CS 3214 Fall 2024 Midterm Solutions

October 31, 2024

Contents

1 OS Principles and Process States (30 pts)

1.1 Basic OS Principles (10 pts)

Check if the following statements are true or false.

a) Traditionally, operating system kernels are trusted control programs that have unconstrained access to all resources of a machine.

☑ true / ☐ false

b) User processes run in user mode, but control processes such as shells run in kernel mode.

☐ true / ☑ false

Shells and administrative control processes run in user mode, too.

c) In systems exploiting dual-mode operation, instructions that read from or write to the accumulator register (e.g., `$rax` on x86_64) are privileged.

☐ true / ☑ false

The accumulator is an ordinary register and not sensitive in any way.

d) On most general purpose OS, the kernel requires exact knowledge of a program's resource needs before a process can be admitted for scheduling.

☐ true / ☑ false

No, the OS will infer a program's resource needs based on its requests and behavior.

e) A limitation of most OS designs is their complete lack of fail-stop mechanisms to respond to programmer or user errors that cause programs to misbehave.

☐ true / ☑ false

No, they include multiple fail-stop mechanisms, such as the default disposition of signals such as `SIGSEGV` or `SIGPIPE`, which lead to process termination.

1.2 Multiprocessor Scheduling (4 pts)

A CS3214 student has been reading about multiprocessor scheduling and the simplified process state diagram. In order to reduce the overhead associated with scheduling and simplify the implementation, they propose the following scheme. Since they've learned that most processes in a typical system are in the `BLOCKED` state, they'll reserve 3/4 of the available processors for `BLOCKED` processes, and the remaining 1/4 of the available processors for `READY` or `RUNNING` processes. This way, scheduling is simplified because processes can be easily assigned to processors based on their state.

Critique this idea:

Since `BLOCKED` processes cannot actually use any CPU time, this idea would leave 3 quarters of the CPU capacity unused.

1.3 Understanding Process States (16 pts)

Consider the following short C program where `#include` statements were elided for brevity:

```
1  static int input()
2  {
3      pid_t child = fork();
4      if (child == 0) { // child process
5          int n;
6          scanf("%d", &n);
7          exit(n);
8      } else {
9          int status;
10         waitpid(child, &status, 0);
11         return WEXITSTATUS(status);
12     }
13 }
14
15 int main(int ac, char *av[])
16 {
17     printf("%d\n", input());
18 }
```

a) (4 pts) Suppose the program is saved in a file `childio.c` and compiled to an executable called `childio`. What is the output when a user types

