CS 3214 Fall 2021 Test 1 Solution

October 11, 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. This includes sites such as chegg.com, which will be monitored. The open Internet stipulation does not apply to such sites.
- 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 Basic OS Functions (12 pts)

Find out if the following statements related to operating systems are true or false. If true, just write **true**. If false, write **false** and provide a corrected statement that includes a concise explanation of why the original statement was false.

1. A process encapsulates an instance of a running program.

   True.

2. Processes obtain services from the kernel through system calls.

   True.

3. Processes can temporarily disable the CPU's ability to receive interrupts in order to prevent interference from the OS.

   False. Disabling interrupts is a privileged operation user processes are prevented from doing; the OS must always retain the ability to receive and process interrupts to preempt processes.

4. When the kernel encounters an exception due to a bug, the OS can typically recover and spawn a new kernel to continue.

   False. The kernel is underlying control program - if it encounters an exception due to bug, the OS can typically not recover (for instance, in Windows, the "blue screen of death" results). Spawning a new kernel is tantamount to restarting the machine (and loss of all currently running processes), so it is not "recovery" that would allow continuation of use. Even kernels like Linux that "tolerate" certain exceptions in the kernel ("oops") do not respawn a new kernel and, in general, cannot guarantee recovery.

5. OS kernels use file descriptors or handles to provide processes with indirect access to resources such as files.

   True.

6. Multiple processes running on a computer typically share the same virtual address space.

   False. Each process has their own, separate virtual address space.

2 Flexible Binaries (8 pts)

Unix comes with thousands of utilities that can be combined in a myriad of ways. Normally, these are in separate binaries which the user's shell invokes. However, it is possible to combine some of them into what we will call flexible binaries.

In this question, you are asked to implement a flexible binary that combines the function of the standard `echo` and `sleep` utilities.

Write a file `flexible.c` such that, when it is compiled via `gcc -o flexible -Wall -Werror flexible.c`, the user can create symbolic links like so:

```
$ ln -s flexible flexecho
$ ln -s flexible flexsleep
```

Then, if the user types `flexecho` or `flexsleep`, the program acts like `echo(1)` or `sleep(1)`, respectively.

```
$ ./flexecho acts just like echo
acts just like echo
$ ./flexsleep 5
(no output here, but 5 seconds pass)
$
```

You do not need to implement any command line options for either utility, and you may assume that the user invokes them in the intended way (no error handling is required).

You will need to make sure, however, that the flexible binary can be invoked using a relative or absolute pathname, i.e., `./flexecho` or `/home/ugrads/you/flexecho` both must work, depending on where the executable is located. In other words, your implementation should not assume that the current directory is in the user's `PATH`.

Hint: You may use the `basename(3)` function in your implementation, see https://man7.org/linux/man-pages/man3/basename.3.html.

[Solution]

"Flexible" binaries are actually called multibinaries. They are frequently used in small, embedded systems such as BusyBox. They rely on examination of `argv[0]` passed to main.

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

int
main(int ac, char *av[])
{
    char *base = basename(av[0]);
    if (!strcmp(base, "flexecho")) {
        for (int i = 1; i < ac; i++)
            printf("%s%c", av[i], i == ac-1 ? '\n' : ' ');
    } else
    if (!strcmp(base, "flexsleep")) {
        sleep(atoi(av[1]));
    }
}
```

3 Mystery Tool (8 pts)

As you noticed in the last problem, many Unix utilities (or at least expository versions of them) can be written in a few lines of code.

Here is one example, which unfortunately lacks comments describing its function.

```c
#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[])
{
    if (fork() == 0) {
        execvp(av[1], av+1);
        exit(EXIT_FAILURE);
    } else {
        int status;
        wait(&status);
        exit(!WEXITSTATUS(status));
    }
}
```

Your task in this problem is to reverse-engineer this program and provide a brief description as to how it works, and how it could be used in a useful manner (give an example). Include in your description an appropriate name for this utility.

[Solution]

An appropriate name for this utility would be `not`. When invoked with `not cmd arg1 arg2 arg3 ...`, it will run `cmd arg1 arg2 arg3 ...` in a child process, passing the arguments it has received to the child process. When the child process exits, it examines its exit status and exits itself with its negation.

Example use

```
$ ./not test 1 = 1; echo $?
1
$ ./not test 0 = 1; echo $?
0
```

or in combination with a logical shell operator

