CS 3214 Fall 2022 Midterm Solutions

October 31, 2022

Contents

1 Operating Systems (10 pts)

1.1 Dual-Mode Operation (5 pts)

Which of the following are operations that processes executing in user mode are allowed to do?

i) Executing privileged machine instructions.
☐ may do / ☑ may not do.

See `hlt` demo.

ii) Making system calls in order to access kernel services.
☑ may do / ☐ may not do.

It's in fact the only way to access services such as I/O.

iii) Issuing machine I/O instructions to program the network interface controller (NIC) when they need to communicate over the network.
☐ may do / ☑ may not do.

An exception here may be so-called kernel bypass solutions that for performance reasons provide direct device access for trusted applications. If you knew about those, we would have expected you to mention them. The general case here is no: user processes do not have direct access to devices - the OS protects and abstracts such access, for instance through sockets.

iv) Executing infinite loops.
☑ may do / ☐ may not do.

v) Replacing their running program with a new one (via a system call).
☑ may do / ☐ may not do.

e.g. `exec()`

1.2 Process States (5 pts)

Which of the following statements regarding the simplified process state model are true?

i) A computer's power consumption is strongly correlated to the number of processes that are in the `RUNNING` state.
☑ true / ☐ false

Idle CPUs consume far less power than those executing code.

ii) The OS migrates `BLOCKED` processes from overloaded CPUs to less utilized CPUs to allow them to be scheduled there.
☐ true / ☑ false

`BLOCKED` processes cannot make use of a CPU, so migrating them to another will not do them any good.

iii) Most processes executing on a typical portable computer such as an Android or iPhone are in the `BLOCKED` state.
☑ true / ☐ false

Otherwise, the battery would be drained extremely fast.

iv) Processes move into the `READY` state when they are ready to commence an I/O operation.
☐ true / ☑ false

No, they move into the `READY` state when they are ready to run on a CPU, e.g., after I/O operations have completed.

v) A shell such as `bash` or `cush` spends most of its time in the `BLOCKED` state.
☑ true / ☐ false

The shell is either waiting for user input or waiting for a foreground process to terminate.

2 Unix Processes and IPC (30 pts)

2.1 Creating Processes (7 pts)

Consider the following Unix program, which we will name `forktest`:

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

int
main()
{
    if (fork() == 0) {
        printf("child process\n");
    } else {
        printf("parent process\n");
    }
}
```

a) (3 pts)

Which are possible outputs of this program? Assume that the user executing this program is not currently executing many processes on the same machine.

☐ i) Only

```
child process
parent process
```

☐ ii) Only

```
parent process
child process
```

☑ iii) Either of i) or ii). Parent and child process may execute in any order

☐ iv) One of

```
parent process
```

or

```
child process
```

(depending on whether the process was a child or a parent process).

b) (2 pts) Now suppose the user is about to run this program on a machine where the same user is already executing $n-1$ processes, where n is the user's process (`nproc`) limit. For instance, on rlogin, for undergraduate students, the `nproc` limit is set to $1,500$ processes a student may simultaneously run.

What would happen if the user typed in `./forktest` to run this program at the shell prompt?

The user would be able to start this program in a new process, but the attempt at forking another child process would fail. (The output would then be just `parent process`)

c) (2 pts) What software engineering rule when it comes to writing systems programs is violated in this example?

The return values of system calls must always be checked. Here, the case where `fork()` returned -1 due to exhausting the user's process limit was ignored.

2.2 `posix_spawn` (6 pts)

In Project 1, we recommended the use of the `posix_spawn` function as opposed to using `fork()` directly. `posix_spawn` is a function that internally combines the `fork()` and `exec()` system calls. Consider the following program named `spawn.c`:

```
1  #include <spawn.h>
2  #include <string.h>
3  #include <stdio.h>
4
5  int global;
6  int
7  main(int ac, char *av[])
8  {
9      int local = 44;
10     if (ac == 1) {
11         global = 42;
12         local++;
13         pid_t child;
14         char * argv[] = { av[0], "-child", NULL };
15         posix_spawnp(&child, av[0], NULL, NULL, argv, NULL);
16     } else {
17         if (!strcmp(av[1], "-child")) {
18             printf("global = %d local = %d\n", global, local);
19         }
20     }
21 }
```

This program is compiled to an executable called `spawn`.
When Dr. Back ran this program at a shell prompt, he obtained the somewhat weird-looking output

