CS 3214 Fall 2020 Test 1 Solution

November 6, 2020

Contents

Rules

- This exam is open book, open notes, and open Internet, but in a read-only way.
- You are not allowed to post or otherwise communicate with anyone else about these problems.
- You are required to cite any sources you use, except for lecture material, source code provided as part of the class materials, and the textbook. Failure to do so is an Honor Code violation.
- If you have a question about the exam, you may post it as a *private* question on Piazza. If it is of interest to others, we will make it public as a clarification.
- Any errata to this exam will be published prior to 12h before the deadline on the website.

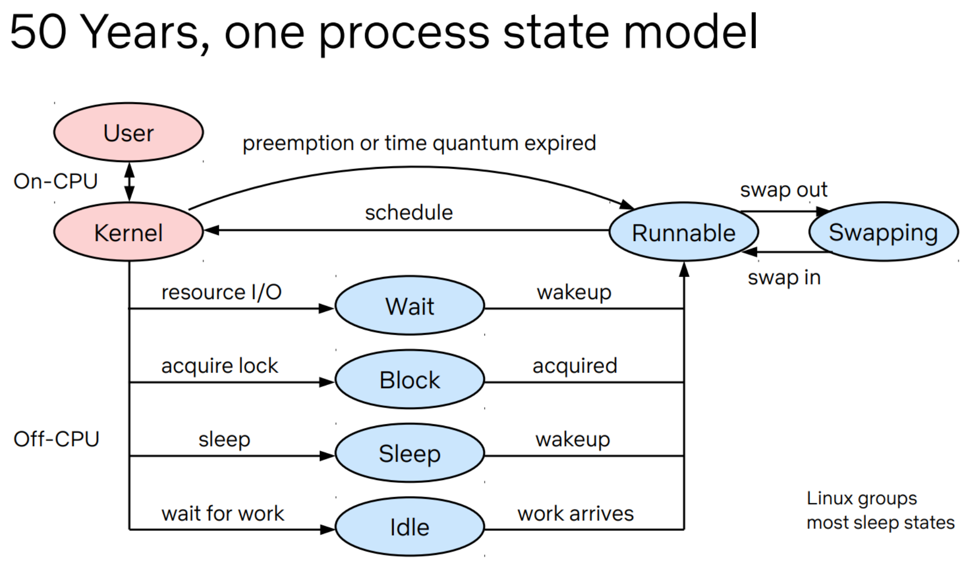

Figure 1: Source: Brendan Gregg, UbuntuMasters 2019: `http://www.brendangregg.com/Slides/UM2019_BPF_a_new_type_of_software.pdf`

1 Process Management (22 pts)

1.1 Understanding Process States (4 pts)

Brendan Gregg is a performance engineer at Netflix and the author of a number of books and talks related to Linux performance. His 2019 talk given at UbuntuMasters 2019 contains the slide shown in Figure 1 in which he presents his view of how to model a process's (or thread's) state as it is relevant to its execution.

Based on your understanding of how the OS manages processes, map the states shown in Figure 1 to the simplified process state diagram discussed in our lectures. Link each state shown in Figure 1 (i.e., blue and red ovals) to which state in the simplified diagram it maps, if any.

[Solution]

- The red states (User/Kernel) map to the RUNNING state: a process is executing, either in user or in kernel mode
- The blue Wait, Block, Sleep, and Idle states are all captured by the BLOCKED state: a process cannot make progress since it's waiting for some event to unblock it
- The blue Runnable state is what we call READY: a process could make progress, but must wait for a CPU to become available
- The blue Swapping state was not modeled in our state diagram

