CS 3214 Spring 2022 Test 1 Solution

March 4, 2022

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.
- If you have a question about the exam, you may post it as a *private* question on Discourse addressing the instructors (account names: Godmar_Back and djwillia and Liting_Hu). To do so, choose New Topic and then click the plus sign and select New message. If your question is of interest to others, we will make it public as a clarification and tag it with the `test1` tag.

- Any errata to this exam will be published prior to 12h before the deadline on the website and as announcements on the Discourse website. Otherwise, the Discourse forum will be closed during the exam.

1 Operating Systems (20 pts)

1.1 OS/Unix Facts (5 pts)

True/False questions. Find out if the following statements related to operating systems and/or Unix 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. The operating system is not responsible for resource allocation between competing processes.

   False. Managing resource allocation is a core responsibility of an OS, along with providing abstractions for processes and isolation.

2. An interrupt table is a kernel-internal data structure that contains the addresses of the handlers for the various interrupts needed to handle a system's physical devices.

   True. (User processes cannot access this table, but the kernel must manage it.)

3. A user-level process needs to involve the kernel to change the address translation information in the memory management unit (MMU) even when making changes to its own address space.

   True. (User processes cannot manipulate sensitive parts of the hardware directly, which includes the MMU.)

4. When a non-shell Unix process is killed with a SIGKILL signal that is sent to the process's process group, all of its descendants are typically killed as well.

   True. (Descendants typically inherit the parent's process group, so they stay in the same process group and thus are killed as well. An exception are shells or similar control programs that create new process groups for their children.)

5. A Unix kernel guarantees that data will be read in the same order from the "read" end of a Unix pipe as it was originally written to the "write" end of the pipe.

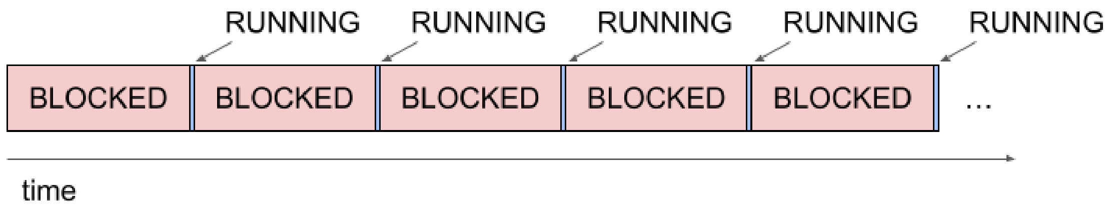
   True. Unix pipes provide in-order transmission.

1.2 Understanding Process States (10 pts)

As we discussed in lecture, operating systems model the state of the processes they run using a state model, which in its simplified form places active processes in one of three states: `READY`, `RUNNING`, or `BLOCKED`.

Suppose you had a tool that could record which state a process is in and plot this over time.

1. Consider the following timeline



(a) (3 pts) Write a program, in a programming language of your choice, that could have produced this timeline when executed on an OS.

Any program that blocks for some time, then repeats, will do. Simplest is probably using sleep() (or timing out on other timed event functions while no event is delivered.)

[Solution.]

```
#include <unistd.h>

int
main()
{
    while (1)
        sleep(1);
}
```

(b) (2 pts) The bash `time` builtin provides an overview of how much time a process took to complete (`real`), along with how much CPU time the process spent in user mode (`user`) or kernel mode (`sys`). For instance, when I run `time echo CS3214` the echo program completes near instantaneously and the output would be:

```
$ time echo CS3214
CS3214

real   0m0.000s
user   0m0.000s
sys    0m0.000s
```

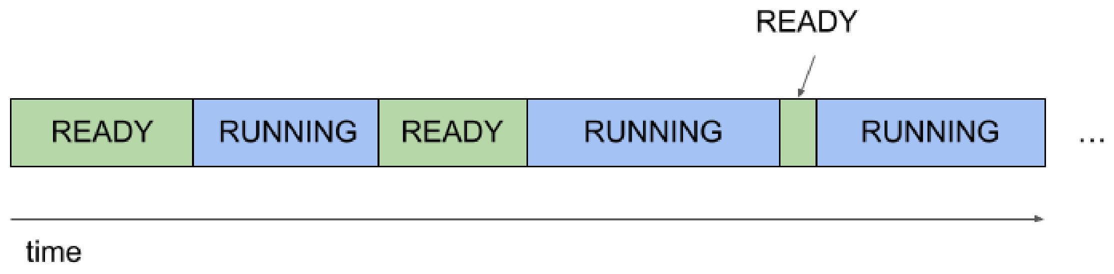
If the program you wrote in part (a) persisted for about 5 seconds of wall clock time using the pattern displayed above, what would a possible output of `time` be? (Hint: `time` also works for programs that are terminated using the `SIGINT` signal.)

