CS 3214 Sample Midterm

Solutions are shown in this style. This exam was given Spring 2018.

1. Compiling and Linking (17 pts)

a) (4 pts) *Separate Compilation.* Consider the following module.c file:

```
// module.c
int iiii;
static double dddd;
extern float eeee;
short ssss = 42;

static float
ffff(float gggg) {
    return gggg + eeee;
}
```

When running `gcc -c -m32 module.c; nm module.o` which of the following is output? (Hint: -m32 compiles in 32-bit mode in which integers are 32-bit long.) Check one!

A ❑	B ❑	C ☑	D ❑

`00000000 b dddd`
`00000000 t ffff`
`00000004 C iiii`
`00000000 D ssss`
	`00000000 b dddd`
	`U eeee`
	`00000000 t ffff`
	`00000004 D iiii`
	`00000000 s ssss`
	`00000004 d gggg`
		`00000000 b dddd`
		`U eeee`
		`00000000 t ffff`
		`00000004 C iiii`
		`00000000 D ssss`
			`00000000 d dddd`
			`U eeee`
			`00000000 T ffff`
			`00000004 C iiii`
			`00000000 D ssss`

Explanation:
A is missing the external reference to eeee which shows up as U
B includes gggg, which is local variable that's resolved by the compiler
D has ffff as a global symbol, but it's static. Ditto for ssss.

b) (4 pts) As you may remember from exercise 2, the `size` command outputs the size (in bytes) taken up by the text, data, and bss sections of an executable or object module.
If we then ran `size module.o`, what output would we see?
Hint: since weakly defined symbols have not been resolved (we have not invoked the linker), `size` will not include any memory needed for them yet.

	text	data	bss	dec	hex	filename
A ❑	70	12	4	86	56	module.o
B ☑	70	2	8	80	50	module.o

C ❑	`text data bss dec hex filename` `70 2 12 84 54 module.o`
D ❑	`text data bss dec hex filename` `0 2 8 10 A module.o`

Explanation:
The short ssss takes up 2 bytes in the data section, and the double dddd takes up 8 bytes in the BSS section. As per hint, the weak global iiii is not counted. There are no other definitions. The module contains a function, so the size of the program text cannot be 0.

c) (9 pts) Let's add a second file, main.c, as follows:

```
// main.c
float ffff(float gggg);

int main()
{
    ffff(0.5);
}
```

If you compiled and linked the two files like so, you'd see:

```
$ gcc main.c module.c
/tmp/cc2sWLpp.o: In function `main':
main.c:(.text+0xd): undefined reference to `ffff'
/tmp/ccxL3Lms.o: In function `ffff':
module.c:(.text+0xd): undefined reference to `eeee'
collect2: error: ld returned 1 exit status
```

Which strategies will work to fix these errors?
Check all that apply (i.e., strategies that would, when applied in isolation, correct the error)!

To correct/avoid the `undefined reference to `ffff’` error, we can

A ❑	Move the declaration `float ffff(float gggg)` from the main.c file to a .h header file (and include it in both module.c and main.c)
B ❑	Add the keyword `extern` to the declaration in main.c, e.g. `extern float ffff(gggg)`
C ❑	Add the keyword `static` to the declaration in main.c
D ☑	Remove the keyword `static` from the definition in module.c

To correct/avoid the `undefined reference to `eeee'` error, we can

A ☑	Replace the keyword `extern` with the keyword `static` in the declaration of eeee in module.c
B ☑	Add `float eeee;` to main.c
C ☑	Remove the keyword extern from the declaration of eeee in module.c
D ❑	Add `extern float eeee;` to main.c
E ☑	Add `float eeee;` to a header file that is included at the top of both main.c and module.c

You can try all of these out yourselves and look at the resulting symbol tables.

This was graded like 9 true/false questions (1 pts for correct answer each.)