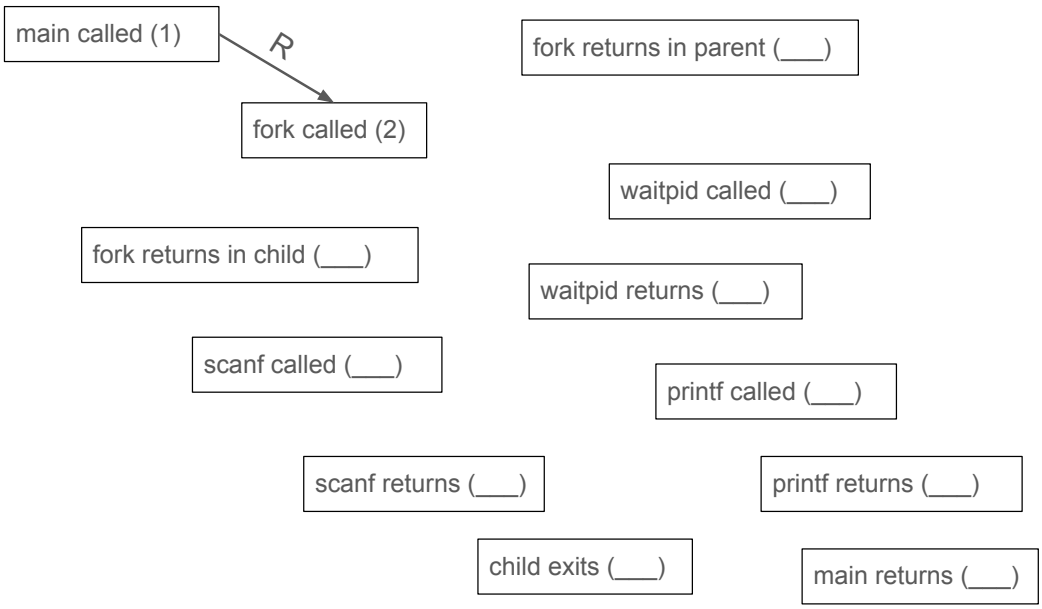
```
echo 42 | ./childio
```

The output is 42

b) (12 pts) Now consider what happens if the user runs the program like so

```
./childio
21
```

Here, the user first types `./childio` and then types `21` followed by the enter key, both at human speed. Below is an incomplete chart that displays some of the events that occur when this program is executed:

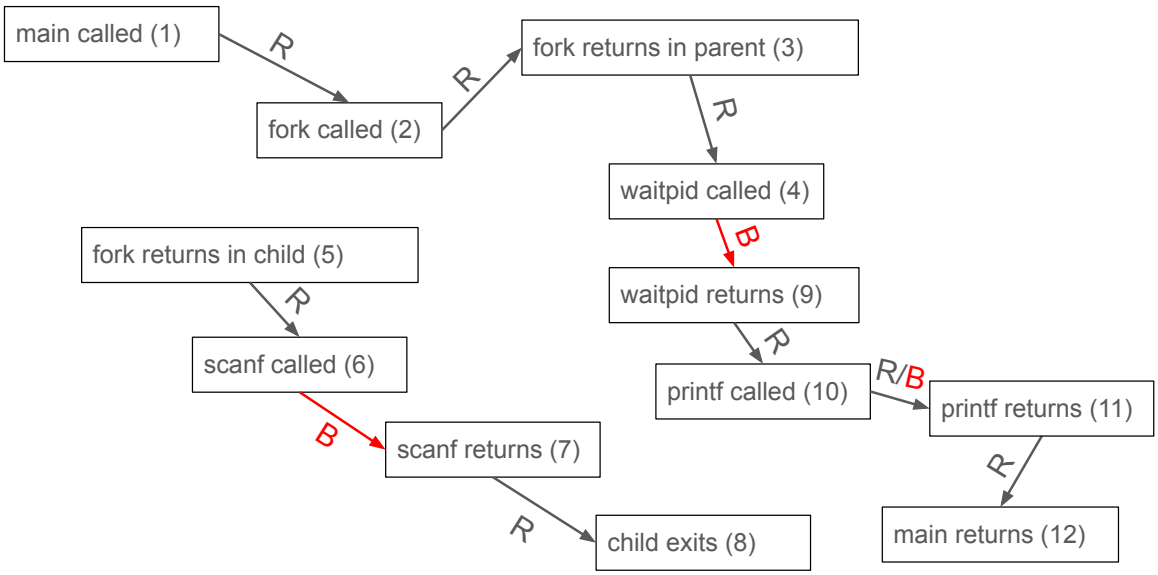


Complete the chart as follows:

- Connect blocks that execute in the context of one process with an edge, in order. Label each edge with an R if the process is in the `RUNNING` state between the events. Label the edge with a B if the process is in the `BLOCKED` state during the transition.
- Assuming that the system executing this program has only a single logical CPU, map each event to an integer number that represents a possible ordering in which a scheduler may schedule these events. Enter that number in the provided space for each block `(___)`.

The first edge between "main called (1)" and "fork called (2)" is already filled in for you. The event "main called (1)" occurs first, and "fork called (2)" occurs second, and the process executing these events is in the `RUNNING` state between these events.