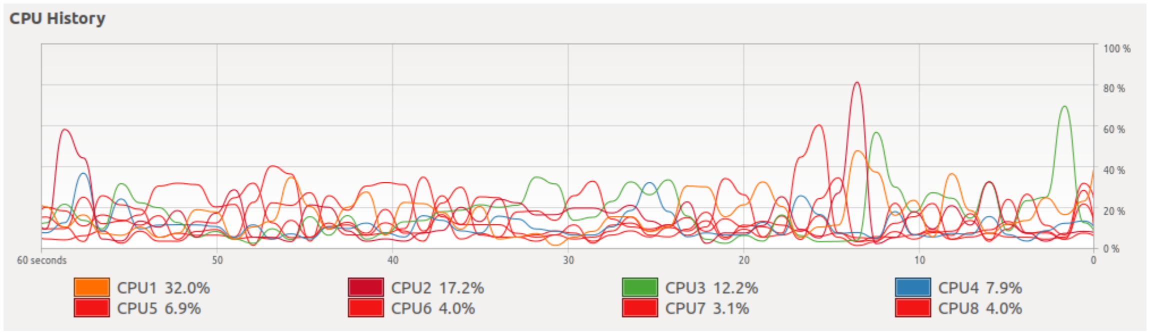

Figure 2: A snapshot obtained from GNOME System Monitor while preparing this test. The chart scrolls to the left, updating continuously with new samples. The percentage CPU utilization shown for each CPU represent the last sample taken.

1.2 A CPU Usage Scenario (6 pts)

When Dr. Back prepared this test, he opened the GNOME System Monitor application which displays CPU usage over time. He took a screenshot which is shown in Figure 2.

Answer the following questions:

- During the time frame shown, were there any processes in the `READY` state? If so, estimate, roughly, what percentage of the time frame such processes were in the `READY` state.
- Give one example of an application Dr. Back could have been using during the time frame shown (aside from GNOME System Monitor and the application used to take the screenshot) and explain how this application could produce the CPU usage pattern seen.
- Give one example of an application Dr. Back definitely was not using during the time frame shown and explain why.

Justify your answers for all 3 parts. If it is not possible to answer the question with the information given, explain why!

[Solution]

1. There were no processes in the READY state (other than for very small amounts of time during state transitions) because the OS will not place processes in this state if a CPU is available to schedule them. Here, the overall CPU consumption was under 100% during the entire time frame - hence, any READY process could be immediately scheduled and placed into the RUNNING state.

   Side note: This argument assumes perfect and immediate load balancing between the ready queues maintained by each CPU. If a process becomes READY on a CPU that is currently RUNNING a process, that second process needs to be migrated to an idle CPU. During the small delay during which this takes place the process is also in the READY state.

2. The applications running showed short bursts of CPU usage (being in the RUNNING state) followed by periods with no CPU usage (while in the BLOCKED state). One application that

shows this pattern is a web browser such as Chrome or Firefox: they are BLOCKED most of the time, except for short periods in which they react to keyboard/mouse input, or update their UI in response to a network message received (such as in a tab showing new Discord messages).

3. He could not have used any applications that are steadily CPU bound, such as a program doing numerical computations (e.g. a bitcoin miner, or a CPU-based video encoding program, or a buggy cush program that go itself into an infinite loop because it failed to check system call return codes).

1.3 Forking a Chain (8 pts)

Consider a program `deep` which, when compiled, is run with 2 command line arguments *depth* and *time*, both representing small positive integers.

When run with a *depth* of 4 and a *time* of 3 using the bash builtin `time`, it should output:

```
$ time ./deep 4 3

real     0m3.002s
user     0m0.001s
sys      0m0.001s
```

If one runs the `ps fx` command to output the process tree while the command is running, you would see this:

```
20367 pts/5    Ss     0:00 bash
28615 pts/5    S+     0:00  \_ ./deep 4 3
28616 pts/5    S+     0:00      \_ ./deep 4 3
28617 pts/5    S+     0:00          \_ ./deep 4 3
28618 pts/5    S+     0:00              \_ ./deep 4 3
```

Thus, *depth* represents the depth of the process chain created, and *time* represents the amount of real time passed. The amount of user and system time **must** always be negligibly small. Once the `time` command outputs to the terminal, all processes created by `deep` are gone.

Implement a `deep.c` that meets the above criteria.

