CS 3214 Spring 2021 Test 1 Solution

October 7, 2021

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 material, and the textbook. Failure to do so is an Honor Code violation.

1 Processes and Kernels (14 pts)

1.1 Process States (4 pts)

At the beginning of the COVID lockdown, Dr. Back downloaded a 100MB large directory of high-resolution Zoom background images from the university website and added it to the Zoom client on his computer. Now, whenever he opens the Settings tab in Zoom and tries to go to the virtual background tab (where a preview of all the images will be shown), his Zoom client appears to stall. The User Interface (UI) becomes unresponsive for almost half a minute before the images finally appear in the UI. This very annoying situation got him thinking about process states.

1. (2 pts) While the Zoom client is unresponsive, what process state is it most likely in (RUNNING, READY, or BLOCKED)? Justify your hypothesis.

2. (2 pts) How could Dr. Back confirm or reject your hypothesis?

[Solution]

It's most likely in the RUNNING state (presumably busy running image processing code to create thumbnails on the fly for the large images).

Dr. Back could verify this by observing his system's CPU utilization, which should hover around 100% if the process was in the RUNNING state. This is, in fact, what happened and it also caused the CPU's fan to kick in, making noticeable noise.

If the CPU utilization had not been at 100%, the process would have been in the BLOCKED state. This could occur, hypothetically, if the program tried to repeatedly contact a remote server with very slow response times. (This is, however, unlikely here based on the information provided since Dr. Back had already downloaded the Zoom images into a local directory folder.)

1.2 User vs Kernel Mode (10 pts)

Matthew Garrett is a security engineer working for Google who recently submitted a patch for inclusion into the Linux kernel. The following abbreviated excerpt is directly quoted from Garrett's blog:

> *Making hibernation work under Linux Lockdown*
>
> Linux draws a distinction between code running in kernel (kernel space) and applications running in userland (user space). This is enforced at the hardware level - in x86–speak, kernel space code runs in ring 0 and user space code runs in ring 3. If you're running in ring 3 and you attempt to touch memory that's only accessible in ring 0, the hardware will raise a fault. No matter how privileged your ring 3 code, you don't get to touch ring 0.
>
> Kind of. In theory. Traditionally this wasn't well enforced. At the most basic level, since root can load kernel modules, you could just build a kernel module that performed any kernel modifications you wanted and then have root load it. Technically user space code wasn't modifying kernel space code, but the difference was pretty semantic rather than useful. But it got worse - root could also map memory ranges belonging to PCI devices, and if the device could perform DMA you could just ask the device to overwrite bits of the kernel. (...)

> It turns out that there were a number of ways root was effectively equivalent to ring 0, and the boundary was more about reliability (ie, a process running as root that ends up misbehaving should still only be able to crash itself rather than taking down the kernel with it) than security. After all, if you were root you could just replace the on-disk kernel with a backdoored one and reboot. Going deeper, you could replace the bootloader with one that automatically injected backdoors into a legitimate kernel image. We didn't have any way to prevent this sort of thing, so attempting to harden the root/kernel boundary wasn't especially interesting.
>
> In 2012 Microsoft started requiring vendors ship systems with UEFI Secure Boot, a firmware feature that allowed systems to refuse to boot anything without an appropriate signature. This not only enabled the creation of a system that drew a strong boundary between root and kernel, it arguably required one - what's the point of restricting what the firmware will stick in ring 0 if root can just throw more code in there afterwards? What ended up as the Lockdown Linux Security Module provides the tooling for this, blocking userspace interfaces that can be used to modify the kernel and enforcing that any modules have a trusted signature.

The rest of the article discusses the proposed patch, which ensures that Lockdown works even when users use hibernation to save a system's state and then power off the machine and then restore the state of a subsequent power on cycle. The full article can be found at `https://mjg59.dreamwidth.org/55845.html` for those interested.

Based on your reading of this excerpt, answer the following questions:

1. (2 pts) In the author's opinion, does having a privileged root user increase or decrease the level of security a system provides? Please provide a brief justification.

   [Solution] It decreases the level of security since under the existing Linux model, the root superuser can load kernel modules and use DMA that could subvert the existing dual mode protection mechanism.

2. (2 pts) Does the author endorse the argument that dual-mode operation increases robustness because it can provide an additional line of defense against bugs in kernel code? Please provide a brief justification.

   [Solution] No. Dual-mode operation does not defend against bugs in kernel code since kernel code runs in kernel mode. When the author says that processes running as root that end up misbehaving crash themselves rather than taking down the kernel they are referring to processes running in user mode, not kernel code.