```
$ ./not test 0 = 1 && echo 'cmd returned true'
cmd returned true
$ ./not test 1 = 1 || echo 'cmd returned false'
cmd returned false
```

Note that an exit status of 0 signals a logical true since `EXIT_SUCCESS` is defined to be 0.

As a matter of fact, our rlogin machines have a program `/usr/bin/not` installed which provides exactly this functionality. It is not a standard utility, however.

4 Help, My Program Got Stuck (10 pts)

Developers often encounter situations in which a program they run appears to get "stuck." We shall define "getting stuck" as an apparent and indefinite lack of progress—the program has stopped outputting anything to its standard output and also has not exited or crashed, and in the absence of user intervention will stay stuck for the foreseeable future.

1. (2 pts) Suggest one (of many possible) diagnostic steps a developer can take when they experience a program "getting stuck," which will help them find out the root cause of the lack of progress.

   [Answer:] Possible steps include:

   - attaching strace (via strace -p) to determine if the process is blocked in an unfinished system call
   - attaching gdb (via gdb -p) and interrupting the process to find out to where in its code it has made progress.

2. (3 pts) Write a program in a language of your choice that gets stuck and that is in the BLOCKED state when getting stuck.

   [Answer:] Numerous answers are possible: any system call that cannot complete immediately, such as

   - `pause()`
   - `sleep(1000)`
   - `read(0, ...)`

3. (3 pts) Write a program that gets stuck but is in the READY or RUNNING state when getting stuck.

   [Answer:] To be in the READY or RUNNING state, the process needs to be perform continuous execution. The simplest way to accomplish this is via some kind of infinite loop that does not execute any blocking calls.

4. (2 pts) Why can't we ask you to write a program that is guaranteed to be in the READY state when getting stuck?

   [Answer:] A work-conserving scheduler will immediately schedule processes that are in the READY state as soon as a processor is available, thus putting the process in the RUNNING state. Conversely, a preemptive scheduler may preempt a RUNNING process's access to the CPU at any time, putting it back into the READY state. Thus, the process may switch between those states, which is outside the programmer's control.

The terms BLOCKED, READY, and RUNNING refer to the simplified process state diagram discussed in the lectures.

You **may not** use programs provided elsewhere in this test as one of your examples.

5 Clogged Pipes (10 pts)

Consider the following program in which error handling is omitted for brevity.

```c
#define _GNU_SOURCE
#include <fcntl.h>
#include <unistd.h>
#include <stddef.h>
#include <sys/wait.h>

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

static pid_t
run_process(char *exe, char *arg1, int std_in, int std_out)
{
    pid_t child = fork();
    if (child == 0) {       // fork child process
        char *argv[] = { exe, arg1, NULL };
        if (std_in != -1)   // redirect stdin
            dup2(std_in, STDIN_FILENO);
        if (std_out != -1)  // redirect stdout
            dup2(std_out, STDOUT_FILENO);
        execvp(exe, argv);
    }
    waitpid(child, NULL, 0); // wait for child
    return child;
}

int
main(int ac, char *av[])
{
    int fd[2];
    pipe2(fd, O_CLOEXEC);
    run_process("cat", av[1], -1, fd[WRITE_END]);
    close(fd[WRITE_END]);   // close parent's write end
    run_process("wc", "-m", fd[READ_END], -1);
    close(fd[READ_END]);    // close parent's read end
}
```

This program is compiled with `gcc -Wall -o piping piping.c`. When run like so:

```
$ ./piping piping.c
909
```

the program outputs a result, the number of characters in the source file.

However, when run like so:

```
$ ./piping /usr/share/dict/words
```

the program appears to not finish, it gets "stuck."

1. (4 pts) Explain why this program finished when run in the first way but didn't finish when run the second way.

   [Solution:] The first time, the amount of data the `cat` program wrote into the pipe was smaller than the pipe's internal buffer, allowing the write to complete and `cat` to exit. Thus, the piping program would return from its `waitpid` call and fork and wait for the second process running `wc`.

   However, when faced with a large file, `cat` was unable to write the full amount of data into the pipe since it exceeded the size of the pipe buffer. At this point, the process entered the BLOCKED state. Since the piping program waited for the first child process before spawning the second second, it remained in the BLOCKED state as well, effectively causing deadlock.

2. (6 pts) Repair the program so that it completes successfully, i.e.,

   ```
   $ ./piping-repaired /usr/share/dict/words
   4953680
   ```

   [Solution:]

   The solution is to start both processes first so that they can run concurrently, and then wait for each of them. This code performs the same system calls as the original code but in a different order.

   ```c
   #define _GNU_SOURCE
   #include <fcntl.h>
   #include <unistd.h>
   #include <stddef.h>
   #include <sys/wait.h>

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

   static pid_t
   run_process(char *exe, char *arg1, int std_in, int std_out)
   {
       pid_t child = fork();
       if (child == 0) {       // fork child process
           char *argv[] = { exe, arg1, NULL };
           if (std_in != -1)   // redirect stdin
               dup2(std_in, STDIN_FILENO);
           if (std_out != -1)  // redirect stdout
               dup2(std_out, STDOUT_FILENO);
           execvp(exe, argv);
       }
   ```

```
    return child;
}

int
main(int ac, char *av[])
{
    int fd[2];
    pipe2(fd, O_CLOEXEC);
    pid_t child1 = run_process("cat", av[1], -1, fd[WRITE_END]);
    close(fd[WRITE_END]);   // close parent's write end
    pid_t child2 = run_process("wc", "-m", fd[READ_END], -1);
    close(fd[READ_END]);    // close parent's read end
    waitpid(child1, NULL, 0); // wait for child 1
    waitpid(child2, NULL, 0); // wait for child 2
}
```

Note: you should not write a program that performs the functionality of counting the number of characters in a file; you should make the minimum amount of changes to the given program to make it complete and perform its functionality in the intended way. The repaired program must send its data through a pipe. This can be accomplished by rearranging the system calls made by the program without adding new ones or removing any.

6 Linking (16 pts)

6.1 A Linker Puzzle (10 pts)

Three files are involved in a build process: `moduleA.c`, `moduleB.c`, and `headerfile.h`.

The following is known:

- When compiled with