[Solution.] The program would show about 5s of real time, and a very small amount of CPU time consumption since it was almost always blocked:

```
$ time timeout 5 ./sleeploop

real   0m5.002s
user   0m0.002s
sys    0m0.001s
```

2. Now consider the following timeline



(a) (3 pts) Write a program, in a programming language of your choice, that could have produced this timeline when executed on an OS.

[Solution.] Any program that doesn't engage in operations that could lead to blocking would do, the simplest is of course an infinite loop:

```
int
main()
{
    while (1);
}
```

(b) (2 pts) Based on this timeline, what can you infer about the momentary state of the machine on which this process ran?

[Solution.] Since the scheduler placed the process into the READY state, we can conclude that the total number of READY or RUNNING must have exceeded the number of available CPUs/cores on that machine, at least momentarily. (Assuming that a work conserving scheduler is used that never leaves CPUs intentionally idle.)

The ellipses indicate that the plot would continue in the patterns displayed.

1.3 Instruction Shuffle (5 pts)

As part of a new fuzzing system, Dr. Back wrote a Python script that takes the assembly code output by a compiler, identifies the regions of instructions that belong to one function and shuffles them randomly.

For instance, if the program is

```
#include <stdio.h>

int
main(int ac, char *av[])
{
    for (int i = 1; i < ac; i++)
        printf("%s%c", av[i], i == ac - 1 ? '\n' : ' ');
}
```

then the compiler output (without optimizations) would be

```
        .file        "echo.c"
        .text
        .section        .rodata
.LC0:
        .string        "%s%c"
        .text
        .globl        main
        .type        main, @function
main:
.LFB0:
        .cfi_startproc
        pushq        %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        subq        $32, %rsp
        movl        %edi, -20(%rbp)
        movq        %rsi, -32(%rbp)
        movl        $1, -4(%rbp)
        jmp        .L2
.L5:
        movl        -20(%rbp), %eax
        subl        $1, %eax
        cmpl        %eax, -4(%rbp)
        jne        .L3
        movl        $10, %ecx
        jmp        .L4
.L3:
        movl        $32, %ecx
.L4:
        movl        -4(%rbp), %eax
        cltq
        leaq        0(,%rax,8), %rdx
        movq        -32(%rbp), %rax
        addq        %rdx, %rax
        movq        (%rax), %rax
        movl        %ecx, %edx
        movq        %rax, %rsi
        movl        $.LC0, %edi
        movl        $0, %eax
        call        printf
        addl        $1, -4(%rbp)
.L2:
        movl        -4(%rbp), %eax
        cmpl        -20(%rbp), %eax
        jl         .L5
        movl        $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size        main, .-main
        .ident        "GCC: (GNU) 8.5.0 20210514 (Red Hat 8.5.0-10)"
        .section        .note.GNU-stack,"",@progbits
```

A random shuffle may be:

```
        .file        "echo.c"
        .text
        .section        .rodata
.LC0:
        .string        "%s%c"
        .text
        .globl        main
        .type        main, @function
main:
.LFB0:
        .cfi_startproc
.L4:
        pushq        %rbp
        .cfi_def_cfa_offset 16
        movl        -4(%rbp), %eax
        movl        $10, %ecx
        cmpl        -20(%rbp), %eax
        movl        -4(%rbp), %eax
        movl        $.LC0, %edi
        movq        (%rax), %rax
        cltq
        movl        %edi, -20(%rbp)
        subq        $32, %rsp
        .cfi_def_cfa 7, 8
        cmpl        %eax, -4(%rbp)
        addl        $1, -4(%rbp)
        movl        $0, %eax
        ret
        movq        -32(%rbp), %rax
        subl        $1, %eax
        addq        %rdx, %rax
        .cfi_def_cfa_register 6
        movq        %rsp, %rbp
        call        printf
        leave
        movl        -20(%rbp), %eax
.L3:
        .cfi_offset 6, -16
        movl        $0, %eax
        movq        %rsi, -32(%rbp)
.L2:
        leaq        0(,%rax,8), %rdx
        jmp        .L2
.L5:
        movq        %rax, %rsi
        jmp        .L4
        movl        $1, -4(%rbp)
        movl        %ecx, %edx
        jne        .L3
        jl        .L5
        movl        $32, %ecx
        .cfi_endproc
.LFE0:
        .size        main, .-main
        .ident        "GCC: (GNU) 8.5.0 20210514 (Red Hat 8.5.0-10)"
        .section        .note.GNU-stack,"",@progbits
```