2. Understanding Unix Processes (30 pts)

a) (5 pts) How many processes are created when the user types in ./fork3 on the command line, where fork3 is the executable corresponding to the following C program:

```
// fork3.c
int
main()
{
    fork();
    if (fork())
        fork();

    sleep(1000000);
}
```

Check one!

A ❑	B ❑	C ❑	D ☑	E ❑
3	4	5	6	7

Explanation: `ps f` would show:

```
  PID TTY      STAT   TIME COMMAND
18573 pts/1    Ss     0:00 -bash
22455 pts/1    S      0:00  \_ ./fork3
22456 pts/1    S      0:00  |   \_ ./fork3
22458 pts/1    S      0:00  |   |   \_ ./fork3
22460 pts/1    S      0:00  |   |   \_ ./fork3
22457 pts/1    S      0:00  |   \_ ./fork3
22459 pts/1    S      0:00  |   \_ ./fork3
22461 pts/1    R+     0:00  \_ ps f
```

b) (5 pts) Consider the following program wallclockpuzzle, which is compiled from wallclockpuzzle.c, shown below:

```
// wallclockpuzzle.c
int
main()
{
    if (fork()) {
        sleep(2);
    } else {
        sleep(4);
    }
}
```

Consider what happens when running the ‘time’ bash built-in as follows

```
$ time ./wallclockpuzzle
```

The user will see output that has the following format:

```
real    0m<REAL>s
user    0m<USER>s
sys     0m<SYSTEM>s
```

where (a) **`<REAL>`**, (b) **`<USER>`**, and (c) **`<SYSTEM>`** are replaced with (a) the number of seconds that elapsed as could be observed on a wall clock, (b) the number of CPU seconds the process and all its descendants spent executing in user mode, and (c) the number of CPU seconds the process and its descendants spent executing in kernel mode, respectively.

What’s a possible output of running `time` as shown above?

A ❑	B ❑	C ❑	D ☑
real 0m2.004s user 0m0.001s sys 0m2.002s	real 0m4.004s user 0m6.001s sys 0m0.002s	real 0m2.004s user 0m2.001s sys 0m0.002s	real 0m2.004s user 0m0.001s sys 0m0.002s

Explanation: the shell waits only for the process it started, not for grandchildren, and this process exits after 2 wall clock seconds. Neither the parent nor the child use up significant CPU time since they spend all of their time being BLOCKED while sleeping in the sleep() system call.

c) (20 pts) Implement the functions `ll_popen()` and `ll_pclose()` according to the specification given below. (Error handling is not required.)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/wait.h>

/*
NAME
       ll_popen, ll_pclose - pipe stream to or from a process

SYNOPSIS

       struct ll_pipe *ll_popen(const char *command, enum ll_pipe_direction
type)

       int ll_pclose(struct ll_pipe *pipe)

DESCRIPTION
       The  ll_popen() function opens a process by creating a pipe,
       forking, and invoking the shell.  Since a pipe is by definition
       unidirectional, the type argument may specify only reading
       or writing, not both; the resulting stream is correspondingly
       read-only or write-only.

       The command argument is a pointer to a null-terminated string
       containing a shell command line.  This command is passed to
       /bin/sh using the -c flag;  interpretation,  if  any, is
       performed by the shell.

       The type argument is either READ_FROM_PROCESS or WRITE_TO_PROCESS
       depending on whether the resulting file descriptor is to be used
       to read the process's standard out or to write to the process's
       standard input stream.

       input is the same as that of the process that called popen().

       The return value from ll_popen() is a dynamically allocated
       struct that contains a field 'fd' which the caller may use to
       access the connected file descriptor.

