CS 3214 Spring 2023 Midterm Solutions

April 6, 2023

1 Operating System Concepts (19 pts)

1.1 Basic OS Ideas (5 pts)

Check if the following statements are true or false.

a) Operating Systems provide abstractions that can be accessed through interfaces.

☑ true / ☐ false

b) Examples of abstractions provided by an OS include CPU control registers (e.g., `%cr3`), memory-mapped I/O devices, and devices such as timer chips/circuitry.

☐ true / ☑ false

These are examples of details abstractions are designed to hide.

c) The OS kernel is the boot program that allows a computer to start up, at which point it will finish and pass control to application programs.

☐ true / ☑ false

The kernel is not just a boot program, but it provides services to applications and manages resources while the machine is running.

d) Dual-mode operation provides a line of defense against bugs in untrusted user code.

☑ true / ☐ false

e) When a process terminates, the OS will close all of its low-level file descriptors automatically.

☑ true / ☐ false

1.2 BLOCKED processes (7 pts)

In class, we discussed how to model the execution of processes using a simplified process state model. Check what events would cause a process to enter the BLOCKED state:

a) the process issues the `sleep()` system call with an argument of 10 seconds

☑ yes / ☐ no

b) the process enters an infinite loop

☐ yes / ☑ no

c) the scheduler switches out the process to run a different process (time sharing)

☐ yes / ☑ no

d) the process performs the `getpid()` system call

☐ yes / ☑ no

e) the process attempts to lock a (process-shared) mutex that is held by another process

☑ yes / ☐ no

f) the process performs a read on a file descriptor corresponding to a file whose data has not yet been read from an I/O device

☑ yes / ☐ no

g) the process uses C11 atomics to loop on a flag (e.g., "`while (!done) ...`")

☐ yes / ☑ no

1.3 To syscall or not to syscall (7 pts)

As we learned in class, a system call is the mechanism with which a process in userspace utilizes services from the OS kernel. But not all services must be implemented in the kernel. For the cases below, identify which would most likely require a system call:

a) routine that performs a string comparison

☐ yes / ☑ no

b) routine that spawns a thread that can simultaneously execute on another CPU

☑ yes / ☐ no

c) routine that computes a cryptographic signature over a piece of data

☐ yes / ☑ no

d) routine that writes data to an I/O device

☑ yes / ☐ no

e) routine that sets up a communication channel between two processes

☑ yes / ☐ no

f) routine that spawns multiple lightweight cooperatively scheduled (non-preemptive) tasks

☐ yes / ☑ no

g) routine that computes a thumbnail of a jpg image

☐ yes / ☑ no

2 Unix Processes and IPC (27 pts)

2.1 Four forks (8 pts)

No OS exam would be complete without a puzzle surrounding the `fork()` system call. This one involves four programs A through D, shown below.

Program A

```
#include <unistd.h>

int
main()
{
    if (fork())
        fork();
    sleep(1000);
}
```

Program B

```
#include <unistd.h>

int
main()
{
    if (!fork())
        fork();
    sleep(1000);
}
```

Program C

```
#include <unistd.h>

int
main()
{
    if (fork())
        if (fork())
            fork();
    sleep(1000);
}
```

Program D

```
#include <unistd.h>

int
main()
{
    if (fork())
        fork();
    else
        fork();
    sleep(1000);
}
```

Tree 1

```
    PID CMD
1025278 -bash
1031687  \_ ./ft
1031688      \_ ./ft
1031690      |   \_ ./ft
1031689      \_ ./ft
```

Tree 2

```
    PID CMD
1025278 -bash
1030367  \_ ./ft
1030368      \_ ./ft
1030369      \_ ./ft
```

Tree 3

```
3809960 -bash
3979126  \_ ./ft
3979127      \_ ./ft
3979128      \_ ./ft
3979129      \_ ./ft
```

Tree 4

```
    PID CMD
1025278 -bash
1031222  \_ ./ft
1031223      \_ ./ft
1031224          \_ ./ft
```

Each of the four programs (A through D) produced one of the four process tree diagrams (1 through 4) when started in the background. Which program produced which tree?