A possible answer is shown below



Because the problem stipulated that input occurred at human speed, we know that scanf will actually block (as opposed to returning already input data). Similarly, waitpid will also block.

The numbering shows one possible schedule. Any schedule is accepted that meets the following conditions:

- The ordering is consistent with each process's time line (that is, events that occur later in one process must also occur later in the schedule)
- "fork returns in child" occurs after "fork called"
- "child exits" occurs before "waitpid returns"
- No two events occur at the same time (the problem assumed a single processor).
- The edge between "printf called" and "printf returns" can be labeled either R or B, depending on the nature of the terminal and what the kernel needs to do to output a character there (for instance, it may block if this involves taking kernel-internal locks or semaphores, or even when it involves scrolling a terminal).

We didn't deduct if you connected "fork called" and "fork returns in child" with an edge.

2 Of Pipes and Shells (16 pts)

2.1 IPC via Pipes (10 pts)

Consider the following C program which makes use of pipes to perform interprocess communication:

```
1  const int READ_END = 0;
2  const int WRITE_END = 1;
3
4  int main(int ac, char *av[])
5  {
6      int p1[2], p2[2];
7      pipe(p1);
8      pipe(p2);
```

```
9       char *msg = "3214";
10      char a, c = 'Z';
11      for (char *p = msg; *p; p++) {
12          if (fork() == 0) {
13              read(p1[READ_END], &a, 1);
14              write(STDOUT_FILENO, &a, 1);
15              write(p2[WRITE_END], &c, 1);
16              return 0;
17          } else {
18              write(p1[WRITE_END], p, 1);
19              read(p2[READ_END], &a, 1);
20              write(STDOUT_FILENO, &a, 1);
21          }
22      }
23  }
```

Answer the following questions:

a) (6 pts) Provide all possible outputs of this program. You may assume that no system call fails due to resource exhaustion. (Note that the output of this program may or may not be unique due to the multitude of processes involved.)

> This program creates 2 pipes and forks 4 child processes. The parent sends one character from the sequence `3214` to each child process via pipe 1, then reads a character from pipe 2. Each child reads a character from pipe 1 and outputs it to standard output before sending back 'Z' via pipe 2. The parent reads this character from pipe 2 (and outputs it to standard output) before continuing with the next character.
>
> Although the processes execute concurrently, each pipe write must happen before the matching read, and thus `3Z2Z1Z4Z` is the only possible output.

b) (2 pts) The program calls `fork()`, but it does not call any variation of the `wait` system call. Will the execution of this program leave zombies behind once it has finished executing and has exited? Say why or why not.

> No. Zombies are dead processes whose parent is still alive (but hasn't reaped them). However, in this case, as the question posits, the parent process has exited. The program can create zombie processes while it's running since it doesn't reap its children, but these zombies would be reaped as soon as the parent exits.

c) (2 pts) The program never calls `close()` on any of the pipe file descriptors. Will the execution of this program lead to resource leaks that are due to this failure to close these file descriptors? Say why or why not.

> No. When a process exits, all file descriptors it has open are automatically closed.

2.2 Know Your Shell (6 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) (2 pts) Run the `nm` command with the `-S` switch on files `a.o`, `b.o`, and `c.o` and redirect both its standard output and standard error streams to a file `log.txt`. Run the command as a foreground job.

`nm -S a.o b.o c.o >& log.txt`
or also
`nm -S a.o b.o c.o >log.txt 2>&1`

b) (2 pts) Grep a dictionary file `words` for the letter `a`, then pipe the result to a second invocation of grep that looks for the letter `b`, and finally pipe that result into the word count (`wc`) command which shall be given the `-l` switch. Run this job in the foreground.

`grep a words | grep b | wc -l`
We also accepted `cat words | grep a | grep b | wc -l`

c) (2 pts) Assume that you have just started a run of `fjdriver.py` to test your p2, but you realize that you forgot to apply a bug fix. What key do you press to terminate `fjdriver.py` and return to the command line (without leaving any stopped processes behind?)