The resulting shuffled .s files are then assembled, linked, and run.

1. (3 pts) Discuss 3 distinct ways in which these programs can fail, where fail is defined as not fulfilling the function of the original program. Briefly describe each way using the concrete terminology used by Unix users and programmers.

   [Solution.]

   - The program fails with an invalid memory access, signaled by `SIGSEGV` or `SIGBUS`
   - The program enters an infinite loop
   - The program exits without outputting anything

   These are in fact the most common. Theoretically possible are also other failures such as `SIGILL` (for instance, if a push/ret combination occurs).

2. (2 pts) What security risks, if any, does running this shuffled assembly code on our rlogin machines entail? Justify your answer.

   [Solution.] The echo program shown above will not entail significant security risk because any failure of the program will simply lead to the termination of the offending process, which is something that OS are designed to do. In the worst case (time out) the user would have to kill the process, but they can do without incurring significant risks.

2 Library OSes (8 pts)

A library operating system is an OS structure in which a significant fraction of functionality traditionally implemented in the OS kernel is instead implemented in a userspace library linked directly with the application.

In class, we have learned about how the system call interface is the mechanism in which processes communicate with the OS kernel. The OS kernel provides relatively high-level interfaces to hardware resources on the machine. For example, the OS kernel typically will provide access to the network via socket-related system calls and file descriptors, and it typically provides access to storage via filesystem-related system calls and file descriptors.

In a library OS, the functionality that was once implemented in the kernel as a system call may be implemented partially or entirely in userspace libraries. Network stacks (e.g., TCP/IP protocols) are implemented in userspace libraries; these libraries interact with network devices via low-level interfaces with the kernel, such as *raw sockets* or in some cases bypass the kernel entirely and communicate directly with the hardware. Filesystems are implemented in userspace libraries and interact with storage devices via low-level *block storage* interfaces with the kernel.

Figure 1 shows a diagram of traditional vs. libOS architectures.

