CS 3214 Spring 2024 Midterm Solutions

March 25, 2024

Contents

1 Operating System Concepts (28 pts)

1.1 Basic OS Principles (10 pts)

Check if the following statements are true or false.

a) An OS kernel is a trusted control program that manages a computer's hardware and provides abstractions that enable user programs to use the hardware's features.

☑ true / ☐ false

b) A typical operating system can execute kernel code directly while running in kernel mode, but to ensure safety, it must emulate user code by running in user mode.

☐ true / ☑ false

User code is directly executed, not emulated in any way.

c) Traditional system calls always involve a mode switch.

☑ true / ☐ false

d) In systems that exploit dual-mode operation, the CPU will silently ignore attempts by user processes to execute privileged instructions.

☐ true / ☑ false

No, such attempts will cause faults that trap into the kernel.

e) The use of per-process address spaces is an essential mechanism that helps the OS to isolate processes from each other.

☑ true / ☐ false

1.2 On Process States (10 pts)

In class, we discussed how to model the execution of processes using a simplified process state model. Determine whether the following statements are true or false.

a) When a process transitions from the `RUNNING` into the `BLOCKED` state while there are processes in `READY` state, a context switch to another process must occur.

☑ true / ☐ false

b) External interrupts can trigger a transition of a process from the `BLOCKED` to the `READY` state.

☑ true / ☐ false

c) Overload situations are characterized by the presence of many more `BLOCKED` than `READY` processes in a system.

☐ true / ☑ false

Having more `BLOCKED` than `READY` processes is normal; an overload situation would have many more `READY` than `RUNNING` processes.

d) A system is idle if and only if there are no processes in the `BLOCKED` state.

☐ true / ☑ false

It's idle if there are no processes in the `RUNNING` state.

e) A process attempting to write to a full pipe will stay in the `RUNNING` state until the pipe is drained by a process reading from the pipe.

☐ true / [✓ false]

No, the process will move into the `BLOCKED` state until the pipe is drained.

1.3 Name The Abstraction (8 pts)

In the context of computer systems, abstraction is the process of hiding the unimportant details and complexity of concrete facilities, resources, or features and instead providing a simplified and uniform way of accessing and using such facilities. Name one abstraction for each of the following facilities:

	Concrete Facility	Abstraction
(2 pts)	CPU	process or thread
(2 pts)	RAM	private address spaces, virtual memory, or just address spaces
(2 pts)	Hard Disk	Files, or file descriptors
(2 pts)	Terminal Device	standard IO streams (or the file descriptors referring to stdin, stdout, etc.

2 Unix Processes, IPC, and Signals (22 pts)

2.1 A Program Combinator (14 pts)

Consider the following incomplete C program `pc.c` which makes use of the POSIX API:

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

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

int
main(int ac, char *av[])
{
    int fd[2];
    ________________;

    int child = ______;
    if (child == 0)
        dup2(____________, STDOUT_FILENO);
    else
        dup2(____________, STDIN_FILENO);

    ____________________;
    ____________________;

    if (child == 0) {
        ____________________;
    } else {
        av[2] = NULL;
        execvp(av[1], av+1);
28      }
29  }
```

When compiled to an executable `pc`, this program can be run as shown in the following 3 example invocations:

```
$ ./pc rev echo 4123SC
CS3214
$ ./pc rev date
4202 TDE MA 84:20:01 32 raM taS
$ ./pc wc cat /etc/passwd
     41     100    2314
```

Complete this program by adding statements on line 12, 14, 16, 18, 20, 21, and 24. For the purposes of this problem, error checking is not required. It is also not required to avoid extraneous file descriptors as long as they do not prevent the invoked programs from terminating.