```
[gback@holly midtermf22]$ ./spawn
[gback@holly midtermf22]$ global = A local = B
```

where A and B are two integer numbers output on line 18.

a) (2 pts) What are the values of A and B?

A = 0, B = 44

b) (2 pts) Why does the second line of the terminal output read

```
[gback@holly midtermf22]$ global = A local = B
```

This program does not wait for the child process it spawns, therefore, the shell may return to the prompt before the child process has a chance to perform its I/O.

c) (2 pts) Could Dr. Back have seen a different output on his terminal? If so, what would that output have looked like?

Yes - had the child run faster, the output would have been:

```
global = A local = B
[gback@holly midtermf22]$
```

2.3 System Calls and I/O (10 pts)

a) (2 pts) The following Rust program is compiled and run on a Linux machine:

```
fn main() {
    println!("Hello World!");
}
```

The output `Hello World!` appears on the user's terminal.

Name 2 (different) system calls this program will definitely make. Only consider system calls that are part of the executable the Rust compiler will create, or a shared library loaded by it – i.e., if you ran `strace` on this program, consider only calls after the `exec()` system call that starts the program. Do not consider any system calls made by the shell when it launches the program.

Call 1: `write(1)` *Call 2:* `exit(1)`

Note that the program will not execute any `open(1)` or similar calls.

b) (8 pts) Consider the following Python 3 program:

```
import sys
print ("error", file=sys.stderr)
```

When running this command in a pipeline with the `rev` command like so:

```
$ python3 message.py | rev
error
```

we see it output `error`. The `rev(1)` command reverses the characters on each line of its standard input stream and outputs the result to standard output.

i) (2 pts) Why was the output of the Python script not reversed?

Because the Python script outputs its message to its standard error stream, but the shell sets up the pipe to forward its standard output stream to `rev`

ii) (6 pts) Using a utility `so2se`, we see

```
$ ./so2se python3 message.py | rev
rorre
```

Implement `so2se` in C. Your implementation should work for any program (not just python3 and message.py) and it should make sure that whatever the program outputs to its standard error stream will be received by the `rev` program. Error checking is not required. The program can be implemented in less than 8 lines of well-formatted code.

Write your code here:

```
#include <unistd.h>
int
main(int ac, char *av[])
{
    dup2(STDOUT_FILENO, STDERR_FILENO);
    execvp(av[1], av+1);
}
```

(We gave full credit even if you mixed up the order of `STDOUT_FILENO` and `STDERR_FILENO` in the `dup2()` call.)

Alternate solution using `posix_spawn`:

```
#include <unistd.h>
#include <spawn.h>
#include <sys/wait.h>

int
main(int ac, char *av[])
{
    posix_spawn_file_actions_t file_actions;
    posix_spawn_file_actions_init(&file_actions);
    posix_spawn_file_actions_adddup2(&file_actions,
        STDOUT_FILENO, STDERR_FILENO);

    pid_t child;
    posix_spawnp(&child, av[1], &file_actions, NULL, av+1, NULL);
    waitpid(child, NULL, 0);
}
```

Note that solutions that attempt to use an additional pipe in addition to the one created by the shell would then be responsible for forwarding the data through this pipe, which is unnecessary and inefficient:

```
#define _GNU_SOURCE 1
#include <unistd.h>
#include <spawn.h>
#include <fcntl.h>
#include <sys/wait.h>

const int WRITE_END = 1;
const int READ_END = 0;

int
main(int ac, char *av[])
{
```

```
        int fd[2];
        pipe2(fd, O_CLOEXEC);
        posix_spawn_file_actions_t file_actions;
        posix_spawn_file_actions_init(&file_actions);
        posix_spawn_file_actions_adddup2(&file_actions,
            fd[WRITE_END], STDOUT_FILENO);
        posix_spawn_file_actions_adddup2(&file_actions,
            fd[WRITE_END], STDERR_FILENO);

        pid_t child;
        posix_spawnp(&child, av[1], &file_actions, NULL, av+1, NULL);
        int bread = 0;
        char buf[8192];
        close(fd[WRITE_END]);
        while ((bread = read(fd[READ_END], buf, sizeof buf)) > 0)
            write(STDOUT_FILENO, buf, bread);

        waitpid(child, NULL, 0);
    }
```

2.4 Know Your Shell (7 pts)

Translate the following three statements from English into a shell command and/or keystrokes. You may use valid syntax of any widely used shell, including `cush`.

a) (3 pts) Run the gcc compiler with the `-v` flag on file `bad.c`, then search what it outputs to standard output or standard error for the word `error`. Redirect all lines containing this word into a file called `errors.txt`.