       The ll_pclose() function shall wait for the associated process to
       terminate and return the exit status of the command as returned
       by waitpid(2).
*/
struct ll_pipe {
    int child_pid;  // pid of child process
    int fd;         // file descriptor used to read/write
};

enum ll_pipe_direction {
    READ_FROM_PROCESS,
    WRITE_TO_PROCESS
};
```

```
struct ll_pipe *ll_popen(const char *command, enum ll_pipe_direction type)
{
    int fd[2];

    pipe(fd);
    pid_t child = fork();

    if (child == 0) {
        const char *av[] = { "/bin/sh", "-c", command, NULL };
        if (type == READ_FROM_PROCESS) {
            dup2(fd[1], 1);
        } else {
            dup2(fd[0], 0);
        }
        close(fd[0]);
        close(fd[1]);
        execv(av[0], (char * const *) av);
        abort();
    } else {
        struct ll_pipe * ret = malloc(sizeof(*ret));
        ret->child_pid = child;
        if (type == READ_FROM_PROCESS) {
            close(fd[1]);
            ret->fd = fd[0];
        } else {
            close(fd[0]);
            ret->fd = fd[1];
        }
        return ret;
    }
}

int ll_pclose(struct ll_pipe *pipe)
{
    int status;
    close(pipe->fd);
    waitpid(pipe->child_pid, &status, 0);
    free(pipe);

    return WEXITSTATUS(status);
}

/* Test program follows. */
int main(int ac, char *av[])
{
    bool reading = !strcmp(av[1], "-r");
    struct ll_pipe * pipe = ll_popen(av[2],
            reading ? READ_FROM_PROCESS : WRITE_TO_PROCESS);
    char c;
    if (reading) {
        while (read(pipe->fd, &c, 1) > 0)
            write(1, &c, 1);
    } else {
        while (read(0, &c, 1) > 0)
```

```
            write(pipe->fd, &c, 1);
    }

    ll_pclose(pipe);
}
```

If your functions work correctly, this program would output:

```
$ ./popen -r "head -3 /etc/motd"

Welcome to the Computer Science remote login service.

$
```

Or

```
$ head -3 /etc/motd | ./popen -w wc
      3       8      56
```

3. MultiThreading (35 pts)

a) (5 pts) Consider the following program:

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

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

static void*
thread(void *arg)
{
    pthread_mutex_lock(&lock);
    printf("%s\n", (char *) arg);
    pthread_mutex_unlock(&lock);
}

int
main()
{
  pthread_t t[5];
  const char * msgs[] = { "A", "B", "C", "D", "E" };

  for (int ti = 0; ti < 2; ti++)
    pthread_create(t+ti, NULL, thread, (void *)msgs[ti]);

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

  for (int ti = 2; ti < 5; ti++)
    pthread_create(t+ti, NULL, thread, (void *)msgs[ti]);

  for (int ji = 2; ji < 5; ji++)
    pthread_join(t[ji], NULL);
}
```

How many different possible outputs does this program have?

A	B	C	D	E	F
❑	❑	❑	❑	☑	❑
1	2	5	6	12	120

Explanation: A and B can be printed in any order, as can C, D, and E, but C cannot be printed until after A & B have been printed, hence 2! * 3! = 12 possible outputs.

b) (8 pts) Carefully examine the following program:

```
sem_t s[5];

static void*
thread_A(void *_)
{
    printf("A");
    sem_post(&s[0]);
    sem_post(&s[1]);
}

static void*
thread_B(void *_)
{
    printf("B");
    sem_wait(&s[0]);
    sem_post(&s[2]);
}

// Note the order of the printf,
// sem_wait, sem_post etc. calls
// in each thread_* function!
```

```
static void*
thread_C(void *_)
{
    sem_wait(&s[2]);
    printf("C");
    sem_post(&s[3]);
}

static void*
thread_D(void *_)
{
    sem_wait(&s[3]);
    sem_wait(&s[4]);
    printf("D");
}

static void*
thread_E(void *_)
{
    sem_wait(&s[1]);
    printf("E");
    sem_post(&s[4]);
}
```

```
int
main()
{
    for (int i = 0; i < 5; i++)
        sem_init(&s[i], 0, 0);

    void * (*f[])(void *) = { thread_A, thread_B,
            thread_C, thread_D, thread_E };

    pthread_t t[5];
    for (int i = 0; i < 5; i++)
        pthread_create(t+i, NULL, f[i], NULL);

    for (int i = 0; i < 5; i++)
```

```
        pthread_join(t[i], NULL);
}
```

Which of the following are **not** a possible output of this program?
Check all that apply!