```
1   #include <unistd.h>
2   #include <stdlib.h>
3   #include <stdio.h>
4
5   const int READ_END = 0;
6   const int WRITE_END = 1;
7
8   int
9   main(int ac, char *av[])
10  {
11      int fd[2];
12      pipe(fd);
13
14      int child = fork();
15      if (child == 0)
16          dup2(fd[WRITE_END], STDOUT_FILENO);
17      else
18          dup2(fd[READ_END], STDIN_FILENO);
19
20      close(fd[WRITE_END]);
21      close(fd[READ_END]);
22
23      if (child == 0) {
24          execvp(av[2], av+2);
25      } else {
26          av[2] = NULL;
27          execvp(av[1], av+1);
28      }
29  }
```

2.2 A Signal Puzzle (8 pts)

Consider the following program in which header files were elided for brevity. The program uses the signal support routines with which you are familiar from project 1. `SIGUSR1` is a signal set aside for user-defined purposes.

```
volatile sig_atomic_t signal_flag;
static void handle_signal(int sig, siginfo_t *c, void *p)
```

```
3  {
4      if (sig == SIGCHLD)
5          write(STDOUT_FILENO, "A", 1);
6      else
7          write(STDOUT_FILENO, "B", 1);
8      signal_flag = 1;
9  }
10
11 int main()
12 {
13     signal_set_handler(SIGUSR1, handle_signal);
14     signal_set_handler(SIGCHLD, handle_signal);
15     int pid = fork();
16     if (pid) {
17         write(STDOUT_FILENO, "C", 1);
18         kill(pid, SIGUSR1);
19         while (signal_flag == 0)
20             continue;
21         write(STDOUT_FILENO, "D", 1);
22     } else {
23         while (signal_flag == 0)
24             continue;
25         write(STDOUT_FILENO, "E", 1);
26     }
27 }
```

What does this program output when run on a Linux machine?

`CBEAD`

This program forks; the child process loops on line 23 until it receives a signal. Thus, the parent process outputs `C` first. After outputting `C`, the parent will send `SIGUSR1` to the child, which causes the child process to execute line 7 and output `B`. The parent, meanwhile, spins on line 19. After the signal handler returns in the child process, the child process output `E` on line 25 and exits. Exiting causes `SIGCHLD` to be sent to the parent, which triggers the signal handler and executes line 5, outputting `A`. Upon return, the parent breaks out of the loop in line 19 and outputs `D` on line 21. Note that after fork, parent and child have separate copies of `signal_flag`.

3 Multithreading (32 pts)

3.1 Data Race or Not (12 pts)

Consider the following C program

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdatomic.h>
4  #include <stdlib.h>
5
6  int X = 39, Z = 41;
7  atomic_int Y = 40;
8  pthread_cond_t C = PTHREAD_COND_INITIALIZER;
9  pthread_mutex_t L = PTHREAD_MUTEX_INITIALIZER;
10
11 static void *thread1(void *_arg) {
```

```
12      pthread_mutex_lock(&L);
13      X = 42;
14      pthread_cond_signal(&C);
15      pthread_mutex_unlock(&L);
16      return NULL;
17  }
18
19  static void *thread2(void *_arg) {
20      pthread_mutex_lock(&L);
21      while (X == 39)
22          pthread_cond_wait(&C, &L);
23      pthread_mutex_unlock(&L);
24      Y = X;
25      return NULL;
26  }
27
28  static void *thread3(void *_arg) {
29      while (Y == 40)
30          continue;
31      Z = Y;
32      return NULL;
33  }
34
35  int
36  main()
37  {
38      pthread_t t[3];
39      pthread_create(t+0, NULL, thread1, NULL);
40      pthread_create(t+1, NULL, thread2, NULL);
41      pthread_create(t+2, NULL, thread3, NULL);
42
43      for (int i = 0; i < 3; i++)
44          pthread_join(t[i], NULL);
45
46      printf ("%d\n", Z);
47  }
```

The C11 memory model defines data races in terms of conflicting evaluations of expressions that refer to memory locations such as the locations at which global variables are stored.

a) (2 pts) Do the evaluations of `X` on lines 13 and 21 constitute a data race? Explain why/why not.

   No, because the unlock/lock operations provided by lock L provides a happens-before relationship.

b) (2 pts) Do the evaluations of `X` on lines 13 and 24 constitute a data race? Explain why/why not.