3. (6 pts) How important is hardening the "root/kernel boundary" in your personal opinion for a personal computer that you own and exclusively use? (Note: this question asks for an opinion justified by facts. You are not asked to produce a balanced argument—take a position and argue for that position!) Please provide your answer in 200 words or less.

   [Solution] There is a long history behind this feature, see [1], [2], [3]. The proponent argument is laid out by Garrett, and argues that having a chain of trust (in which the firmware loads only a signed kernel, a signed kernel loads only signed modules, etc.) makes it harder for an attacker who has compromised the root account to become established in a system in a manner that survives a reboot.

The opposing argument is largely that the lockdown idea is not compatible with all hardware and may not work with applications that expect direct access to the hardware, and that it also protects only against a particular type of threat - after all, an attacker that has gained root privileges can often take advantage of a machine already without needing to compromise the kernel. For instance, they can read all files stored on the computer already. There is also some concern that this technology could lead to a loss of control of what owners of hardware are allowed to do with it; although in its current implementation lockdown can be turned off with a keystroke on a physically attached keyboard.

2 Elegant `bash` Conditionals (10 pts)

In a post that was recently circulated on HackerNews, developer Tim Visée shared their knowledge of bash's `&&` and `||` control operators[1].

The post discusses how bash allows you to express a logical "and" between two commands using the `&&` operator as follows:

```
$ test -f x && echo file x exists
```

If (and only if) a file by the name 'x' exists, the 'echo' command will be run. Other examples include:

```
$ test 3 = 3 && echo three is equal to three
three is equal to three
$ test 3 != 2 && echo three is not equal to two
three is not equal to two
$ test 3 != 3 && echo three is not equal to three
$
```

Note that `test` is not a shell builtin, but a Unix command. Type `man test` to see its man page [URL].

For this problem, implement a program called `and` that accepts two commands and runs the second one only if the first one succeeded. A command succeeds if (and only if!) it calls the `exit()` system call with the value `EXIT_SUCCESS`.

Your program should allow multiple command line arguments to be passed to both the first and the second command. The 2 sets are separated by a double hyphen `--` which may not appear in the list of the command line arguments to either program.

Using the `and` command, you could say:

```
$ ./and test 3 = 3 -- echo three is equal to three
three is equal to three
$ ./and test 3 != 2 -- echo three is not equal to two
three is not equal to two
$ ./and test 3 != 3 -- echo three is not equal to three
$
```

[1] `https://timvisee.com/blog/elegant-bash-conditionals/`

and so on. For the first invocation, the first command is `test` and its arguments are `"3"`, `"="`, and `"3"`, whereas the second command is `echo` and its arguments are `"three"`, `"is"`, `"equal"`, `"to"`, and `"three"`.

Your program can be kept short; error handling is not required (in other words, you may assume that both the first and second command are given and exist in the user's path, and that the command line given to `and` contains both a first and second command and exactly one double-hyphen separating the first and the second command's arguments. However, your program should not make any assumptions about how many arguments are passed to the first or second command. Your program should also not make any assumptions regarding anything else about the first or second command.

Hint: an implementation may exploit the fact that a program's own `argv[]` array is writable.

[Solution] (shown here without error checking for brevity)

The shortest solution is

```c
// and.c reusing parent process for 2nd conditional
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int
main(int ac, char *av[])
{
    char **sep = av;
    while (*sep && strcmp(*sep, "--") != 0)
        sep++;
    *sep++ = NULL;

    if (fork() == 0) {
        execvp(av[1], av+1);
    } else {
        int status;
        wait(&status);
        if (WIFEXITED(status) && WEXITSTATUS(status) == EXIT_SUCCESS)
            execvp(sep[0], sep);
        return WEXITSTATUS(status);
    }
}
```

but using a separate fork for the second conditional is ok, too:

```c
// and.c solution
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

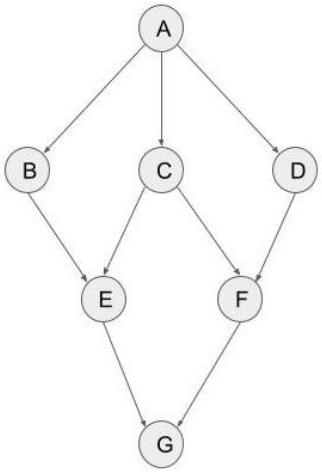
```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int
main(int ac, char *av[])
{
    char **sep = av;
    while (*sep && strcmp(*sep, "--") != 0)
        sep++;
    *sep++ = NULL;