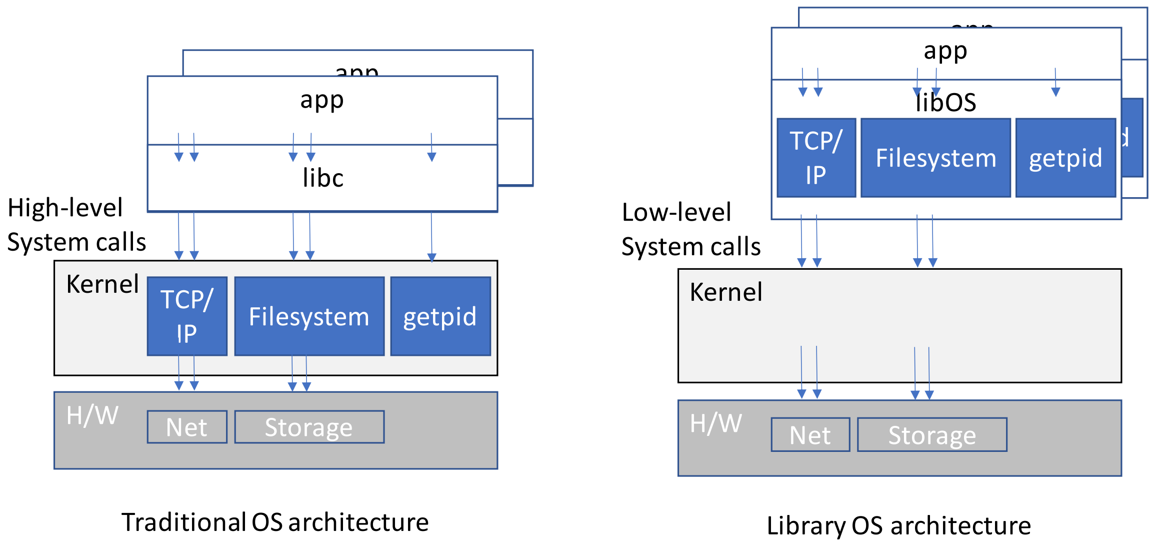


Figure 1: Traditional vs. libOS architectures

Your task is to consider the advantages and disadvantages of the library OS architecture from the perspective of the user and/or system designer. Consider the usual characteristics users and/or system designers aim to achieve (e.g., robustness, security, performance, implementation complexity, etc.). Then answer the following questions:

1. What is one potential advantage of adopting a library OS architecture?

   [Solution.]

- From the system designer's perspective; greater isolation due to a thinner, lower-level system call abstraction, similar to virtualization (see hotCloud)
- Related to the above, faults or vulnerabilities in the libOS "kernel" are contained to a single application/libOS rather than having the ability to take out the entire host
- The application can run specialized versions of kernel services, tailored to the application's specific needs, which can be high performance or memory safe. (see MirageOS)
- Some system calls can be faster due to not needing to context switch to kernel mode. For example, a filesystem on top of a ramdisk (like tmpfs) can be implemented entirely in userspace. Similarly the library OS is well suited to kernel bypass devices (see Arrakis) for performance.
- Possible to use in new hardware environments with limited OS support in the hardware (see Haven)

2. What is one potential disadvantage of adopting a library OS architecture?

   [Solution.]

   - Multiprocessing, while not impossible (see Graphene) is unnatural, complex and inefficient. Think of what fork or pipes would look like in this architecture.
   - There is no protection between OS services in the library OS and the application; an errant memory write could easily modify/corrupt a kernel data structure.
   - Existing kernel implementations of kernel services are high performance and library OS implementations can struggle to match their performance. (see StackMap)
   - No familiar OS-level debugging tools (e.g., ps, top, strace, netstat, etc.) (see Cantrill)

3 Unix Processes and IPC (34 pts)

3.1 Zombies Oh My (12 pts)

Consider a program `zombie.c` which is compiled to a binary called `zombie` using the command

```
gcc -o zombie zombie.c
```

When a user runs this program, the following session results:

```
[gback@holly zombietree]$ gcc zombie.c -o zombie
[gback@holly zombietree]$ ./zombie &
[1] 4074762
[gback@holly zombietree]$ ps f
    PID TTY      STAT   TIME COMMAND
3988368 pts/17   Ss     0:00 -bash
4074762 pts/17   S      0:00  \_ ./zombie
4074763 pts/17   S      0:00  |   \_ ./zombie
4074764 pts/17   Z      0:00  |       \_ [zombie] <defunct>
4074765 pts/17   Z      0:00  |       \_ [zombie] <defunct>
4074766 pts/17   Z      0:00  |       \_ [zombie] <defunct>
4074769 pts/17   R+     0:00  \_ ps f
```

1. (2 pts) When the user types

   ```
   [gback@holly zombietree]$ kill -9 4074764 4074765 4074766
   [gback@holly zombietree]$ ps f
   ```

   next, what would the output be?

   [Solution.] The output would be exactly the same as above. Attempts to kill zombie processes are futile because these processes have already terminated.

2. (2 pts) Next, when the user types

   ```
   [gback@holly zombietree]$ kill -9 4074763
   ```

   and hits enter, they see

   ```
   [gback@holly zombietree]$
   [1]+  Done                    ./zombie
   ```

   If the user now typed

   ```
   [gback@holly zombietree]$ ps f
   ```

   how many processes related to the zombie program would `ps` report?

   [Solution.] Zero processes. The unconditional kill will terminate the zombie's parent process, in the course allowing the zombies to be reaped - the OS will keep a zombie process around only as long as its parent is alive. Once the parent is gone, there is no wait who could possibly wait for the zombie process, thus there's no need to keep it around longer.

3. (8 pts) Reconstruct `zombie.c` so that it behaves in exactly the way shown above, including matching the same Linux process states that are shown in the ps output.