Hit Ctrl-C

3 Multithreading (36 pts)

3.1 A Parallel Function (10 pts)

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.*

Consider the following program:

```
1  static int n;
2  static _Atomic long total;
3
4  static void *operate_on_part(void *_arr)  /* Note that multiple threads will run this */
5  {
6      long *arr = _arr, acc = 0;
7      for (int i = 0; i < n; i++)
8          acc += arr[i];
9
10     total += acc;
11     return NULL;
12 }
13
14 const int N_THREADS = 4;
15 static long operate_on_array(long *arr, long len)
16 {
17     // support only arrays where len is a multiple of 4
18     assert (len % N_THREADS == 0);
19     n = len/N_THREADS;
```

```
20      total = 0;
21      pthread_t t[N_THREADS];
22      for (int i = 0; i < N_THREADS; i++)
23          pthread_create(t+i, NULL, operate_on_part, arr + i*(len/N_THREADS));
24      for (int i = 0; i < N_THREADS; i++)
25          pthread_join(t[i], NULL);
26      return total;
27  }
28
29  int main()
30  {
31      long v[20];
32      for (int i = 0; i < 20; i++)
33          v[i] = i + 1;
34
35      printf("%ld\n", operate_on_array(v, 20));
36  }
```

a) (2 pts) Which memory locations have "conflicting evaluations" as defined above? Refer to them using the variable names used in the program. There are two.[1]

i) (1st) `total`

ii) (2nd) `n`

iii) (3rd) all elements of `v`

b) (4 pts) For each conflicting evaluation, explain whether or not it constitutes a data race, and why it does or does not:

i) (1st) There is no race on `total` since it's declared to be atomic.

ii) (2nd) There is no race on `n` because the only write to it happens before the subsequent reads by the 4 threads. The happens-before relationship is created via `pthread_create`.

iii) (3rd) There is no race on any element of `v` because the only write to it happens in main before the subsequent reads by the 4 threads. The happens-before relationship is also created via `pthread_create`.

We accepted any 2 of the 3 answers.

c) (3 pts) Ignoring the overhead of thread creation and joining, what speedup factor would you expect of the `operate_on_array` function when compared to a sequential function that performs the same functionality? Assume that the machine on which the code runs has a sufficient number of cores, at least 4 independent cores.

We would expect 4x speedup for this embarrassingly parallel function since each quarter of the array can be summed up in parallel.

[1] Actually, three. I had originally intended to omit the `main` function which has conflicting evaluations of the elements of v, but then left it in for illustration purposes.

3.2 Semaphore Puzzle (10 pts)

```
1  int x = 40;
2  pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
3  pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
4  sem_t s1, s2;
5
6  static void *thread_A(void *_)
7  {
8      pthread_mutex_lock(&lock1);
9      sem_wait(&s1);
10     printf("A");
11     sem_post(&s2);
12     sem_post(&s2);
13     pthread_mutex_unlock(&lock1);
14     return NULL;
15 }
16
17 static void *thread_B(void *_)
18 {
19     pthread_mutex_lock(&lock1);
20     printf("B");
21     sem_post(&s1);
22     pthread_mutex_unlock(&lock1);
23     return NULL;
24 }
25
26 static void *thread_C(void *_)   // used twice, see below
27 {
28     sem_wait(&s2);
29     pthread_mutex_lock(&lock2);
30     printf("%d", ++x);
31     pthread_mutex_unlock(&lock2);
32     return NULL;
33 }
34
35 int main()
36 {
37     sem_init(&s1, 0, 0);   // sets initial value to 0
38     sem_init(&s2, 0, 0);
39
40     const int N_THREADS = 4;
41     // Note: thread_C intentionally occurs twice
42     void * (*f[])(void *) = { thread_B, thread_A,
43                               thread_C, thread_C };
44     pthread_t t[N_THREADS];
45     for (int i = 0; i < N_THREADS; i++)
46         pthread_create(t+i, NULL, f[i], NULL);
47
48     // Send a SIGALRM signal in one second;
49     // default disposition is process termination. Assume
50     // that the scheduler will not unduly delay any thread.
51     alarm(1);
52     for (int i = 0; i < N_THREADS; i++)
53         pthread_join(t[i], NULL);
54 }
```

List all possible outputs of this program, with a brief justification for each. This program makes use of an alarm signal that, if triggered, will terminate the entire process including all threads.