```
gcc -v bad.c |& grep error > errors.txt
```

b) (2 pts) Start `nginx` in the background, redirecting its standard output stream to `/tmp/nginx.log`

```
nginx > /tmp/nginx.log &
```

c) (2 pts) Bring background job #3 into the foreground, then attempt to terminate it. (Assume your shell currently has a job using job id #3 running.)

```
fg 3
^C
```

(This is cush syntax, bash would be `fg %3` instead.)

3 Multithreading (36 pts)

3.1 Thread Support in ISO-C (9 pts)

Consider the following multi-threaded program which utilizes ISO-C's multi-threading facilities, which are similar to the similarly named POSIX Thread facilities.

```
1  #include <threads.h>
2  #include <stdio.h>
3
4  int global = 3;
5
6  int thread_fun(void *_arg) {
7      global++;
8      return 0;
9  }
10
11 int
12 main()
13 {
14     thrd_t t1, t2;
15     thrd_create(&t1, thread_fun, NULL);
16     thrd_create(&t2, thread_fun, NULL);
17     thrd_join(t1, NULL);
18     thrd_join(t2, NULL);
19     printf("global %d\n", global);
20     return 0;
21 }
```

a) (3 pts) When compiled and run using a compiler and library that *strictly* implements the ISO C17 standard, what are the legal outputs[1] of this program, and why?

This problem contains a data race on line 7 where two threads perform read and write accesses to a shared variable. Therefore, ISO C specifies that this program exhibits undefined behavior. This means that any and all output is legal - the ISO C specification imposes "no requirements." The program could not compile, could crash, or it could output 3, 4, 5, 6, or "pineapple," for instance.

b) (3 pts) What are the legal outputs of this program if you switched lines 16 and 17 so that the program would read:

```
15     thrd_create(&t1, thread_fun, NULL);
16     thrd_join(t1, NULL);
17     thrd_create(&t2, thread_fun, NULL);
18     thrd_join(t2, NULL);
```

The only legal output is 5. This program has no data race since starting thread 2 after joining thread 1 introduces a happens-before relationship which removes the data race since the accesses to `global` are no longer concurrent.

c) (3 pts) What are the legal outputs of this program if line 4 is replaced with:

```
4  _Atomic int global = 3;
```

(Assume that lines 16 and 17 occur in their original order.)

The only legal output is 5. The compiler will ensure that the increment operation on `global` is compiled to an atomic increment instruction. Such modifications of atomic variables are by definition not data races.

[1] "Legal" here means output that can be produced while complying with the standard.

3.2 Condition Variables (10 pts)

Consider the following program, which is a simulation of transfers of money between bank accounts executed by multiple threads.

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  struct account {
6      pthread_mutex_t mutex;  // protects balance
7      pthread_cond_t cond;    // signals transfers
8      int balance;            // current balance
9  };
10
11 struct thread_info {
12     struct account *from;   // account from which to transfer
13     struct account *to;     // account to which to transfer
14     int n;                  // number of transfers
15     int amount;             // amount of each transfer
16 };
17
18 static void *
19 transfer_thread(void *_arg)
20 {
21     struct thread_info *info = _arg;
22     for (int i = 0; i < info->n; i++) {
23         pthread_mutex_lock(&info->from->mutex);
24         pthread_mutex_lock(&info->to->mutex);
25
26         if (info->from->balance >= info->amount) {
27             info->from->balance -= info->amount;
28             info->to->balance += info->amount;
29         }
30         pthread_mutex_unlock(&info->to->mutex);
31         pthread_mutex_unlock(&info->from->mutex);
32     }
33     free(info);
34     return 0;
35 }
36
37 static pthread_t
38 start_transfer_thread(struct account *from,
39                       struct account *to,
40                       int n, int amount)
41 {
42     struct thread_info * info = malloc(sizeof(*info));
43     info->from = from; info->to = to;
44     info->n = n; info->amount = amount;
45     pthread_t t;
46     pthread_create(&t, NULL, transfer_thread, info);
47     return t;
48 }
49
50 int
51 main()
52 {
```

```
53 #define ACCOUNT(name, initialbalance) \
54     struct account name = { \
55         .balance = initialbalance, \
56         .mutex = PTHREAD_MUTEX_INITIALIZER, \
57         .cond = PTHREAD_COND_INITIALIZER \
58     }
59     ACCOUNT(alice, 0);
60     ACCOUNT(bob, 150000);
61     ACCOUNT(carol, 50000);
62     ACCOUNT(dave, 0);
63
64     pthread_t t[3];
65     t[0] = start_transfer_thread(&bob, &alice, 10000, 15);
66     t[1] = start_transfer_thread(&carol, &alice, 10000, 5);
67     t[2] = start_transfer_thread(&alice, &dave, 10000, 20);
68     for (int i = 0; i < 3; i++)
69         pthread_join(t[i], (void **) NULL);
70
71     printf("balances %d %d %d %d\n",
72             alice.balance, bob.balance,
73             carol.balance, dave.balance);
74
75     return 0;
76 }
```