[Solution]

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int
main(int ac, char *av[])
{
    int n = atoi(av[1]);
    int t = atoi(av[2]);
    for (int i = 0; i < n-1; i++)
```

```
        if (fork()) {
            wait(NULL);
            exit(0);
        }

    sleep(t);
}
```

Side note: guaranteeing that only *time* seconds of real time passed requires that the last child process that sleeps is scheduled immediately after it wakes up from sleep (which places it in the READY state). On a heavily loaded machines this process may compete with other processes also wanting to be scheduled. Any such delays would increase the wall clock time that has passed. In general, when you use `sleep()`, there is no guarantee that the program will be run right after the sleep period has passed.

1.4 Terminating Processes (4 pts)

Consider a program `armored` that, when executed and subjected to the bash `kill` builtin command, terminates randomly with a probability of ca. 20%. The program will not exit for other reasons. For instance, the following shell interaction may result:

```
$ ./armored &
[1] 19202
$ jobs
[1]+  Running                 ./armored &
$ kill %1; jobs
[1]+  Running                 ./armored &
$ kill %1; jobs
[1]+  Running                 ./armored &
$ kill %1; jobs
[1]+  Running                 ./armored &
$ kill %1; jobs
[1]+  Running                 ./armored &
$ kill %1; jobs
-bash: kill: (19202) - No such process
[1]+  Terminated              ./armored
```

Write `armored.c`.

[Solution]

A solution could involve either handling or ignoring the `SIGTERM` signal that the builtin kill sends. For instance, the signal handler could randomly decide whether to exit or return:

```
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void
```

```
sigterm_handler()
{
    if (random() % 5 == 0) {
        signal(SIGTERM, SIG_DFL);
        raise(SIGTERM);
    }
}

int
main()
{
    signal(SIGTERM, sigterm_handler);
    for (;;)
        sleep(1);
}
```

Or, the program could ignore SIGTERM for roughly 80% of the time:

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

int
main()
{
    for (;;) {
        signal(SIGTERM, SIG_IGN);
        sleep(4);
        signal(SIGTERM, SIG_DFL);
        sleep(1);
    }
}
```

Note that merely blocking SIGTERM to keep it temporarily at bay is not a solution:

```
#include <signal.h>
#include <unistd.h>
#include "signal_support.h"

int
main()
{
    for (;;) {
        signal_block(SIGTERM);
        sleep(4);
        signal_unblock(SIGTERM);
        sleep(1);
    }
}
```

in this program, the first SIGTERM signal sent will (eventually) lead to the termination with a 100% chance.

2 Randomly Corrupted Executables (10 pts)

Consider the following short C program which takes 3 command line arguments denoting the names of 2 files and an integer passed to the `srandom()` function. It will read the first file and create the second file with the content of the first one, except that it will randomly corrupt one out of approximately every 1024 bytes.

```c
// run as
// ./randomize_it <old> <new> <seed>
// e.g.
// ./randomize_it cush cushr 42
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>

int
main(int ac, char *av[])
{
    int from = open(av[1], O_RDONLY);
    int to = open(av[2], O_CREAT | O_TRUNC | O_WRONLY, 0777);
    int seed = atoi(av[3]);
    srandom(seed);
    char buf[1024]; // change one byte every 1024
    size_t bread;
    while ((bread = read(from, buf, sizeof buf)) > 0) {
        buf[random() % bread] = random();
        write(to, buf, bread);
    }
}
```

Imagine you applied this program to your `cush` executable from Project 1 to obtain a `cushr` executable with a seed you choose.

1. (4 pts) Discuss a possible outcome for what happens if you then tried to run the randomized executable `./cushr`

   Note: simply stating "segmentation fault" is not a sufficient answer: if the randomized executable encounters a segmentation fault, you must provide an explanation for why.

   [Solution] The following are examples of acceptable answers:

   - Nothing happens: all perturbations occur in areas of the binary that aren't actually used or are masked by subsequent parts of the execution
   - A segmentation fault occurs due to an illegal instruction when an instruction was changed

- A segmentation fault occurs due to an invalid memory access when a linker-provided address was changed
- A segmentation fault occurs due to a bus error stemming from a corrupted memory address
- Error during the dynamic linking process when metadata for the dynamic linker is corrupted
- Data corruption to variable or string values
- Control flow corruption that causes semantically-incorrect control flow given the program's inputs

It is important to discuss one specific error and its cause.

2. (2 pts) Apparently, the fact that we pose this question means that there is apparently little risk to others or to the integrity of the system as a whole in running such corrupted executables. Provide a brief explanation for why this is so.