    if (fork() == 0) {
        execvp(av[1], av+1);
    } else {
        int status;
        wait(&status);
        if (WIFEXITED(status) && WEXITSTATUS(status) == EXIT_SUCCESS) {
            if (fork() == 0)
                execvp(sep[0], sep);
            wait(&status);
            return WEXITSTATUS(status);
        }
        return WEXITSTATUS(status);
    }
}
```

3 MapReduce (20 pts)

MapReduce is a widely-used programming model for data processing for modern applications such as business analytics and deep learning. A powerful aspect of the MapReduce model is that it can distribute information across many computers and this achieves scale. This is desirable, because to solve a bigger problem, all you have to do is add more computers.

In this question, we will try to learn some of the basics of the MapReduce model, especially how it distributes data between different computers. Consider the following graph, where nodes A...G represent different computers. For this question, assume each of the nodes is a process (created with our friend, the `fork()` call). Data is sent to A, which then distributes, i.e., maps in the MapReduce vocabulary, the data to the nodes in the next stage, here (B, C, D). Each of the nodes then processes the incoming data, and further gathers, i.e., reduces, the data to the next stage, here represented by nodes (E, F). In real MapReduce systems, this reduction may include sorting of data etc., but you can assume it to be a simple processing step for the purposes of this problem. Finally, the last node (G) combines and merges all of the data and outputs the results. This completes the processing cycle. Do note that any of the stages (B, C, D) or (E, F) could be scaled by adding more nodes to it. Supporting such scaling is however not required for this question.

1. **(12 pts)** You will write a program `mapreduce.c` that implements the basic data flow rep-



resented by the above figure using Unix pipes. Your program acting as the parent will fork processes that represent nodes A. . . G. You will create pipes to enable communication between the processes as shown by the arrows, e.g., main process to A, A-B, A-C, A-D, etc. Once this is done, your main process will send to A via IPC a sequence of numbers $0, 1, 2, \ldots, 99$. A, upon receiving these numbers, will add its name (A) to the front (left) of the number and pass it to (B, C, D) in a round robin fashion. For example, 0 will become A0 and then sent to B, 1 goes to C as A1, and 2 to D as A2, and so on. The receiving nodes (B, C, D) will prepend their own names to the input and pass in on to the next node in the flow graph shown in the figure. For simplicity, nodes with multiple input sources (E, F, G) process data sequentially from their sources, starting from the left to the right as shown in the diagram. E.g, E will read all of the data from B and then from C. Similarly, F will read data first from C and then from D. Consider the example of data item 0: It goes from 0 to A0 to BA0 to EBA0 to GEBA0. All downward flow will use round-robin distribution where needed. Finally, G will print all of the output data to `stdout`, one data item on each line.

[Solution] A solution is shown below. Note that you were not required to support a variable number of tasks although the code shown below does support it.

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

#define PROCS 7
#define NPIPES 10
#define TOTAL 100
#define MSG_SIZE 20
```

```
static int pip[NPIPES][2]; //driver-A, A-B, A-C, A-D, B-E, C-E, C-F, D-F, E-G, F-G
static int a_to_b, a_to_c, a_to_d, c_to_e, c_to_f, total;

/* Read `total` messages from read end of pipe pair in `fromfd` and
 * round-robin distribute them to pipes at `(tofd + i) % stride`
 *
 * For simplicity, we send always 20-byte messages padded with 00 bytes.
 * If `pad` is false, we do not pad.
 */
static void process(const char *name, int fromfd[2], int tofd[][2],
                    int total, int stride, bool pad)
{
  char msgi[MSG_SIZE], msgo[MSG_SIZE];

  for (int count = 0; count < total; count++) {
    read(fromfd[0], msgi, MSG_SIZE);
    snprintf(msgo, sizeof msgo, "%s%s", name, msgi);
    write(tofd[count%stride][1], msgo, pad ? MSG_SIZE : strlen(msgo));
  }
}

static int divceil(int x, int y)
{
  return (x + y - 1) / y;
}

static void task_a()
{
  process("A", pip[0], &pip[1], total, 3, true);
}

static void task_b()
{
  process("B", pip[1], &pip[4], a_to_b, 1, true);
}

static void task_c()
{
  process("C", pip[2], &pip[5], a_to_c, 2, true);
}

static void task_d()
{
  process("D", pip[3], &pip[7], a_to_d, 1, true);
}
```

```
static void task_e()
{
  process("E", pip[4], &pip[8], a_to_b, 1, true);
  process("E", pip[5], &pip[8], c_to_e, 1, true);
}

static void task_f()
{
  process("F", pip[6], &pip[9], c_to_f, 1, true);
  process("F", pip[7], &pip[9], a_to_d, 1, true);
}

static void task_g()
{
  int stdio[][2] = { { 0, 1 } };
  process("G", pip[8], stdio, a_to_b + c_to_e, 1, false);
  process("G", pip[9], stdio, c_to_f + a_to_d, 1, false);
}

int
main(int ac, char *av[])
{
  // compute how many messages will be sent
  total = ac > 1 ? atoi(av[1]) : TOTAL;
  a_to_b = divceil(total, 3);
  a_to_c = divceil(total-1, 3);
  a_to_d = divceil(total-2, 3);
  c_to_e = divceil(a_to_c, 2);
  c_to_f = divceil(a_to_c-1, 2);

  // creating all the pipes
  for (int i = 0; i < NPIPES; i++)
    if (pipe(pip[i])<0) {
      fprintf(stderr, "Pipe creation error\n");
      return EXIT_FAILURE;
    }

  // let's create the processes next
  for (int i = 0; i < PROCS; i++)
    if ((fork()) == 0) {
      switch ('A' + i) {
        case 'A': task_a(); break;
        case 'B': task_b(); break;
        case 'C': task_c(); break;
        case 'D': task_d(); break;
        case 'E': task_e(); break;
```

```
        case 'F': task_f(); break;
        case 'G': task_g(); break;
      }
      exit(EXIT_SUCCESS);
    }

  // send numbers to A process
  for (int i = 0; i < total; i++) {
    char msg[MSG_SIZE];
    snprintf(msg, sizeof msg, "%d\n", i);
    write(pip[0][1], msg, MSG_SIZE);
  }

  for (int i = 0; i < PROCS; i++)
    wait(NULL);
}
```