This program can deadlock when 2 threads simultaneously attempt to transfer money from account A to account B and vice versa. Note that this question is not about deadlock, which cannot occur because Bob and Carol transfer money to Alice, which Alice transfers to Dave.

Instead, this question is about ensuring that no one overdraws their account: money should only be transferred from accounts if a nonnegative balance remains after the transfer – this is checked on line 26. Unfortunately, as presented, this code doesn't work. It should output

```
balances 0 0 0 200000
```

because Bob makes 10,000 transfers of $15 to Alice, Carol makes 10,000 transfers of $5 to Alice, and Alice makes 10,000 transfers of $20 each to Dave who should thus end up with 200,000. But the actual output varies, e.g., it could be:

```
balances 13100 0 0 186900
```

Fix this program using the monitor pattern and condition variables. A condition variable was already introduced (see line 7) and initialized (line 57), but it is not yet used in the required way.

Outline which statements you would need to add, and where. Use line numbers. Hint: you will need to insert one or multiple statements in 2 locations.

☐ After line 23 add

```
while (info->from->balance < info->amount)
   pthread_cond_wait(&info->from->cond, &info->from->mutex);
```

(-1 for adding this after line 24 or 25, which would lock the 'to' account while waiting for a transfer.)

☐ After line 28, add

```
pthread_cond_broadcast(&info->to->cond);
```

(-1 for using `pthread_cond_signal` here, which wouldn't signal if 2 transactions could continue with the transferred money.)

3.3 Deadlock and Semaphores (11 pts)

A developer debugs a multithreaded program that appears to have gotten stuck. Below is a protocol of their gdb session:

```
$ gdb deadlocksem
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-19.el8
(...)
(gdb) run
Starting program: /home/courses/cs3214/midtermf22/deadlock/deadlocksem
(...)
[New Thread 0x155554d3f700 (LWP 3105337)]
[New Thread 0x155554b3e700 (LWP 3105338)]
^C
Thread 1 "deadlocksem" received signal SIGINT, Interrupt.
0x000015555510f6cd in __pthread_timedjoin_ex () from /lib64/libpthread.so.0
(gdb) thread apply all backtrace

Thread 3 (Thread 0x155554b3e700 (LWP 3105338)):
#0  0x0000155555116da6 in do_futex_wait.constprop () from /lib64/libpthread.so.0
#1  0x0000155555116e98 in __new_sem_wait_slow.constprop.0 () from /lib64/libpthread.so.0
#2  0x00000000004007a4 in thread2 (_arg=0x0) at deadlocksem.c:21
#3  0x000015555510e1ca in start_thread () from /lib64/libpthread.so.0
#4  0x0000155554d79e73 in clone () from /lib64/libc.so.6

Thread 2 (Thread 0x155554d3f700 (LWP 3105337)):
#0  0x000015555511782d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x0000155555110ad9 in pthread_mutex_lock () from /lib64/libpthread.so.0
#2  0x000000000040077d in thread1 (_arg=0x0) at deadlocksem.c:13
#3  0x000015555510e1ca in start_thread () from /lib64/libpthread.so.0
#4  0x0000155554d79e73 in clone () from /lib64/libc.so.6

Thread 1 (Thread 0x155555538740 (LWP 3105333)):
#0  0x000015555510f6cd in __pthread_timedjoin_ex () from /lib64/libpthread.so.0
#1  0x000000000040080e in main () at deadlocksem.c:34
(gdb) thread 2
[Switching to thread 2 (Thread 0x155554d3f700 (LWP 3105337))]
#0  0x000015555511782d in __lll_lock_wait () from /lib64/libpthread.so.0
(gdb) frame 2
#2  0x000000000040077d in thread1 (_arg=0x0) at deadlocksem.c:13
13            pthread_mutex_lock(&lock);
(gdb) p lock
$1 = {__data = {__lock = 2, __count = 0, __owner = 3105338, __nusers = 1, ... }
```

a) (3 pts) What states (in the sense of the simplified process/thread state diagram) are the three threads of this process in?