   [Solution.] Since killing process 4074763 caused process 4074762 to exit, we conclude that process 4074762 was waiting for 4074763. A sleep puts process 4074763 in the `S` state. The reconstruction therefore is given by:

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

int
main()
{
    if (fork() == 0) {
        for (int i = 0; i < 3; i++) {
            if (fork() == 0)
                exit(0);    // zombie
        }
        sleep(1000);
    } else {
        wait(NULL);
    }
}
```

3.2 Unix Signals (10 pts)

The following program, `sendsig`, generates signals in random order to a pid specified as its command line argument:

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

int main(int argc, char **argv) {
    pid_t pid = atoi(argv[1]);

    printf("sending signals to %d\n", pid);

    if (random() % 2 == 0) {
        kill(pid, SIGUSR1);
        kill(pid, SIGUSR2);
    } else {
        kill(pid, SIGUSR2);
        kill(pid, SIGUSR1);
    }

    sleep(5);

    kill(pid, SIGKILL);
}
```

As you know, a program can catch signals, but sometimes it is useful for the program to ensure that signals are handled in a specific order. Remember, signals are delivered asynchronously and ordering is not guaranteed by the system! In this question, your task is to write a program, `recvsig` that catches the signals sent by `sendsig` *in a given order* and prints messages to standard out upon receipt of signals (e.g., in signal handlers).

For example, if you run `recvsig` in one terminal, then `recvsig` will output its own pid and a message indicating that it is waiting for signals.

```
$ ./recvsig
976488 waiting for signals...
```

Then, running `sendsig` on a second terminal,

```
$ ./sendsig 976488
```

will result in `sendsig` sending signals (in a random order) to the specified pid, in this case `recvsig`. Your program, `recvsig` should then output, the following messages in the following order, regardless of the order `sendsig` generated them:

```
got SIGUSR1
got SIGUSR2
```

The program should then continue to wait for at least 5 seconds until the final signal from `sendsig` arrives, causing the bash shell to output the following message:

```
Killed
```

To be clear, a full run of your program, `recvsig`, from a bash shell, where 976488 is just an example pid (yours will differ), should look like this after you run `sendsig` in another bash shell:

```
$ ./recvsig
976488 waiting for signals...
got SIGUSR1
got SIGUSR2
Killed
```

Your program may not print messages indicating it received a signal unless the signal was actually received. Your program must output messages in the specified order, regardless of the order the signals arrive in. Your program must not exit before the final SIGKILL arrives from `sendsig`. You may find the signal support functions from p1 (e.g., `signal_support.[ch]`) helpful.

[Solution]

Even though we know the signals will arrive asynchronously in a random order, we can manage the order that the application receives them through use of signal blocking/unblocking, similar to how we blocked/unblocked SIGCHLD in project 1. Specifically, if `recvsig` begins with both SIGUSR1 and SIGUSR2 temporarily blocked, we know the kernel will not deliver those signals, but queue them for later. If `recvsig` unblocks only SIGUSR1, SIGUSR2 will be queued until it is unblocked, even if it arrived before SIGUSR1. In the provided solution, we unblock SIGUSR2 in the signal handler for SIGUSR1. Then the queued signal for SIGUSR2 arrives and usr2hdlr is called.

Our program must wait around long enough to receive the final signal (SIGKILL). There are a number of ways to do so, including `sigwait`, but one thing to be aware of is that certain system calls like `sleep` will return when *any* signal is delivered. Hence, if you thought about sleeping for a long time to wait for the SIGKILL, you must do so in a loop or else the program will exit prematurely.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <assert.h>
#include <unistd.h>
#include "signal_support.h"


void usr1hdlr(int sig, siginfo_t *info, void *a) {
    assert(sig == SIGUSR1);
    printf("got SIGUSR1\n");
    signal_unblock(SIGUSR2);
}

void usr2hdlr(int sig, siginfo_t *info, void *a) {
    assert(sig == SIGUSR2);
    printf("got SIGUSR2\n");
}


int main() {
    signal_block(SIGUSR1);
    signal_block(SIGUSR2);

    signal_set_handler(SIGUSR1, usr1hdlr);
    signal_set_handler(SIGUSR2, usr2hdlr);

    printf("%d waiting for signals...\n", getpid());
    signal_unblock(SIGUSR1);

    for(;;)
        sleep(10000);
}
```

3.3 Mystery Tool (12 pts)

Consider the following excerpts of system call traces which were obtained by running a mystery tool you are asked to reverse engineering in this question.

When started with `./mysterytool date`, the following 2 system call traces resulted[1]