Program	A	B	C	D
Tree	2	4	3	1

2.2 Pipes, I/O redirection, and `posix_spawn` (8 pts)

A group working on their project 2 attempted to add support for pipes to their shell, but their program did not work. The program below shows the logic they ended up implementing:

```
1  #define _GNU_SOURCE
2  #include <spawn.h>
3  #include <stdio.h>
4  #include <fcntl.h>
5  #include <unistd.h>
6
7  const int WRITE_END = 1;  // this definition is correct per man page
8  const int READ_END = 0;
9  char **environ;
10
11 int
12 main(int argc, char *argv[])
13 {
14     pid_t child_pid;
15     posix_spawn_file_actions_t file_actions;
16     posix_spawn_file_actions_init(&file_actions);
17
18     int pipefds[2];
19     pipe2(pipefds, O_CLOEXEC);
```

```
20
21      posix_spawn_file_actions_adddup2(&file_actions, pipefds[WRITE_END], STDOUT_FILENO);
22      posix_spawn_file_actions_adddup2(&file_actions, pipefds[READ_END], STDIN_FILENO);
23      posix_spawnp(&child_pid, argv[1], &file_actions, NULL, argv+1, environ);
24
25      dup2(pipefds[WRITE_END], STDOUT_FILENO); // dup2(old, new)
26      printf("started %s\n", argv[1]);
27  }
```

a) (2 pts) When this program is compiled and run with

```
./prog cat
```

what would the visible output be and why?

There would be no output because the current program's standard output file descriptor was redirected to the pipe created on line 19, thus the `printf()` statement on line 26 will write data into the pipe.

b) (4 pts) What effect would running `./prog cat` have on the system on which it runs? Justify your answer by describing what the code displayed will actually do.

Careful examination of the `_adddup2` calls reveals that the cat's standard output will be redirected to the write end of the pipe whereas the read end of the pipe is redirected to its standard input. Hence, whatever `cat` writes to its standard output will be sent to its standard input. Initially, `cat` will read the string `"started ..."` from there, output it to its standard output (into the pipe), then read it from there, ad infinitum. Since the parent process does not wait for the child process it creates, the result is a runaway cat process that uses up one core on the machine it runs, making it difficult for students to benchmark their p2. This is in fact what happened to a number of groups in p1.

c) (2 pts) How would the program's effect change if line 26 were removed?

In this case, `cat` would be trying to read from the pipe that is connected to its standard input, but there is nothing to read. Since the pipe's write-end is still open, it'll be blocked there forever (or until the process is terminated). The result is a leftover ("orphaned") `cat` process that is however not consuming any CPU time.

2.3 System Calls and I/O (5 pts)

Consider the following excerpt of a system call trace:

```
openat(AT_FDCWD, "a.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 5
write(5, "Hello, World\n", 13)          = 13
close(5)                                = 0
exit(0)                                 = ?
```

Write a program in a language of your choice that would have produced this system call trace. (You **may not** use the C functions of the same name as the entries in the strace, regardless of the language.)

Here's a Java program:

```
import java.io.*;

public class WriteFile {
    public static void main(String []av) throws IOException {
        var fw = new PrintWriter(new FileWriter("a.txt"));
        fw.println("Hello, World");
        fw.close();
    }
}
```

The complete program (headers, class and function definitions) didn't need to be shown as long as the relevant API calls that would trigger those system calls were visible.

2.4 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 `ps` command with the flag `ax` and sort the output using the `sort` utility, which should be given the `-n` flag.

```
ps ax | sort -n
```

b) (2 pts) Start `apache2` in the background, redirecting its standard output and error streams to `/var/log/httpd2.log`

```
apache2 >& /var/log/httpd2.log &
```

or also

```
apache2 2>&1 >/var/log/httpd2.log &
```

c) (2 pts) Assume that you have just started a long-running command as a foreground job. You change your mind and you wish to send this job into the background. (Assume your shell currently has no other jobs running.) Describe what keystrokes and/or commands you would need to type.

First, type Ctrl-Z, then enter `bg %1` or on bash just `bg` works as well.

3 Multithreading (31 pts)

3.1 Threads and Variables (10 pts)

Consider the following C11 program that uses C11's `_Thread_local` and `_Atomic` keywords:

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <stdint.h>
5
6  static _Thread_local int tls;
7  static int * X;
8  static _Atomic int Y;
9
10 static void print(int index) {
11     printf("tls %d %d\n", index, tls);
12 }
```

```
13
14  static void*
15  thread_func(void *_arg)
16  {
17      uintptr_t myarg = (uintptr_t) _arg;
18      tls = *X + myarg;
19      print (myarg);
20      Y += tls;
21      return NULL;
22  }
23
24  int
25  main()
26  {
27      const int N_THREADS = 4;
28
29      pthread_t t[N_THREADS];
30      X = malloc(sizeof(int));
31      *X = 42;
32      for (uintptr_t i = 0; i < N_THREADS; i++) {
33          pthread_create(t+i, NULL, thread_func, (void *) i);
34      }
35
36      for (int i = 0; i < N_THREADS; i++)
37          pthread_join(t[i], NULL);
38      print (N_THREADS);
39      printf ("%d\n", Y);
40  }
```

a) (6 pts) Provide one possible output of this program.