A ❑	B ❑	C ❑	D ☑	E ❑
ABCED	BACED	AEBCD	AECBD	BAECD

Explanation: The semaphores impose constraints A → C → D and B → C → D and A → E → D. (Note that thread_B call printfs before waiting.) Only answer D violates one of these.

This was graded as follows: If D was not checked, 0 pts. If D was checked, -2 pts deduction for any other option also checked, capped at 0.

c) (22 pts) Consider the program shown below, which contains a bug.

```
#include <pthread.h>
#include <stdio.h>
#include <sys/time.h>
/*
 * The following program measures the time required to start one thread,
 * and reports it within the thread itself.
 */

struct time_range {
    struct timeval before, after;
    int time_available;
    pthread_mutex_t lock;
    pthread_cond_t cond;
} tr;

static void* thread(void *_) {
    struct timeval diff;
    pthread_mutex_lock(&tr.lock);
    while (!tr.time_available)
         pthread_cond_wait(&tr.cond, &tr.lock);
    pthread_mutex_unlock(&tr.lock);
    timersub(&tr.after, &tr.before, &diff);
    printf("Creating this thread took %ld microseconds\n", diff.tv_usec);
}

int main() {
    pthread_t t;

    pthread_mutex_init(&tr.lock, NULL);
    pthread_cond_init(&tr.cond, NULL);

    gettimeofday(&tr.before, NULL);
```

```
    pthread_create(&t, NULL, thread, NULL);
    gettimeofday(&tr.after, NULL);

    pthread_mutex_lock(&tr.lock);
    tr.time_available = 1;
    pthread_cond_signal(&tr.cond);
    pthread_mutex_unlock(&tr.lock);

    pthread_join(t, NULL);
}
```

i. (2 pts) Describe the problem in one sentence!

The thread could access tr.after before it has been set.

ii. (4 pts) This kind of bug is commonly classified as (check one!)

A ❑	B ❑	C ❑	D ☑
Starvation	Deadlock	Atomicity Violation	Order Violation

iii. (12 pts) Fix the bug by introducing a **condition variable in a manner that avoids busy-waiting.** Add your bug fixes in the empty spaces in the code on the previous page. You may not change any of the existing code. Add additional variables as needed and show their initialization (if any)! You must use a condition variable for credit.

See above.

For credit, it was required that you use a condition variable, not a semaphore.

iv. (4 pts) Would it have been possible to correct the bug using just a mutex (and without condition variables or semaphores?) If so, briefly sketch how! If not, explain why not! (Maximum 2 sentences.)

Yes, in fact, it would have been easier to do so – the main thread could acquire a mutex before calling pthread_create() and release it after calling gettimeofday(&tr.after). The thread would acquire the lock before accessing tr.after, thus ensuring that it won't access tr.after until after the main thread set it.

Code below.

```
struct time_range {
    struct timeval before, after;
    pthread_mutex_t lock;
} tr;
```

```
static void* thread(void *_)
{
    struct timeval diff;
    pthread_mutex_lock(&tr.lock);  // won't get lock until tr.after is set.
    pthread_mutex_unlock(&tr.lock);

    timersub(&tr.after, &tr.before, &diff);
    printf("Creating this thread took %ld microseconds\n", diff.tv_usec);
}

int
main()
{
    pthread_t t;

    pthread_mutex_init(&tr.lock, NULL);

    pthread_mutex_lock(&tr.lock);   // acquire lock before spawning thread.
    gettimeofday(&tr.before, NULL);
    pthread_create(&t, NULL, thread, NULL);
    gettimeofday(&tr.after, NULL);
    pthread_mutex_unlock(&tr.lock); // release lock after setting tr.after

    pthread_join(t, NULL);
}
```

4. Potpourri (18 pts)

a) (4 pts) The following paragraph is from Cantrill & Bonwick's 2008 paper on Real-world Concurrency. One word was elided from the paragraph. Find out which word!

