐ false

b) Definitions of static functions produce global symbols, whereas the omission of the static modifier produces local symbols.

☐ true / ☑ false

It's the other way around.

c) Statically linked executables are generally larger than dynamically linked executables for the same program.

☑ true / ☐ false

d) In the linker command line, changing the order of libraries and/or object files can affect the success or failure of the linking process.

☑ true / ☐ false

e) Position-independent code simplifies the creation of shared libraries by removing restrictions on where in a virtual address space a library may be loaded.

☑ true / ☐ false

f) Programs can rely on the relative order in which the global variables they define are allocated in memory.

☐ true / ☑ false

No, this is unspecified behavior, and typically not even implementation-defined.

g) Compilers are specific to languages such as C, C++, Go, or Rust, but linkers can process and combine ELF object modules created by any compiled language.

☑ true / ☐ false

h) Link-time optimization (or LTO) allows the inlining of functions even when the compiler did not process a definition of a function when compiling the C code of the translation unit in which the function was called.

☑ true / ☐ false

2 Analyzing Programs for Data Races (16 pts)

Consider the following program:

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <semaphore.h>
5  #include <stdbool.h>
6
7  // global state
8  pthread_mutex_t lock
9     = PTHREAD_MUTEX_INITIALIZER;
10 sem_t sem;
11 int ai;
12 int g;
13
14 static void *
15 first_thread(void *_unused)
16 {
17     pthread_mutex_lock(&lock);
18     g++;
19     pthread_mutex_unlock(&lock);
20     sem_wait(&sem);
21     ai = ai + g;
22     return NULL;
23 }
24
25 static void *
26 second_thread(void *_unused)
27 {
28     pthread_mutex_lock(&lock);
29     int gbefore = g++;
30     pthread_mutex_unlock(&lock);
31     ai = gbefore;
32     sem_post(&sem);
33     return NULL;
34 }
```

```
35 int main() {
36     sem_init(&sem, 0, 0);
37     pthread_t t[2];
38     void * (*func [2])(void*) = {
39         first_thread, second_thread
40     };
41
42     g = 40;
43     for (int i = 0; i < 2; i++)
44         pthread_create(&t[i], NULL,
45                        func[i], NULL);
46
47     for (int i = 0; i < 2; i++)
48         pthread_join(t[i], NULL);
49     printf ("%d\n", g);
50     printf ("%d\n", ai);
51     return 0;
52 }
```

The C11 memory model defines a data race 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 another*
>
> *If a data race occurs, the behavior of the program is undefined.*

i) (4 pts) Consider variables `ai` and `g` defined on lines 10 and 11, respectively. For each of these variables, list the line numbers that contain conflicting evaluations.

a) `ai`

Conflicting evaluations of `ai` occur on lines 21, 31, and 50

b) `g`

Conflicting evaluations of `g` occur on lines 18, 21, 29, 42, and 49

ii) (7 pts) For each variable that has conflicting evaluations, determine whether they constitute a data race. If not, define the happens-before relationship between each conflicting evaluation. You may use short-hand notation to refer to the type of happens-before relationship, i.e., *(line number 1) – (type of HB relationship) → (line number 2) ...*

a) `ai`

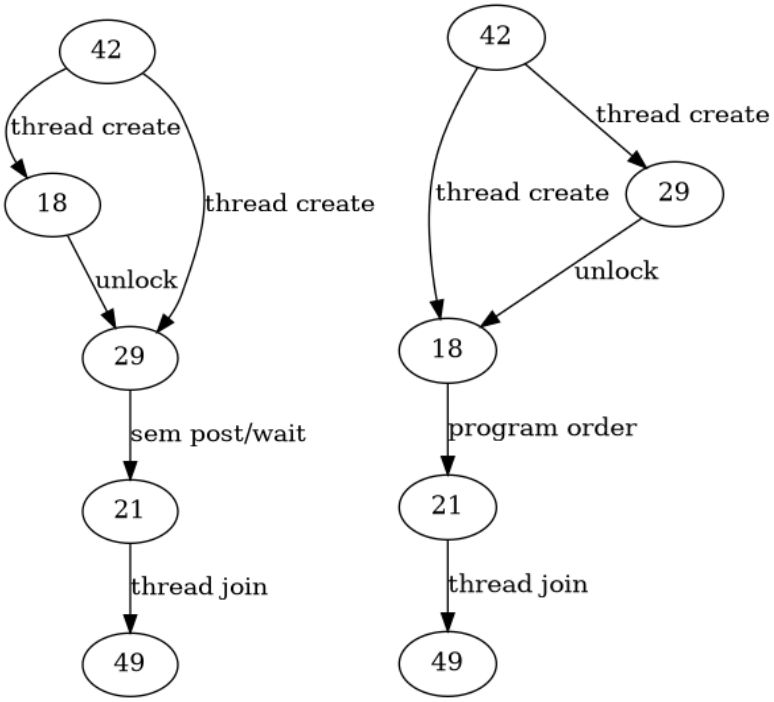
There is no data race on `ai`.
31 – (via semaphore post/wait) → 21 – (via thread join) → 50

b) `g`

There is no data race on `g`.
42 – (via thread create) → 18
42 – (via thread create) → 29
18 – (via unlock) → 29 or 29 – (via unlock) → 18

29 – (via semaphore post/wait) → 21
18 – (program order) → 21
21 – (via thread join) → 49

The following two diagrams show the possible HB relationships, depending on which thread acquires the mutex first:



Graders: although there is a total of 9 possible HB relationships for parts a) and b), assign full points (7 pts) if students correctly list at least 6 in total for parts a) and b).

iii) (5 pts) What are the possible outputs of this program?

Possible outputs of this program are `42 83` and `42 82` depending on whether the first or the second thread acquires the mutex `lock` first. (In the first case, `ai` will be 41, in the second case, `ai` will be 40 on line 21. `g` will always be 42.)

3 The Monitor Pattern (22 pts)

Monitors are supported (and mandatory) in some languages, but unfortunately, in the C/POSIX world, monitor semantics must be achieved by careful and correct use of its constituents.

Consider the following variations on a program that uses 2 threads, one of which computes a random number which the other then prints. All variations use the following main function and include appropriate header files. All variations compile successfully without warnings.