```
tls 0 42
tls 1 43
tls 2 44
tls 3 45
tls 4 0
174
```

The first four lines (tls 0 through tls 3) could appear in any order.

b) (2 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*

Lines 18 and 31 contain accesses to a shared variable `X`, but there is no lock protecting these accesses. Based on the definition above, do these accesses constitute a data race? Say why or why not.

They do not constitute a data race because there is a happens-before relationship between the write on line 31 in the main thread and the read accesses on line 18 in each of the four threads. The HB relationship follows from the fact that everything that occurs in a (parent) thread happens before anything that's run in any (child) threads later created by that thread.

c) (2 pts) Lines 20 and 39 contain accesses to a shared variable `Y`. Do the accesses to this variable constitute a data race? Say why or why not.

They also do not constitute a data race because the variable is marked with `_Atomic` so it falls under the exception for atomic operations.

3.2 Condition Variables (8 pts)

Consider the following program which uses a condition variable with the goal of letting one thread know about a result computed by a second thread.

```
1  #include <pthread.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <time.h>
5
6  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
7  pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;
8  char *coin;
9
10 static void*
11 thread_A(void *_)
12 {
13     pthread_mutex_lock(&mutex);
14     coin = random() % 2 == 0 ? "head" : "tail";
15     printf("A %s\n", coin);
16     pthread_cond_signal(&condvar);
17     pthread_mutex_unlock(&mutex);
18     return NULL;
19 }
20
21 static void*
22 thread_B(void *_)
23 {
24     pthread_mutex_lock(&mutex);
25     pthread_cond_wait(&condvar, &mutex);
26     printf("B %s\n", coin);
27     pthread_mutex_unlock(&mutex);
28     return NULL;
29 }
30
31 int
32 main()
33 {
34     const int N_THREADS = 2;
35     void * (*f[])(void *) = { thread_B, thread_A };
36
37     srand(time());
38     pthread_t t[N_THREADS];
39     for (int i = 0; i < N_THREADS; i++)
40         pthread_create(t+i, NULL, f[i], NULL);
41
42     for (int i = 0; i < N_THREADS; i++)
43         pthread_join(t[i], NULL);
44 }
```

a) (4 pts) What are the possible outputs of this program? For each possible output, state whether the program will finish or not.

The possible outcomes are either `A head` (or `A tail`) followed by `B head` (or `B tail`) when the program completes, or just `A head` (or `A tail`) in which case the program "hangs", that is, does not finish.

b) (4 pts) Complete the program so that thread B will reliably output the value of `coin` after it has been set on line 14.

To that end, before line 25, add this statement

```
while (coin == NULL)
```

3.3 Semaphores (5 pts)

Consider all possible outputs of the following program:

```
1  #include <pthread.h>
2  #include <semaphore.h>
3  #include <stdio.h>
4
5  sem_t s1, s2;
6
7  static void* thread_A(void *_)
8  {
9      printf("A");
10     sem_post(&s1);
11     return NULL;
12 }
13
14 static void* thread_B(void *_)
15 {
16     printf("B");
17     return NULL;
18 }
19
20 static void* thread_C(void *_)
21 {
22     sem_wait(&s2);
23     printf("C");
24     return NULL;
25 }
26
27 static void* thread_D(void *_)
28 {
29     sem_wait(&s1);
30     printf("D");
31     sem_post(&s2);
32     return NULL;
33 }
34
```