This program has 2 possible outputs, which are `BA4142` and not outputting anything at all. (The user would see a message `Alarm Clock` output by the shell.)

Explanation: Thread A and B run concurrently and compete for lock 1. If thread A acquires lock 1 first, it will wait on the semaphore while still holding the lock, preventing thread 2 from acquiring the lock. Since both instances of C are blocked waiting on semaphore 2, this means that all threads in this process are blocked indefinitely and a deadlock has occurred. In one second, an alarm will go off and as described in the code will terminate the process, which includes the termination of all the threads within this process, without having anything output at all.

If, however, thread B acquires lock 1 first, it will print B, then post to semaphore 1. Once thread A acquires lock 1, it will proceed past print A, and then post twice to semaphore 2. These two posts allow the 2 instances of thread C to move past their call to semaphore wait and compete to acquire lock 2. The first C thread to acquire it will compute and output 41, the second 42.

3.3 MPSC (10 pts)

In exercise 3, you implemented a message passing abstraction called a multi-producer, single-consumer channel (mpsc), which provides a way for threads to communicate safely without directly sharing memory. Consider the following incomplete program, which uses the API you implemented:

```
1  #include "mpsc.h"
2  struct txrx {
3      struct mpsc_sender *tx;
4      struct mpsc_receiver *rx;
5  };
6
7  static void *server(void *_txrx)
8  {
9      struct txrx * p = _txrx;
10     for (;;) {  // add code here
11
12
13
14
15
16
17
18
19
20
21
22     }
23     return NULL;
24 }
25 int main()
26 {
27     struct mpsc_channel ch[2];
28     for (int i = 0; i < 2; i++)
29         mpsc_sync_channel(/* CHANNEL_SIZE= */10, ch + i);
30
31     struct txrx p = {
32         .tx = ch[0].sender,
33         .rx = ch[1].receiver
34     };
35     pthread_t serverthread;
36     pthread_create(&serverthread, NULL, server, &p);
37
38     uintptr_t res = 0, s, N;
39     scanf("%lu", &N); assert (N % 2 == 0);
40     for (int i = 1; i < N; i += 2) {
41         assert (mpsc_send(ch[1].sender, i) == SEND_SUCCESS);
42         assert (mpsc_send(ch[1].sender, i+1) == SEND_SUCCESS);
43         assert (mpsc_recv(ch[0].receiver, &s) == RECV_SUCCESS);
44         res += s;
45     }
46     if (res == N * (N+1)/2) printf("Gauss agrees.\n");
47     // add code here
48
49
50     pthread_join(serverthread, NULL);
51 }
```

a) (8 pts) Complete the program such that if the user runs this program and then types in an even number between 2 and 1000000, the program will print "Gauss agrees."

b) (2 pts) Complete the program to ensure that it exits after printing this message.

You may change only the sections in the program as marked: the body of the for loop in the `server` function, and before the call to `pthread_join()` in `main()`. Perform error checking as necessary. The number of lines to add to the `server` function does not need to match the number of lines left open; the given space is only roughly proportional to what must be added.

The server in the program reads two numbers from a channel and sends back their sum through the other channel. To shut down the server, the main thread can drop the sender which causes the server to detect an error when trying to receive.