   [solution] There is little risk to other processes and the operating system as a whole due to the process isolation provided by the combination of operating system and hardware primitives, e.g., CPU modes, virtual memory/MMU, and file system permissions. Whatever goes wrong with the corrupted executable will likely result in a localized failure that will affect only that process. You must point-out both virtual memory and file system permissions for full credit.

3. (2 pts) Even though the risk of harmful effects to others due to executing the randomly-modified program is small (i.e., unlikely in the common case), persistent effects are possible. Describe one potential negative, persistent, effect (however unlikely) of running the randomly-modified executable.

   [solution] It is possible—though unlikely—that such a small perturbation will result in code that makes system calls that create permanent changes in the environment. For instance, if the system call code for "open" were changed to "unlink" or similar (or the offset in the PLT), then users typing "a < file" into cush would delete `file` instead of reading from it. There are other "exploitable" system calls, e.g., system(), exec(). For a related read check out the academic paper, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks" from IEEE Security and Privacy 2016.

4. (2 pts) What effect does using different seeds have on the resulting randomized-binary, assuming the same input executable?

   [solution] The location and value of the changes change in a unpredictable fashion; each different seed resulting in a different program, potentially with a different execution behavior. It is important to connect the effect of changing the seed to the resulting randomized binary.

3 IPC via Pipes (12 pts)

3.1 Fork and Pipes (8 pts)

Consider the following program:

```c
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int
6  main()
7  {
8      int fd[2][2];
9      pipe(fd[0]);
10     pipe(fd[1]);
11
12     int pid = fork();
13     if (pid == 0) {
14         for (int i = 0; i < 5; i++) {
15             /* you may add code here */
16             printf("child\n");
17             /* you may add code here */
18         }
19     } else {
20         for (int i = 0; i < 5; i++) {
21             /* you may add code here */
22             printf("parent\n");
23             /* you may add code here */
24         }
25     }
26 }
```

1. Assuming that the terminal outputs entire lines atomically, and assuming that none of the system calls fail, how many possible outputs does this program have?

2. Edit the program by adding code only in some or all of the areas marked on lines 15, 17, 21, and 23 so that the *only* possible output becomes:

   ```
   parent
   child
   parent
   child
   parent
   child
   parent
   child
   parent
   child
   ```

   You may not change other areas of the program.

[Solution]

There are $\binom{10}{5} = 252$ possible outputs.

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

int
main()
{
    int fd[2][2];
    pipe(fd[0]);
    pipe(fd[1]);

    int pid = fork();
    if (pid == 0) {
        for (int i = 0; i < 5; i++) {
            char c = 0;
            read(fd[0][0], &c, 1);
            printf("child\n");      // wait for ping
            fflush(stdout);
            write(fd[1][1], &c, 1); // send pong
        }
    } else {
        for (int i = 0; i < 5; i++) {
            char c = 0;
            printf("parent\n");
            fflush(stdout);
            write(fd[0][1], &c, 1); // send ping
            read(fd[1][0], &c, 1);  // wait for pong
        }
    }
}
```

Flushing the standard output is necessary to ensure that the program works even when run using I/O redirection, as in `pipepc | cat`.

3.2 Optimizing Pipes (4 pts)

A CS3214 student working on the cush project came up with the idea of reusing pipes, thus making fewer system calls and using fewer OS resources. If given a command line such as `a | b | c | d | e` they would create 2 pipes, pipe 1 and 2, then link a and b with pipe 1, b and c with pipe 2, and then, since a and c are one removed from each other, connect c and d with pipe 1. Finally, pipe 2 connects d and e.

Discuss the feasibility of this approach. Justify your answer.

[Solution]

This approach is not feasible because the processes that are part of a pipe must be run concurrently. Because of that, there may be simultaneously data that has left process `a` to be read by process `b` and data that has left process `c` to be read by process `d`. These are different pieces of data that cannot be stored in the same pipe because pipes exploit only a single buffer in which to store data.

4 Linking (16 pts)

An executable is built from 4 separately compiled files called `file1.c` through `file4.c` and a single header file `defs.h`. These files are compiled with gcc 7.5.0 or later like so