```
execve("./mysterytool", ["./mysterytool", "date"], 0x7ffd558eb730 /* 43 vars */) = 0
...
pipe2([3, 4], O_CLOEXEC)                = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
                   child_tidptr=0x7efc675358d0) = 3150789
close(4)                                = 0
...
read(3, "Mon Feb 28 09:29:14 EST 2022\n", 4096) = 29
...
write(1, "mon feb 28 09:29:14 est 2022\n", 29) = 29
read(3, "", 4096)                       = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

(lines broken for readability).

Process 3150789 made the following system calls:

```
dup2(4, 1)                              = 1
execve("/bin/sh", ["sh", "-c", "date"], 0x7ffe1f41fc90 /* 43 vars */) = 0
...
execve("/usr/bin/date", ["date"], 0x55f444fb8430 /* 43 vars */) = 0
...
write(1, "Mon Feb 28 09:29:14 EST 2022\n", 29) = 29
...
exit_group(0)                           = ?
+++ exited with 0 +++
```

However, when started with
`./mysterytool "cat /web/courses/cs3214/spring2022/exams/test1/testfile"`
the following system call trace resulted:

```
execve("./mysterytool", ["./mysterytool",
                         "cat /web/courses/cs3214/spring2022/exams/test1/testfile"],
                         0x7ffeff798bc0 /* 43 vars */) = 0
...
pipe2([3, 4], O_CLOEXEC)                = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
                child_tidptr=0x7f9c9962e8d0) = 3160584
close(4)                                = 0
```

[1]Lines in the output were broken for readability, and non-essential lines were marked with an ellipsis. Certain numbers (such as 43 vars in execve) are specific to the user environment at the time the experiment was performed on rlogin.

```
...
read(3, "A short message with UPPERCASE characters.\n", 4096) = 43
...
write(1, "a short message with uppercase characters.\n", 43) = 43
read(3, "", 4096)                         = 0
exit_group(0)                             = ?
+++ exited with 0 +++
```

Process 3160584 made the following system calls:

```
dup2(4, 1)                                = 1
execve("/bin/sh", ["sh", "-c", "cat /web/courses/cs3214/spring2022/exams/test1/testfile"],
   0x7ffd3c1190f0 /* 43 vars */) = 0
...
execve("/usr/bin/cat", ["cat", "/web/courses/cs3214/spring2022/exams/test1/testfile"],
     0x557068117450 /* 43 vars */) = 0
...
read(3, "A short message with UPPERCASE characters.\n", 1048576) = 43
write(1, "A short message with UPPERCASE characters.\n", 43) = 43
read(3, "", 1048576)                      = 0
...
exit_group(0)                             = ?
+++ exited with 0 +++
```

Note that in both cases, any uppercase characters in the standard output of `date` and `cat` were replaced with lowercase characters.

Reverse-engineer this program in C and provide a brief description as to how it works. Include the description as comments inside the source code. You may use suitable functions that are part of the C stdio and/or POSIX standard library, or you may use low-level I/O functions that make system calls directly.

Notes:

- Note that in the second invocation of the tool, `"cat /web/courses/...."` forms a single argument, and that `/bin/sh -c` expects a single argument, which is parsed as per the usual conventions by the shell that is invoked.
- Also, the two invocations are only examples - your program needs to work for any valid shell command provided as an argument.
- Error checking is not required for the purposes of this problem.
- The use of any form of the `wait()` system call is optional as well.

[Solution.] The strace output shows that the program creates a pipe, then forks. The child process redirects this pipe via dup2 to appear at the standard output file descriptor (1), then executes a shell with the `-c` flag, thus executing the given command. The parent process reads from the pipe until EOF is reached and output a lowercase version of the data read to its standard output.