Thread 1: `BLOCKED` Thread 2: `BLOCKED` Thread 3: `BLOCKED`

b) (8 pts) The program they were running is shown below. One function (`thread2`) is incomplete. Complete it so that it is consistent with what you can infer from the debugging session:

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

sem_t S;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

static void *thread1 (void *_arg)
{
    struct timespec ts = { .tv_sec = 1, .tv_nsec = 0 };
    nanosleep(&ts, NULL);
    pthread_mutex_lock(&lock);
    return NULL;
}

static void *thread2 (void *_arg)
{
    /* Complete this function */
    pthread_mutex_lock(&lock);        // add this
    sem_wait(&S);                     // and this
    return NULL;
}

int
main()
{
    pthread_t t1, t2;
    sem_init(&S, 0, 0);
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, (void **) NULL);
    pthread_join(t2, (void **) NULL);
}
```

3.4 Facts About Threading (6 pts)

Which of the following statements about multithreaded programming are correct?

i) Different kernel-level (aka kernel-supported) threads can be executed on different hardware threads (aka hyperthreads, or SMT) at the same time.
☑ true / ☐ false

Hardware threads appear as independent processors to the OS kernel, and therefore it will schedule kernel-level threads on them.

ii) If a C11 program is free of data races and doesn't use atomics, then the execution of this program will be sequentially consistent.
☑ true / ☐ false

Ensuring sequential consistency was the motivation for demanding data race freedom. In fact, sequential consistency is also ensure for most accesses to atomics, except for the special operations that explicitly use a weaker semantics.

iii) In multithreading programming, the technique of breaking up locks is typically used to reduce the likelihood of deadlock.
☐ true / ☑ false

No, it increases the likelihood of deadlock since it reintroduces a necessary condition: there can now be hold-and-wait, and possibly circular wait.

iv) It is risky to invoke `malloc()` from multiple threads at the same time because the `malloc()` function manages a shared heap, which is not thread-safe.
☐ true / ☑ false

`malloc()` is obviously thread-safe, or your p2 implementation would not have worked. (The POSIX standard demands that `malloc` be implemented in a thread-safe fashion.)

v) POSIX Threads that execute inside a critical section will not be preempted while they execute inside said section.
☐ true / ☑ false

Despite the adjective "critical," a critical section means simply holding a mutex, which does turn off preemption. In fact, user processes cannot turn off preemption at all, otherwise the user could not kill infinitely looping threads or processes.

vi) Using fine-grained locking tends to reduce the loss of CPU utilization that stems from serialization, i.e., it tends to increase CPU utilization.
☑ true / ☐ false

Yes, that's in fact a primary motivation for doing so.

4 Development and Linking (24 pts)

4.1 Symbol Tables (11 pts)

Consider the following C program:

```
#include <stdlib.h>

struct string {
    char *fieldstr;
    int length;
};

static int appendcount;

void append(struct string *p1, struct string *q1)
{
    appendcount++;
    char *from = q1->fieldstr;
    char *copy = realloc(p1->fieldstr, p1->length + q1->length);
    for (int i = 0; i < q1->length; i++) {
```