```
$ gcc -c -Wall -Wmissing-prototypes file?.c
```

which does not cause any errors or warnings.

When the command `nm file?.o` is run, the following output is obtained:

```
file1.o:
                 U f4
0000000000000000 t fun
                 U g2
                 U g4
                 U _GLOBAL_OFFSET_TABLE_
0000000000000007 T main
0000000000000004 C v1

file2.o:
0000000000000000 T g2
0000000000000042 T h2
0000000000000000 b l2
                 U y

file3.o:
0000000000000007 t fun
0000000000000000 t g3
000000000000001d T h3
0000000000000000 b l2
                 U v1
0000000000000004 C x

file4.o:
0000000000000016 T f4
0000000000000036 T g4
0000000000000000 t h4
0000000000000000 b l2
0000000000000004 C x
0000000000000008 C y
0000000000000000 D z
```

1. (10 pts) Reconstruct `file1.c`, `file2.c`, `file3.c`, `file4.c`, and `defs.h` subject to the following conditions:

   - Each file contains one (and only one) `#include` directive of the form `#include "defs.h"`.
   - None of the `.c` files uses the `extern` keyword.
   - The files are compiled with `-Wall -Wmissing-prototypes`
   - It is possible to link the .o files without error by doing

     ```
     gcc file?.o
     ```

   - You may ignore the `_GLOBAL_OFFSET_TABLE_` symbol introduced.

   [Solution]

   A possible solution is shown here:

   `defs.h`

   ```
   int g2(void);
   int h3(void);
   void h2(void);
   void g4(void);
   void f4(void);
   extern int x;
   extern double y;
   extern int v1;
   extern int z;
   ```

   `file1.c`

   ```
   #include "defs.h"

   static void fun() { }
   int v1;

   int
   main()
   {
       fun();
       g2();
       f4();
       g4();
   }
   ```

   `file2.c`

   ```
   #include "defs.h"

   static int l2;
   ```

```
int g2()
{
    y++;
    l2++;
    h2();
    return y;
}

/* defect: h2 is not used elsewhere, should
 * be static */
void h2()
{
}
```

file3.c

```
#include "defs.h"

static void g3()
{
}

static int l2;
static void fun() { l2++; }

// violation of best practice: don't have
// multiple definitions of a common; see
//
int x;

int h3()
{
    g3();
    fun();
    return v1;
}
```

file4.c

```
#include "defs.h"

int x;
double y;

// violation of best practice: z is not accessed
// outside this module, should be static
int z = 44;
```

```
static int l2;
static void h4()
{
    z++;
}

void f4()
{
    l2++;
    h4();
}

void g4()
{
}
```

2. (6 pts) Assume that the symbols defined in this project represent useful and necessary variables and/or functions, and assume that this is a self-contained project (that is, not part of a library or in any way connected to other projects). Describe 3 best practices related to linking that were violated in this project. (Assume however that the use of global variables was unavoidable.)

   [Solution]

   - `file2.c` exports a global function `h2` that is not used elsewhere and should thus be static
   - `file3.c` exports a global function `h3` that is not used elsewhere and should thus be static
   - `file4.c` exports a global variable `z` that is not used elsewhere and should thus be static
   - `file3.c` and `file4.c` both provide definitions of common symbol `x` which violates best practice (flagged by `-Wl,--warn-common`)

   We graded part 1 with an automated grader that deducted for compiler warnings (or errors) and linker errors when compiled/linked as described in the specification.

   Note that I did not count "self-inflicted" outright errors introduced by a faulty reconstruction as bad practices. For instance, if you introduced an error where `v1` was defined as a function in one file and a variable in another, I did not count that.

5 Submission Requirements

Submit a tar file that contains the following files:

- `deep.c` with your answer to Question 1.3
- `armored.c` with your answer to Question 1.4
- `pipepc.c` with your answer to Question 3.1
- `file1.c`, `file2.c`, `file3.c`, `file4.c`, and `defs.h` for question 4.1 and `linkingbestpractices.txt` to answer Question 4.2

- A file `answers.txt` with your answers for questions 1.1, 1.2, Question 2 (parts 2.1 to 2.4), and Question 3.2.