  ```
  gcc -c -Wall -Wmissing-prototypes moduleA.c
  gcc -c -Wall -Wmissing-prototypes moduleB.c
  ```

  there are no warnings. (gcc 8.5.0 is used.)

- Both `moduleA.c` and `moduleB.c` include `headerfile.h`.

- There are no definitions or declarations contained in both `moduleA.c` and `moduleB.c` that are fully identical.

- When linked with

  ```
  gcc -Wall -Wmissing-prototypes moduleA.c moduleB.c -o exe
  ```

  we see no errors.

- When we extract the symbol table from the executable `exe` we see among others the following symbols:

```
0000000000400583 0000000000000020 T fun1
00000000004005a3 0000000000000011 T fun2
000000000040053d 0000000000000007 T fun3
0000000000400536 0000000000000007 t funstatic
000000000040057c 0000000000000007 t funstatic
0000000000400544 0000000000000038 T main
00000000004006a8 0000000000000004 R consti
0000000000601030 0000000000000004 B varc
000000000060101c 0000000000000004 D varg
0000000000601020 0000000000000004 d vars
000000000060102c 0000000000000004 b vars
0000000000601024 0000000000000004 D varsh
                                   U puts@@GLIBC_2.2.5
```

- After collecting the symbol tables of `moduleA.o` and `moduleB.o` into files `moduleA.nm` and `moduleB.nm` we run these through the link checker implemented in exercise 2 like so:

  ```
  link-checker.py moduleA.nm moduleB.nm
  ```

  and see the following issues flagged:

  - `Common symbol varc multiply defined, first in moduleA and now in moduleB`
  - `Static function funstatic of size 0x7 appears in both moduleA and moduleB, check for inlining`
  - `Global symbol consti defined in moduleA is not referenced by any other file, should be static`
  - `Global symbol fun3 defined in moduleA is not referenced by any other file, should be static`
  - `Global symbol varg defined in moduleA is not referenced by any other file, should be static`
  - `Global symbol fun2 defined in moduleB is not referenced by any other file, should be static`

Reconstruct `moduleA.c`, `moduleB.c`, and `headerfile.h`!

If there are multiple reconstructions that meet the conditions described, any of them will be accepted. The addresses and sizes of the symbols do not need to match the ones shown, but the type and scope must.

[Solution:] A possible reconstruction is shown below:

moduleA.c

```c
#include <stdio.h>
#include "headerfile.h"

int varg = 5;
static int vars = 3;
const int consti = 4;
```

```
void fun3() { }

int
main()
{
    fun1();
    funstatic();
    vars++;
    varsh++;
    printf("Pull in libc\n");
}
```

moduleB.c

```
#include "headerfile.h"

static int vars;
int varsh = 4;

void fun1() {
    funstatic();
    vars++;
}

void fun2() {
    funstatic();
}
```

headerfile.h

```
extern void fun1(void);
extern void fun2(void);
extern void fun3(void);

static void funstatic()
{
}

int varc;
extern int varsh;
extern const int consti;
```

6.2 Undefined Behavior (6 pts)

Consider the following 2 C source files. The first one is `ub.c`:

```
1 #include <stdio.h>
2
```

```c
int a[12] = { 0 };

extern void printb();
int
main()
{
    printb();
    a[12] = 42;
    printb();
}
```

and the second one is `ub2.c`:

```c
#include <stdio.h>

static int b;

void printb()
{
    printf("%d\n", b);
}
```

When compiling with `gcc ub.c ub2.c -o ub` we obtain an executable `ub`.

1. (2 pts) As you can see, this C program exhibits undefined behavior. What causes the undefined behavior?

   [Solution:] The access to `a[12]` exceeds the array's bounds, which are `a[0..11]`.

2. (2 pts) Although we know that the compiler and/or executable is, per the C standard, not bound by any rules regarding what to do in the presence of undefined behavior, knowledge of compilers and linkers can sometimes provide an after-the-fact explanation of the actual behavior observed in the presence of potentially undefined behavior. Here, when the program is run we may observe:

   ```
   $ ./ub
   0
   42
   ```

   Based on your knowledge of compilers and linkers, explain how this output may have come about.

   [Solution:] It appears that the linker allocated the variable `b` right after and adjacent to `a` in virtual memory so that the virtual address of where `a[12]` would be (if it existed) coincided with `b`'s virtual address, In other words, `&a[12] == a + 12 == &b`.

3. (2 pts) If the program is compiled with optimization level 2, as in `gcc -O2 ub.c ub2.c -o ub2` the output is

```
$ ./ub2
0
0
```

Based on your knowledge of compilers and linkers, explain how this output may have come about.

[Solution:] Since `b` was declared static, the compiler could conclude during the optimization pass that it was never updated and thus replaced all uses of `b` with its value, which is 0.