```
        copy[p1->length + i] = from[i];
    }
    p1->fieldstr = copy;
    p1->length += q1->length;
}
```

When compiled with `gcc -c` what is the resulting symbol table of the .o file?

Complete the following table, listing for each identifier whether it occurs in the symbol table and if so, what the type of the symbol would be if it were listed by the `nm(1)` command. Use the single letter abbreviation used by nm.

Identifier	Occurs in symbol table	If yes, list type here
`string`	☐ yes/ ☑ no	
`fieldstr`	☐ yes/ ☑ no	
`length`	☐ yes/ ☑ no	
`appendcount`	☑ yes/ ☐ no	`b`
`append`	☑ yes/ ☐ no	`T`
`p1`	☐ yes/ ☑ no	
`q1`	☐ yes/ ☑ no	
`from`	☐ yes/ ☑ no	
`copy`	☐ yes/ ☑ no	
`realloc`	☑ yes/ ☐ no	`U`
`i`	☐ yes/ ☑ no	

All others are type names, field names, names of parameters and local variables which are resolved by the compiler.

4.2 No More Common (5 pts)

After generations of CS3214 students had to learn about common symbols and how they cause potential issues when building programs, David Brown in 2018 filed a bug report against the GNU C compiler asking that gcc stop emitting common symbols by default and provide strong definitions in all cases instead. Quoting Brown:

> "The use of a "common" section to match up tentative object definitions in different translation units in a program is contrary to the C standards (section 6.9p5 in C99 and C11, section 6.7 in C90), gives a high risk of error, may lead to poorer quality code, and provides no benefits except backwards compatibility with ancient code."

This change was implemented in 2020 in gcc 10 (which is not currently the default gcc on the rlogin machines, which is why we weren't using it this semester) and you still had to learn the old rules. Write a program consisting of 2 separately compiled files that would have compiled and linked under the old rules, but will fail to do so under the new ones!

// file1.c goes here	*// file2.c goes here*
`int c;`	`int c;`

Also possible is a strong and a weak definition, such as `int c;` and `int c = 42;`

4.3 Dealing with Linker Errors (8 pts)

a) (4 pts) A developer is trying to build a program like so:

```
$ gcc -Wmissing-prototypes -Wall main.c -o main
/tmp/ccTPr0EA.o: In function `main':
main.c:(.text+0xa): undefined reference to `important_function'
collect2: error: ld returned 1 exit status
```

but obtains the error shown above.

Which of the following are reasonable next steps to take?[2] Check all that apply!

i) Include the appropriate header file for `important_function` in `main.c`.
☐ yes / ☑ no

ii) Find and include an appropriate library on the command line.
☑ yes / ☐ no

iii) Provide a suitable definition for `important_function` in the program.
☑ yes / ☐ no

iv) Provide a forward declaration for `important_function` in `main.c`.
☐ yes / ☑ no

This is a linker error, so including a header file will not help (unless the header file defines this function, but this would not be reasonable to assume since the absence of compile errors shows that the compiler must have seen a forward declaration of that function already.) Header files also do not typically define functions, except for header-only libraries, in which case the definition would have been static, which could not have caused the linker error shown. Although theoretically possible, it is not reasonable to assume that a header file would provide a definition of that function – this would be a gross violation of best practice and would lead to multiply defined symbols as soon as this header file is included into more than one compilation unit.

This error indicates that the linker couldn't find a definition of this function, and thus the programmer needs to find one and include the appropriate library, or implement the function themselves.

b) (4 pts) The developer now tries this:

```
$ gcc -Wmissing-prototypes -Wall main.c -o main -L. -lhelp -lB
./libB.a(lib_B_file1.o): In function `important_function':
lib_B_file1.c:(.text+0xa): undefined reference to `needed_helper_function'
collect2: error: ld returned 1 exit status
```

Given this information, what are reasonable next steps to take? Check all that apply!

i) Include the appropriate header file for `needed_helper_function` in `main.c`.
☐ yes / ☑ no

ii) Switch the order of `-lhelp` and `-lB`
☑ yes / ☐ no

iii) Provide a suitable definition for `needed_helper_function` in the program and remove `-lhelp` from the command line.
☑ yes / ☐ no

iv) Provide a forward declaration for `needed_helper_function` in `main.c`.
☐ yes / ☑

[2] For the purposes of this question, assume that `important_function` and also `needed_helper_function` (used in part b) are functions whose purpose is known to the developer and which the developer could readily implement if necessary.

As in part a) including header files is not a reasonable solution because it doesn't provide a global definition of the needed helper function. The undefined reference occurs in a file of library `libB.a`. If the definition is in library `libhelp.a`, the linker would have not included it because `-lhelp` was before `-lB` on the command line, thus the function wasn't part of the referenced-but-not-yet-resolved set U at the time the linker processed the library `libhelp.a`. Therefore, only switching the order or implementing the needed function are reasonable next steps. A forward declaration in `main.c` would have made no difference whatsoever since `main.c` isn't even using the function in question.