```
1  int main() {
2      // start first and second thread in random order
3      pthread_t t[2];
4      void * (*func [2])(void*) = { first_thread, second_thread };
5      srand(getpid());
6      if (rand() % 2) {
7          func[0] = second_thread;
8          func[1] = first_thread;
9      }
10
11     for (int i = 0; i < 2; i++)
12         pthread_create(&t[i], NULL, func[i], NULL);
13
14     for (int i = 0; i < 2; i++)
15         pthread_join(t[i], NULL);
16     return 0;
17 }
```

a) Consider the following variation:

```
1  int coin_flip;          // 0 or 1, heads or tails
2  bool coin_flip_done;    // true if 'coin_flip' contains valid result
3  // protects coin_flip_done;
4  pthread_mutex_t coin_flip_done_lock = PTHREAD_MUTEX_INITIALIZER;
5
6  static void *
7  first_thread(void *_unused)
8  {
9      pthread_mutex_lock(&coin_flip_done_lock);
10     while (!coin_flip_done)
11         continue;
12
13     pthread_mutex_unlock(&coin_flip_done_lock);
14     printf("Coin flip result was %s\n", coin_flip ? "HEADS" : "TAILS");
15     return NULL;
16 }
17
18 static void *
19 second_thread(void *_unused)
20 {
21     coin_flip = rand() % 2;
22
23     pthread_mutex_lock(&coin_flip_done_lock);
24     coin_flip_done = true;
25     pthread_mutex_unlock(&coin_flip_done_lock);
26     return NULL;
27 }
```

i) (2 pts) Describe which part of the monitor pattern is missing!

A condition variable is missing

ii) (2 pts) What are the possible outputs of this program? If you believe the program may or will not output anything, include this as an option.

The program may output HEADS, TAILS, or it may deadlock. (Deadlock occurs if `first_thread` happens to acquire the lock first, preventing `second_thread` from entering the critical section.)

b) Consider the following variation:

```
1  int coin_flip;          // 0 or 1, heads or tails
2  _Atomic bool coin_flip_done;    // true if 'coin_flip' contains valid result
3  pthread_cond_t coin_flip_done_cond = PTHREAD_COND_INITIALIZER;
4
5  static void *
6  first_thread(void *_unused)
7  {
8      pthread_mutex_t coin_lock = PTHREAD_MUTEX_INITIALIZER;
9      pthread_mutex_lock(&coin_lock);
10     while (!coin_flip_done)
11         pthread_cond_wait(&coin_flip_done_cond, &coin_lock);
12
13     pthread_mutex_unlock(&coin_lock);
14     printf("Coin flip result was %s\n", coin_flip ? "HEADS" : "TAILS");
15     return NULL;
16 }
17
18 static void *
19 second_thread(void *_unused)
20 {
21     pthread_mutex_t coin_lock = PTHREAD_MUTEX_INITIALIZER;
22     pthread_mutex_lock(&coin_lock);
23     coin_flip = rand() % 2;
24     coin_flip_done = true;
25     pthread_cond_signal(&coin_flip_done_cond);
26     pthread_mutex_unlock(&coin_lock);
27     return NULL;
28 }
```

i) (2 pts) Describe which part of the monitor pattern is missing!

The shared mutex (lock) is missing that protects entry to the monitor.
Note that although it appears that each thread acquires a lock and passes it to the call to condition wait, it is not the same lock. Each thread creates its own lock, which is completely ineffective. (If `coin_flip_done` weren't atomic, this would result in undefined behavior.)

ii) (2 pts) What are the possible outputs of this program? If you believe the program may or will not output anything, include this as an option.

The program may output HEADS, TAILS, or it may deadlock due to a lost wakeup. (Deadlock occurs since the checking of the state predicate is not atomic with respect to the call to condition wait, which can lead to a "lost wakeup" situation where the first thread see `coin_flip_done` false, commits to waiting, but the second thread has signaled before it could reach the call to condition wait.)

c) Consider the following variation:

```
int coin_flip;          // 0 or 1, heads or tails
pthread_mutex_t coin_flip_done_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t coin_flip_done_cond = PTHREAD_COND_INITIALIZER;

static void *
first_thread(void *_unused)
{
    pthread_mutex_lock(&coin_flip_done_lock);
    pthread_cond_wait(&coin_flip_done_cond, &coin_flip_done_lock);
    pthread_mutex_unlock(&coin_flip_done_lock);
    printf("Coin flip result was %s\n", coin_flip ? "HEADS" : "TAILS");
    return NULL;
}

static void *
second_thread(void *_unused)
{
    pthread_mutex_lock(&coin_flip_done_lock);
    coin_flip = rand() % 2;
    pthread_cond_signal(&coin_flip_done_cond);
    pthread_mutex_unlock(&coin_flip_done_lock);
    return NULL;
}
```

i) (2 pts) Describe which part of the monitor pattern is missing!

The while loop checking the state predicate is missing.

ii) (2 pts) What are the possible outputs of this program? If you believe the program may or will not output anything, include this as an option.

The program may output HEADS, TAILS, or it may deadlock if the second thread happens to acquire the monitor lock first and signals before the first thread calls condition wait.

d) (10 pts) Finally, write a correct version of this program using the following provided definitions and statements. You may not need all of them.

```
// use these statements, refer to them by number
_Atomic bool coin_flip_done;
bool coin_flip_done;
pthread_mutex_t coin_flip_done_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t coin_flip_done_cond = PTHREAD_COND_INITIALIZER;
while (!coin_flip_done)
if (!coin_flip_done)
continue
coin_flip_done = true;
pthread_cond_signal(&coin_flip_done_cond);
pthread_cond_wait(&coin_flip_done_cond, &coin_flip_done_lock);
pthread_mutex_lock(&coin_flip_done_lock);
pthread_mutex_unlock(&coin_flip_done_lock);
```

Complete the program by writing in the numbers of the definitions and statements in the correct order and correct place:

```
// add necessary definitions here
int coin_flip; // 0 or 1

bool coin_flip_done;                                    // (3)
```

```
5  pthread_mutex_t coin_flip_done_lock = PTHREAD_MUTEX_INITIALIZER;  // (4)
6  pthread_cond_t coin_flip_done_cond = PTHREAD_COND_INITIALIZER;    // (5)
7
8  static void *
9  first_thread(void *_unused)
10 {
11     pthread_mutex_lock(&coin_flip_done_lock);                     // (12)
12     while (!coin_flip_done)                                       // (6)
13         pthread_cond_wait(&coin_flip_done_cond, &coin_flip_done_lock); // (11)
14
15     pthread_mutex_unlock(&coin_flip_done_lock);                   // (13)
16     printf("Coin flip result was %s\n",
17            coin_flip ? "HEADS" : "TAILS");
18     return NULL;
19 }
20
21 static void *
22 second_thread(void *_unused)
23 {
24     coin_flip = rand() % 2;
25
26     pthread_mutex_lock(&coin_flip_done_lock);                     // (12)
27     coin_flip_done = true;                                        // (9)
28     pthread_cond_signal(&coin_flip_done_cond);                    // (10)
29     pthread_mutex_unlock(&coin_flip_done_lock);                   // (13)
30
31     return NULL;
32 }
```

See numbers above. The second thread could also acquire the lock before assigning `coin_flip`, and it would also be ok to signal without the lock (switch (10) and (13)). Either the code or the numbers needed to be provided. We also accepted (2) instead of (3), even though the use of an `_Atomic` is unnecessary and non-idiomatic, but note that using an atomic does not remove the need to lock for the second thread.

4 MPSC (12 pts)

Consider the following program written using the multi-producer, single-consumer channel you implemented in exercise 3.

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

#include "mpsc.h"

struct txrx {
    struct mpsc_sender *tx1;
    struct mpsc_sender *tx2;
    struct mpsc_receiver *rx;
    char id;
};

static void *
thread_A(void *_txrx)
{
    struct txrx *p = _txrx;
    for (int i = 0; i < 3; i++)
        mpsc_send(p->tx1, i);
    mpsc_drop_sender(p->tx1);
    return NULL;
}

static void *
thread_B(void *_txrx)
{
    struct txrx *p = _txrx;
    uintptr_t a;
    while (mpsc_recv(p->rx, &a) == RECV_SUCCESS)
    {
        mpsc_send(p->tx1, a);
        mpsc_send(p->tx2, a);
    }
    mpsc_drop_sender(p->tx1);
    mpsc_drop_sender(p->tx2);
    return NULL;
}

static void *
thread_C(void *_txrx)
{
    struct txrx *p = _txrx;
    uintptr_t rval;
    while (mpsc_recv(p->rx, &rval) == RECV_SUCCESS)
        printf("%c%ld ", p->id, rval);

    return NULL;
}
```

```c
int main()
{
    struct mpsc_channel ch[3];
    for (int i = 0; i < 3; i++)
        mpsc_sync_channel(/* CHANNEL_SIZE= */10,
                          ch + i);

    #define N 4
    pthread_t threads[N];
    int i = 0;
    struct txrx p1 = {
        .tx1 = ch[0].sender,
    };
    pthread_create(&threads[i++], NULL,
                   thread_A, &p1);
    struct txrx p3 = {
        .tx1 = ch[1].sender,
        .tx2 = ch[2].sender,
        .rx = ch[0].receiver,
    };
    pthread_create(&threads[i++], NULL,
                   thread_B, &p3);
    struct txrx p4 = {
        .rx = ch[1].receiver,
        .id = 'A',
    };
    pthread_create(&threads[i++], NULL,
                   thread_C, &p4);
    struct txrx p5 = {
        .rx = ch[2].receiver,
        .id = 'B',
    };
    pthread_create(&threads[i++], NULL,
                   thread_C, &p5);

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

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

Describe and count all possible outputs of this program (you may assume that each `printf` statement is atomic).

Thread A sends integers 0, 1, and 2 to thread B, in this order. Thread B sends 0, 1, and 2 to two instances of thread C, which will output `A0`, `A1`, `A2`, `B0`, `B1`, and `B2` such that `A0 A1 A2` and `B0 B1 B2` remain in order. Thus $\binom{6}{3} = 20$ different outputs are possible, such as `A0 B0 A1 A2 B1 B2` but not `A0 B1 A1 B0 A2 B2`.

5 Deadlock and Performance (18 pts)

5.1 Deadlock (8 pts)

Consider the implementations of your p2 thread pool in stages 3 and 5. Discuss whether resource-related deadlock could occur in your implementation. If so, explain why. If not, explain how you prevented it.

a) (4 pts) In stage 3 (work sharing, single lock, with helping implemented), resource-related deadlock
☐ can / ☑ cannot occur, because

there can be no hold-and-wait with a single lock, removing one of the necessary Coffman conditions.

b) (4 pts) In stage 5 (work stealing, single lock broken into multiple, helping implemented), resource-related deadlock
☐ can / ☑ cannot occur, because

only one lock is held at any time, again avoiding hold-and-wait.
This is true, at least, if you implemented the suggested design. If you didn't you will need to justify here how your design prevents deadlock, such as by enforcing a consistent locking order that avoid circular waits.

5.2 Locking Performance (10 pts)

Fill in the blanks in the following paragraph. If multiple word choices meaningfully complete the paragraph, we'll accept any of them.

Mutexes are synchronization devices that are necessary to ensure correctness in multi-threaded programs. Without their proper use, programs could suffer from multiple concurrency-related defects, including data races and atomicity violations.

However, mutexes incur costs, which can be roughly broken into direct and indirect costs. The indirect cost include the loss of parallelism due to the serialization locks induce. Such serialization can lead to low CPU utilization, particularly when threads spend a relatively large portion of their execution inside critical sections protected by a lock.

To reduce this cost, programmers will redesign their data structures such that locks can be broken into multiple, separate locks, each protecting only a subset of the shared data. However, this process can be difficult, because doing so can increase the risk of deadlocks and atomicity violations.

In addition, using more locks may also increase the direct costs of locking. At a minimum, acquiring an uncontended lock requires an atomic instruction which is about 17× more expensive than a L1 cache hit. If the lock is contended, the calling thread must then enter the kernel to be placed into the `BLOCKED` state. Typically, two context switches are required until the thread has returned and can continue.