   Line 21 happens before line 24 in the same thread, so by transitivity there is a happens-before relationship since line 13 happens before 21.

c) (2 pts) Do the evaluations of `Y` on lines 24 and 29 constitute a data race? Explain why/why not.

   Evaluation of atomics are not considered data races by definition.

d) (2 pts) Do the evaluations of `Z` on lines 31 and 46 constitute a data race? Explain why/why not.

   No, because the call to `pthread_join` creates a happens-before relationship.

e) (4 pts) Overall, does the program contain a data race anywhere? If not, provide all possible output(s) of this program. If so, answer "UB" (for "undefined behavior.")

   There is no data race. The output is 42.

There was a typo in the original exam that wasn't discovered until Dr. Back's section took it. In this version, line 29 read `while (Y == 0)`. In this version, the correct answer to this question is that the possible outputs of the program would be either 40 or 42.

3.2 Checking for Shutdown (4 pts)

Consider the following excerpt from a student's p2 implementation:

```
static bool get_shutdown_thread_safe(struct thread_pool *pool) {
    pthread_mutex_lock(&pool->mutex);
    bool flag = pool->shutdown_flag;
    pthread_mutex_unlock(&pool->mutex);
    return flag;
}
```

a) (2 pts) Is this function indeed thread-safe or does it exhibit undefined behavior when called by multiple threads?

It's thread safe.

b) (2 pts) Describe how using this function can lead to errors.

Although this function is thread safe, it is also rather useless. Since the lock protecting the `shutdown_flag` is dropped when returning from the function the information conveyed by it may be out of date (stale) as soon as the lock is dropped. If this return value were used, for instance, to decide whether to wait on a condition variable the calling thread would deadlock if the shutdown flag is set after the lock has been dropped, as some of you learned the hard way in p2.

3.3 Semaphore Puzzle (8 pts)

Consider the following program:

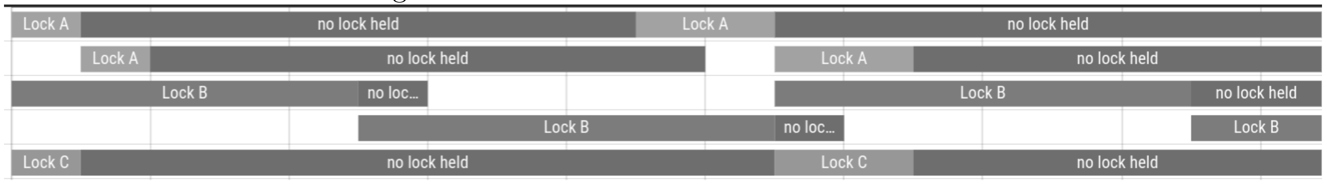
```c
#include <pthread.h>
#include <stdlib.h>
#include <semaphore.h>
#include <stdio.h>

sem_t s1, s2, s3;

static void* thread_A(void *_)
{
    printf("A");
    sem_post(&s1);
    sem_wait(&s2);
    printf("C");
    sem_post(&s3);
    return NULL;
}

static void* thread_B(void *_)
{
    printf("B");
    sem_post(&s2);
    sem_wait(&s1);
    printf("D");
    sem_post(&s3);
    return NULL;
}

static void* thread_C(void *_)
{
    sem_wait(&s3);
    printf("E");
    return NULL;
}

int main()
{
    sem_init(&s1, 0, 0);
    sem_init(&s2, 0, 0);
    sem_init(&s3, 0, 0);

    const int N_THREADS = 3;
    typedef void * (*tf)(void *);
    tf f[] = { thread_A, thread_B, thread_C };

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

    for (int i = 0; i < N_THREADS; i++)
        pthread_join(t[i], NULL);
    printf("F\n");
}
```

List all possible outputs of this program:
A and B may appear in any order in the output. Semaphores s1 and s2 are used to perform a rendezvous. After A and B the next output will be C or D. If C is output, the next output is D or E and if D is output, the next output is C or E. This is because `thread_C` (which outputs E) waits for only a single signal on semaphore s3. F is output last by the main thread after joining all threads. This leaves 8 possible answers:

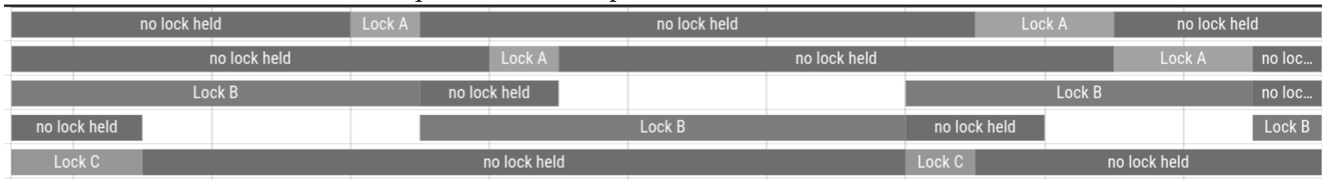
```
ABCDEF
ABCEDF
ABDCEF
ABDECF
BACDEF
BACEDF
BADCEF
BADECF
```

3.4 Lock Performance (8 pts)

A students submitted the following charts for their exercise 3. Their trace started like so



and continued at later timestamps in the same pattern like so:



These charts show traces of where threads were running with locks held and where they were running without. White gaps correspond to times where a thread is blocked. The top line corresponds to thread 1 and the bottom to thread 5 in each chart. Here, threads 1 and 2 acquire Lock A, 3 and 4 Lock B, and 5 acquires Lock C in the pattern shown in these two excerpts. In ex3, we asked for a discussion of the results. What should the student's explanation for this scenario have been? Make sure to discuss critical section lengths and the implications on CPU utilization.

i) (2 pts) Discuss threads 1 and 2.

Threads 1 and 2 hold their shared lock A only for short periods of time relative to the time spent not holding the lock, thus exhibit little wait time and good CPU utilization on their respective cores.

ii) (2 pts) Discuss threads 3 and 4.

Threads 3 and 4 hold their shared lock B for relatively long periods of time relative to the time spent not holding the lock, thus exhibit lots of wait time and CPU utilization drops significantly, parallelism is lost.

iii) (2 pts) Discuss thread 5.

Thread 5 acquires lock C probably redundantly. The lock is never contended, so there's no loss of CPU utilization regardless of how long the lock is held or not.

iv) (2 pts) If these were traces of an actual application, what would you recommend to improve performance?

The developer should investigate if it is possible to either break up lock B or to reduce the length of the critical section during which it is held. (Needed to name only one idea.)

4 Development and Linking (18 pts)

Consider a C file `unit.c`. When compiled with `gcc -c unit.c` the command `nm unit.o` reports the following symbol table:

```
0000000000000000 b bar
0000000000000000 D DAR
000000000000000c t foo
0000000000000000 T FOO
                 U WOZ
```

a) (10 pts) Provide one possible reconstruction of `unit.c`.

```
extern int WOZ;
int FOO() { return WOZ; }

static void foo() { }

int DAR = 42;
static double bar;
```

Note that for `WOZ` to become an external reference ('U') there needs to be an actual reference to it. We can't tell if it's a variable or a function; in my example, it's a variable.

b) (8 pts) `unit.c` does not contain a `main()` function. Consider the following file `mainunit.c`:

```
// mainunit.c
int main() { return 0; }
```

i) (2 pts) What command would link `unit.c` and `mainunit.c` into an executable named `exe`?
`gcc unit.c mainunit.c -o exe` for instance. Also `gcc -o exe unit.o mainunit.c` etc.

ii) (2 pts) Above attempt at linking is bound to fail with an error. What error would you expect the linking command to display?
It would fail with an error such as `undefined reference to 'WOZ'`

iii) (2 pts) Assuming that best practices are followed, why will adding an `#include` statement not fix the error?
Header files contain declarations of functions and variables, but they should not define global symbols, so including a header file will not add the definition needed to resolve this undefined reference.

iv) (2 pts) Provide a suitable fix for the error.
Add a definition for 'WOZ', such as `int WOZ;` or `void WOZ(void) { /* body */ }` depending on whether you declared it as a variable or function in part a).