2. **(2 pts)** From the output of the program, you see that different data elements take different paths. List all the possible paths that you observe. Then for each path, provide a count of the number of data elements that took that path.

   [Solution] The possible data paths and the number of elements that are processed through those paths are as follows:

   ```
   GEBA, 34
   GECA, 17
   GFCA, 16
   GFDA, 33
   ```

3. **(2 pts)** Execute the program 10 times. Is the output deterministic across the multiple runs? Is the actual order in which the reduce tasks are executed deterministic?

   [Solution] The output is deterministic across the multiple runs. This is because IPC order in this case is fixed and same across multiple runs.

   The order in which individual reduce tasks execute is non-deterministic.

4. **(2 pts)** In writing our program, we employed the round-robin policy when distributing data to multiple nodes. Why do you think that is useful for an actual (not modeled) MapReduce system?

   [Solution] The use of round-robin policy evenly distributes the workload of processing the data items across all of the available reducers. It also makes integrating more reducers to the data processing easy. Even load distribution, or load balancing, further helps by making sure a stage is completed quickly. For instance, if B received 50% of the work, but C and D only 25%, B will take much longer than C and D, and the next stage (E, F) would have to wait for B to complete. Round robin help mitigate such problems.

   (Additional information: It is important to understand that round robin is preferable only if all items need same kind of processing and all nodes are identical in computing capabilities. In heterogeneous systems, more advanced policies would need to be developed.)

5. **(2 pts)** If we add one more node each to the stage (B, C, D), and a (E, F), what changes would it entail in your program? Is it easy to change your program? Justify your opinion.

   [Solution] The answer here depends on how you implemented your program.

   For most, the program would have adopted a rigid communication pattern as dictated by the graph. In this case, adding additional nodes would require creating more pipes, setting up new pattern of communication, changing the distribution of data from 3 to 4 nodes in (B, C, D, N1) stage, updating distribution of data from D to (F, N2), and updating reading of data at G from (E, F, N2). All of this makes such changes complex, and while coding effort can be reduced, such a change in a rigid program requires rethinking each step. This means that the program is not easy to change. **This is the answer we expected from most students.**

   However, another way to write the program is to capture the graph as an appropriate data structure, e.g., a nxn matrix of nodes with a 1 representing a connection between nodes and 0 absence of it. Then a matrix parser could be written to setup the communication pattern automatically, and adding more nodes simply means redefining the matrix. In this case, the program is trivial to change. In real MapReduce systems, a meta-scheduler keeps track of the mappers and reducers and handle asynchronous communication between them. There adding a node at any stage is as easy as changing the number of mappers or reducers. **This answer could have come from students who are already aware of map reduce systems or who could not finish their implementation and offered a qualitative response.**

4 Linking (16 pts)

4.1 A Linker Puzzle (10 pts)

After implementing the link checker in exercise 2, a student applied their implementation to a small project consisting of 4 files

- module1.c, compiled to module1.o
- module2.c, compiled to module2.o
- module3.c, compiled to module3.o
- lib.c, compiled to lib.o

using `gcc -Wall -Wmissing-prototypes -c module*.c lib.c`. After obtaining the symbol tables of each .o file using

```
for l in module*o lib.o
do
  nm -S $l > `basename $l .o`.nm
done
```

the link checker[2] reported this:

[2]The output format was slightly changed from the actual exercise, for instance, it now reports how many global symbols a library exports. The output was reformatted for readability.

```
Weak symbol `buf' overridden, defined in module2 and overridden as D in module3
Global symbol `DESCRIPTION' defined in `module1` is not referenced by any other file,
    should be static
Global symbol `data' defined in `module2` is not referenced by any other file, should be static
The 2 global symbols exported by library `lib' share common prefix lib_fun
```

The link-checker was run with: `link-checker.py -l lib.nm module?.nm` The `-l` switch here denotes that the link checker exempts `lib.o` from the requirement that its global symbols are referenced in other compilation units, and it also asks the link checker that a consistent prefix is used for all exported symbols.

The program was then linked with `gcc module?.o lib.o -o main`. The symbol table of the `main` executable contained the following symbols

```
0000000000201020 D buf
0000000000201018 D data
0000000000201010 D DESCRIPTION
000000000000065f T lib_fun1
0000000000000666 T lib_fun2
000000000000061a T main
0000000000000636 T module2_fun
000000000000062f t module2_fun2
000000000000064e T module3_fun
0000000000000647 t module3_fun2
```

where internal symbols introduced by the compiler and runtime library are omitted.

Your task is to reconstruct `module1.c` to `module3.c` and `lib.c`, along with a header file `defs.h` that must be included by all 4 .c files.

Any reconstruction will be accepted that meets the following requirements:

- The modules must compile with the above flags on gcc 8 or later
- The modules must link and produce an executable without linker error
- The symbols listed above must occur in the symbol table of the executable
- All declarations of global nonstatic functions must be contained in `defs.h`
- The link checker should not report errors other than those shown.

[Solution] A possible reconstruction is

defs.h

```c
void lib_fun1(void);
void lib_fun2(void);
void module2_fun(void);
void module3_fun(void);
```

module1.c

```c
#include "defs.h"

const char * DESCRIPTION = "main";

int
main()
{
    module2_fun();
    module3_fun();
}
```

module2.c

```c
#include "defs.h"

char buf[8];
char data = 42;

static void module2_fun2()
{
}

void module2_fun()
{
    module2_fun2();
}
```

module3.c

```c
#include "defs.h"

char buf[8] = "abcdefgh";

static void module3_fun2()
{
}

void module3_fun()
{
    module3_fun2();
}
```

lib.c

```c
#include "defs.h"

void lib_fun1()
{
}
```

```
void lib_fun2()
{
}
```

4.2 A Linking Accident (6 pts)

Consider the following program contained in a file `link2.c`

```
// link2.c
#include <stdio.h>
#include <stdlib.h>

_Bool even;

int
main(int ac, char *av[])
{
    int a = atoi(av[1]);
    even = a % 2 == 0;
    printf("even %d\n", even);
}
```

When compiled with `gcc -Wall link2.c -o main` the resulting `main` executable can be run with `./main 42` and produces the answer `even 1`.

When a programmer used the same code as part of a larger project that also contained a file `link1.c` they built the program using `gcc -Wall link1.c link2.c -o main2`. Here is `link1.c`:

```
// link1.c
int even(int a)
{
    return a % 2 == 0;
}
```

When running the new executable `main2`, the shell prints

```
$ ./main2 42
Segmentation fault (core dumped)
```

Answer the following questions:

1. (2 pts) Explain why the compiler successfully compiled this source code without flagging any errors related to `even`.

   [Solution] Because the compiler compiles the 2 files separately, and because the definition and use of `even` are well-formed in each source file (in link2, as a global variable of type `_Bool`, in link1 as a global function), there is no defect or violation that the compiler could determine. Taking individually, both source code files are perfectly fine.

2. (2 pts) Explain why the linker successfully linked the executable without flagging any errors (if invoked as above), and discuss which rule it may have applied.

   [Solution] The linker sees a `T even` symbol in link1 and a `C even` symbol in link2 and applies the rule that strong symbols override (hide) weak symbols. Thus, the address of the symbol `even` becomes the address at which function `even` is located in the program's text segment.

3. (2 pts) Provide the exact reason (and explanation of) why the program crashed when we attempt to run it.

   [Solution] Since `even` has been resolved to the address of function `even`, writing a value to this address in the assignment `even = a % 2 == 0` causes a segmentation fault - the text segment (where code such as the function `even` is located) is mapped read-only. Attempts to write to it therefore fail since the program lacks permission to write to this address.