Learn to debug postmortem. *Among some Cassandras of concurrency, a deadlock seems to be a particular bogeyman of sorts, having become the embodiment of all that is difficult in lock-based multithreaded programming. This fear is somewhat peculiar, because deadlocks are actually among the simplest pathologies in software: because (by definition) the threads involved in a deadlock cease to make forward progress, they do the implementer the service of effectively freezing the system with all state intact. To debug a deadlock, one need have only a list of threads, their corresponding stack backtraces, and some knowledge of the system. This information is contained in a snapshot of state so essential to software development that its very name reflects its origins at the dawn of computing: it is a* core dump.

The elided word is deadlock

The prize for the funniest non-answer goes to "thread pool," which most certainly do not constitute one of the "simplest pathologies" in software.

b) (4 pts) (Fill in the blank!) A system in which all threads or processes are in the BLOCKED state, but which otherwise is operating normally, is said to be ____________ idle ____________.

Note that the other plausible answer "deadlocked" was clearly ruled out by the stipulation "otherwise is operating normally" which cannot be said about a deadlocked system.

c) (10 pts) Some OS textbooks use the term "Limited Direct Execution" to describe the mechanism by which the OS can abstract and share a physical CPU (or core) between programs. For instance, in the book Operating Systems: Three Easy Pieces, Arpaci-Dusseau & Arpaci-Dusseau describe this mechanism as follows:

In order to virtualize the CPU, the operating system needs to somehow share the physical CPU among many jobs running seemingly at the same time. The basic idea is simple: run one process for a little while, then run another one, and so forth. By time sharing the CPU in this manner, virtualization is achieved. (..)

To make a program run as fast as one might expect, not surprisingly OS developers came up with a technique, which we call limited direct execution. The "direct execution" part of the idea is simple: just run the program directly on the CPU. Thus, when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry point (i.e., the main() routine or something similar), jumps to it, and starts running the user's code. (...)

Sounds simple, no? But this approach gives rise to a few problems in our quest to virtualize the CPU. The first is simple: if we just run a program, how can the OS make sure the program doesn't do anything that we don't want it to do, while still running it efficiently? The second: when we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the time sharing we require to virtualize the CPU?

i. (4 pts) Answer question 1: "how can the OS make sure the program doesn't do anything that we don't want it to do, while still running efficiently?"

The OS runs user programs in a mode with restricted privileges (i.e., user mode). (User mode is an execution mode in which attempts by the program to directly access privileged system resources are rejected by the CPU, leading to a trap that the OS must handle.)

For a correct answer, it sufficed to say “user mode” or “less privileged mode.”

ii. (6 pts) Answer question 2: how can the OS “stop [a program] from running and switch to another process”? (Note: the question asks for 2 related, but separate things: how to stop, and how to switch.)

The OS stops programs from running through the use of interrupts, such as timer interrupts, which force a transfer into kernel mode even if the program does not make a system call or yield.

To switch to another process, the OS must save the state of the current process and restore (or create) the state of the process to which it is switching (aka perform a context switch).

A correct answer needed to include some viable means of CPU preemption (e.g. timer interrupts), as well as a discussion that the OS must switch between contexts in a way that involves saving & restoring the user processes’ state.

Many students here talked about signals, but time-sharing is an independent concept, and the context of the question made it clear that the question asked how a process in “limited direct execution” – running in usermode directly on the CPU can be stopped and then switch to another process.