```
35 int main()
36 {
37     sem_init(&s1, 0, 0);
38     sem_init(&s2, 0, 0);
39
40     const int N_THREADS = 4;
41     void * (*f[])(void *) = { thread_A, thread_B,
42                               thread_C, thread_D };
43
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     for (int i = 0; i < N_THREADS; i++)
49         pthread_join(t[i], NULL);
50 }
```

Assuming that each possible output is equally likely, what is the probability that the output of this program will start with `AB`? Justify your answer by showing your work.

Since the semaphores ensure that A is output before D and that D is output before C, but do not impose any constraints on when B is output, the four possible outputs are `ADCB`, `BADC`, `ABDC`, and `ADBC`, making the probability that it starts with `AB` $\frac{1}{4}$ or 25%.

3.4 Threading Performance (4 pts)

(4 pts) Assume you are trying to optimize the performance of a multi-threaded Linux program that uses 32 threads and runs on a machine with 32 cores. The reported CPU utilization is 100% for about half the cores, and about 0% for the other half. Moreover, the `time` utility reports that the program's threads spend most of their time inside the kernel.

Given this information, describe 2 possible ways in which to improve the performance of this program and describe how they would affect the symptoms described!

i) The low CPU utilization on half the cores is indicative of unnecessary serialization, which could be addressed by breaking up the program's lock or locks (if possible), which would increase parallelism and therefore CPU utilization.

ii) The large amount of time spent in the kernel can indicate high lock contention (that is, the slow path where a thread is blocked in the kernel because it tries to acquire a mutex held by another thread is taken often.) A possible (additional) countermeasure here is to decrease the amount of time executing while holding the lock (shortening the critical section) to reduce the likelihood of this happening.

Other answers may include a redesign to avoid shared state in the first place, such as replication or partitioning, or also the use of lock-free algorithms.

3.5 Deadlock (4 pts)

A multithreaded program uses 3 threads and 3 mutexes labeled m_A, m_B, and m_C. In most sections of its code, only a single mutex is acquired and later released before any other mutex is acquired. However, thread 1 has one section in its code where it first acquires m_A and then it acquires m_B before releasing m_A, whereas thread 3 has a section in its code where it first acquires m_C and then m_A, also before releasing

m_C. The locks are released in the opposite order in which they are acquired and all locks are held for a finite amount of time.

Can a deadlock occur in this situation? Justify your answer.

No deadlock can occur because not all four necessary conditions for deadlock are met. In particular, a circular wait cannot occur since the locking order $m_C \rightarrow m_A \rightarrow m_B$ is maintained when multiple locks are requested.

4 Development and Linking (23 pts)

4.1 Whose job is it anyway? (7 pts)

For each of the tasks below, specify which of the C compiler toolchain programs traditionally performs it:
Choose from: preprocessor, compiler, assembler, linker

a) expand macro definitions preprocessor

b) create a relocatable object file (.o) from an assembly file (.s) assembler

c) resolve local variable references to concrete addresses compiler

d) resolve include files preprocessor

e) create an executable binary from a collection of object files (.o) linker

f) resolve global variable references to concrete addresses or offsets linker

g) resolve function calls to functions defined in other .o files to addresses linker

4.2 Where will x end up? (6 pts)

As we learned in class, an ELF binary has different segments where it puts different information relevant to the program, which will map to different regions in memory when the program runs. For the following examples, identify where x will reside from the following choices:

Choose from: .data, .rodata, .bss, .text, stack, heap, nowhere (nothing is defined)

a) `int x;` // at the top level, i.e., outside any function .bss

b) `const int x = 100;` // at the top level, i.e., outside any function .rodata

c) `int x(void) { return 0; }` .text

d) `void foo(void) { int x; ... }` stack

e) `static int x[100];` .bss

f) `extern void x(void);` nothing

4.3 Fully statically linked binaries vs. dynamically linked binaries (4 pts)

Select whether the following are downsides of fully statically linked binaries:

a) they result in larger binaries

☑ yes / ☐ no

b) they do not contain all library dependencies, thus relying on the system libraries

☐ yes / ☑ no

c) they must be rebuilt on every library change/update

☑ yes / ☐ no

d) they cannot make use of link time optimization

☐ yes / ☑ no

4.4 Linker Errors (6 pts)

Consider the following header file `header.h`:

```
// begin header.h
int abc;
extern void ext_fun(void);
extern void module2_fun(void);
// end header.h
```

and the C source files `module1.c` and `module2.c`:

```
// begin module1.c
#include "header.h"

int
main() {
    ext_fun();
    module2_fun();
}
// end module1.c

// begin module2.c
#include "header.h"

void module2_fun(void)
{
    // implementation here
}
// end module2.c
```

a) (1 pts) What command would a user issue to compile these files into object modules (.o files)?

```
gcc -c module1.c module2.c
```

or, separately

```
gcc -c module1.c
gcc -c module2.c
```

b) (1 pt) What command would a user issue to link the resulting .o files into an executable called `main`? Assume that the user will not include any external libraries in this command.

```
gcc -o main module1.o module2.o
```

c) (4 pts) List all errors the linker will produce when linking these files using the command in part b). (Assume that the user uses gcc 10 or later.)

The errors include a "multiple definition" error related to `abc` and an "undefined reference" error related to `ext_fun`:

```
/opt/rh/gcc-toolset-12/root/usr/bin/ld: module2.o:(.bss+0x0): multiple definition
                  of `abc'; module1.o:(.bss+0x0): first defined here
/opt/rh/gcc-toolset-12/root/usr/bin/ld: module1.o: in function `main':
module1.c:(.text+0x5): undefined reference to `ext_fun'
collect2: error: ld returned 1 exit status
```