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

#include "mpsc.h"

struct txrx {
    struct mpsc_sender *tx;
    struct mpsc_receiver *rx;
};

static void *
server(void *_txrx)
{
    struct txrx * p = _txrx;
    for (;;) {
        uintptr_t a, b;
        mpsc_recv_result rc1 = mpsc_recv(p->rx, &a);
        if (rc1 != RECV_SUCCESS) break;
        mpsc_recv_result rc2 = mpsc_recv(p->rx, &b);
        if (rc2 != RECV_SUCCESS) break;
        mpsc_send_result rc3 = mpsc_send(p->tx, a+b);
        if (rc3 != SEND_SUCCESS) break;
    }
    return NULL;
}

int main()
{
    struct mpsc_channel ch[2];
    for (int i = 0; i < 2; i++)
        mpsc_sync_channel(/* CHANNEL_SIZE= */10, ch + i);

    struct txrx p = {
        .tx = ch[0].sender,
        .rx = ch[1].receiver
    };
    pthread_t serverthread;
    pthread_create(&serverthread, NULL, server, &p);

    uintptr_t res = 0, s, N;
```

```
43      scanf("%lu", &N);
44      for (int i = 1; i < N; i += 2) {
45          assert (mpsc_send(ch[1].sender, i) == SEND_SUCCESS);
46          assert (mpsc_send(ch[1].sender, i+1) == SEND_SUCCESS);
47          assert (mpsc_recv(ch[0].receiver, &s) == RECV_SUCCESS);
48          res += s;
49      }
50      if (res == N * (N+1)/2) printf("Gauss agrees. \n");
51      mpsc_drop_sender(ch[1].sender);
52      pthread_join(serverthread, NULL);
53  }
```

We deducted 1 point for otherwise correct approaches that did not use drop but instead sent a special value to the server to get it to exit. Such approaches unnecessarily reduce the generality of the server code and don't exploit the drop mechanism you implemented in ex3.

3.4 Condition Variables (6 pts)

Determine if the following statements about condition variables are true or false.

a) Condition variables must always be used in combination with a lock/mutex and with application-specific state that determines whether to wait on them.

☑ true / ☐ false

b) The wait operation of a condition variable may be unreliable: it can return before a signal is sent, or even if no signal was sent.

☑ true / ☐ false

c) The wait operation on a condition variable should always be called with the same lock.

☑ true / ☐ false

In fact, Java's `java.util.concurrent` API provides `newCondition` as a method on a lock, tying a condition variable to its lock for its lifetime. The builtin java.lang.Object.wait likewise.

d) Condition variables save CPU time when waiting for events.

☑ true / ☐ false

e) Performance degradation and/or deadlock are the most likely results when condition variables are used improperly.

☑ true / ☐ false

There are certainly other errors that can result from misusing the monitor pattern as a whole, but lost wakeups leading to deadlock, or too frequent wakeups are most likely to result from misusing condition variables themselves.

f) Needlessly broadcasting on a condition variable can cause the "thundering herd" phenomenon where more threads are signaled than are needed to usefully react to the change in the underlying shared state.

☑ true / ☐ false

4 Development and Linking (18 pts)

4.1 Linker Errors (4 pts)

A programmer is trying to link their program, which so far consists of only two files, `moda.c` and `modb.c`, shown below.

```c
// moda.c
#include "moda.h"

int get_g() { return g; }
```

```c
// modb.c
#include "moda.h"

int main()
{
    call_a_function();
}
```

Both files refer to a shared header file `moda.h`. Your task is to reconstruct this header file for two separate and independent scenarios. In both scenarios, notice that the errors occur during linking and not during compilation.

a) (2 pts) Suppose the user runs `gcc -Wall moda.c modb.c -o mod`. The following errors are displayed:

```
$ gcc -Wall moda.c modb.c -o mod
/usr/bin/ld: /tmp/ccYaTiCC.o:(.bss+0x0): multiple definition of `g';
    /tmp/ccS0YHBY.o:(.bss+0x0): first defined here
/usr/bin/ld: /tmp/ccYaTiCC.o: in function `main':
modb.c:(.text+0xa): undefined reference to `call_a_function'
collect2: error: ld returned 1 exit status
```

Reconstruct `moda.h` under these assumptions.

> Here, the header apparently included a definition of g which then resulted in multiple definitions of g. It also must have included a declaration of `call_a_function` or else the compiler would have warned about its implicit declaration.

```c
void call_a_function();
int g;
```

> The original version had a typo where main's body read `call_a_func` instead of `call_a_function`. This typo was not discovered until Dr. Back's section took the exam. Under the version with the typo, the closest reconstruction would have been

```c
void call_a_function();

// needed to suppress "function unused" warning in moda.c
static void call_a_func() __attribute__((__unused__));

static void call_a_func() {
    call_a_function();  // produce unresolved reference
}

int g;
```

which comes close to producing the error message (it will complain about the undefined reference twice still).

b) (2 pts) Now, suppose that the programmer makes some changes to `moda.h`. After these changes, the program still won't link, but now the following errors are displayed:

```
$ gcc -Wall moda.c modb.c -o mod
/usr/bin/ld: /tmp/ccIQ7E2b.o: in function `call_a_function':
modb.c:(.text+0x0): multiple definition of `call_a_function';
    /tmp/ccxkJKm3.o:moda.c:(.text+0x0): first defined here
/usr/bin/ld: /tmp/ccxkJKm3.o: in function `get_g':
moda.c:(.text+0xd): undefined reference to `g'
collect2: error: ld returned 1 exit status
```

Reconstruct `moda.h` given this information.

Now there is a definition of `call_a_function`, but no definition of g. There must have been an (extern) declaration of g or else the compiler would have complained.

```
void call_a_function() { }
extern int g;
```

4.2 Static, Extern, or None of the Above (14 pts)

For this problem, let us consider a small program consisting of two C files that will successfully compile, link, and run, but is (potentially) missing storage class modifiers. Possible places where these storage classes could occur are marked with underscores (_____):

```
1  // units.h
2  ________ int unit1_func();
3  ________ int global;
4  ________ int local;
5  ________ void inc_by_global(int *x)
6  {
7      *x += global;
8  }
```

```
1  // unit1.c
2  #include "units.h"
3
4  int unit1_func()
5  {
6      inc_by_global(&local);
7      return local;
8  }
```

```
1  // unit2.c
2  #include "units.h"
3  #include <stdio.h>
4
5  int global = 5;
6  int main()
7  {
8      int resuf = unit1_func();
9      printf("local %d resuf %d\n",
10             local, resuf);
11 }
```

With the correct modifiers, the program links and when run, outputs

```
local 0 resuf 5
```

a) (4 pts) Add any necessary storage class modifiers. Some may be optional, in which case you may either add them or mark them as unneeded by writing `/* NONE */`. Write them inside the program source code above into the blank spaces.

> From the program output, we can infer that `local` must be static since it has different values in units 1 and 2. `inc_by_global` must also be static or else there would be a multiple definition error. `global` must be declared in the header file – defining it would produce a multiple definition error, and defining it static would produce a compiler error in unit 2. Ditto for `unit1_func`.

```
extern int unit1_func();  // or /* NONE */ - extern is optional for functions
extern int global;
static int local;
static void inc_by_global(int *x) {
    *x += global;
}
```

b) (10 pts) Complete the symbol tables that result when `unit1.c` and `unit2.c` are separately compiled, listing each symbol according to its type (data, bss, or text) and scope (local vs global). Include unresolved references as well. Addresses displayed by nm are omitted.

Output of `nm unit1.o`

```
U  ______________
t  ______________
b  ______________
T  ______________
```

```
U global
t inc_by_global
b local
T unit1_func
```

Output of `nm unit2.o`

```
D  ______________
t  ______________
b  ______________
T  ______________
U  ______________
U  ______________
```

```
D global
t inc_by_global
b local
T main
U printf
U unit1_func
```