This is what the `popen(3)` function does, so a short reconstruction is:

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

int
main(int ac, char *av[])
{
    FILE * in = popen(av[1], "r");
    int c;
    while ((c = fgetc(in)) != EOF)
        fputc(tolower(c), stdout);
}
```

If you're unaware of popen, direct use of system calls (like in ex1) will work, too:

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <ctype.h>

int
main(int ac, char *av[])
{
    int fd[2];
    pipe2(fd, O_CLOEXEC);
    if (fork() == 0) {
        dup2(fd[1], STDOUT_FILENO);
        execl("/bin/sh", "/bin/sh", "-c", av[1], NULL);
    } else {
        close(fd[1]);
        char buf[4096];
        size_t bread;
        while ((bread = read(fd[0], buf, sizeof buf)) > 0) {
            for (int i = 0; i < bread; i++)
                buf[i] = tolower(buf[i]);
            write(STDOUT_FILENO, buf, bread);
        }
    }
}
```

4 Development and Linking (16 pts)

4.1 Undefined Behavior (6 pts)

Understanding the concept and manifestations of undefined behavior (UB) is a crucial skill for any programmer in languages that allow such behavior, such as C, C++, or (unsafe) Rust. To that end you extensively used tools such as the undefined behavior sanitizer (UBSAN) or valgrind in project 1.

In this problem, you are given the output of these tools on three programs and you are asked to reconstruct the programs that produced this output. The programs shall be called `ub1.c`, `ub2.c`, and `ub3.c` and they are compiled as follows:

```
gcc -o ub1 -g -fsanitize=undefined ub1.c
gcc -o ub2 -g ub2.c
gcc -o ub3 -g ub3.c
```

1. (2 pts) ub1 produces the following output when run as `./ub1`

```
ub1.c:9:13: runtime error: index 40 out of bounds for type 'int [40]'
ub1.c:9:13: runtime error: load of address 0x0000006011a0 with insufficient space for an object of type 'int'
0x0000006011a0: note: pointer points here
 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00
              ^
```

[Solution.]

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

static int b[40];

int
main()
{
    return b[40];
}
```

2. (2 pts) ub2 produces the following output when run with `valgrind ./ub2`

```
==4034483== Memcheck, a memory error detector
==4034483== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4034483== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4034483== Command: ./ub2
==4034483==
==4034483== Invalid write of size 1
==4034483==    at 0x4005B4: main (ub2.c:8)
==4034483==  Address 0x520c044 is 0 bytes after a block of size 4 alloc'd
==4034483==    at 0x4C37135: malloc (vg_replace_malloc.c:381)
==4034483==    by 0x4005A7: main (ub2.c:7)
==4034483==
(rest is elided)
```

[Solution.]

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

int
main()
{
    char *p = malloc(4);
    p[4] = 0;
}
```

3. (2 pts) ub3 produces the following output when run with `valgrind ./ub3`

```
==4034493== Memcheck, a memory error detector
==4034493== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4034493== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4034493== Command: ./ub3
==4034493==
==4034493== Syscall param exit_group(status) contains uninitialised byte(s)
==4034493==    at 0x4F418F6: _Exit (in /usr/lib64/libc-2.28.so)
==4034493==    by 0x4E982E9: __run_exit_handlers (in /usr/lib64/libc-2.28.so)
==4034493==    by 0x4E9831F: exit (in /usr/lib64/libc-2.28.so)
==4034493==    by 0x400597: main (ub3.c:8)
==4034493==
(rest is elided)
```

[Solution.]

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

int
main()
{
    int x;
    return x;
}
```

4.2 A Linker Puzzle (10 pts)

Four files are involved in a build process: `module1.c`, `module2.c`, `module3.c`, and `headerfile.h`. The following is known:

- When compiled with

  ```
  gcc -c -Wall -Wmissing-prototypes module[123].c
  ```

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

- All three of `module1.c`, `module2.c`, and `module3.c` include `headerfile.h`.
- The following symbol tables are reported by the `nm` command

  ```
  nm module1.o

  0000000000000000 T main
  0000000000000000 B module1_flag
                   U module2_fun

  nm module2.o

  0000000000000000 t helper
  000000000000000b T module2_fun
                   U module3_fun
  0000000000000008 r msg

  nm module3.o

  0000000000000000 t helper
                   U module1_flag
  0000000000000016 T module3_fun
                   U printf
  ```

- When linked with

  ```
  gcc -Wall -Wmissing-prototypes module[123].c -o main
  ```

  we see no errors.
- When we extract the symbol table from the executable `main` we see, among others, the following symbols:

  ```
  00000000004005a6 t helper
  00000000004005ca t helper
  0000000000400596 T main
  0000000000601028 B module1_flag
  00000000004005b1 T module2_fun
  00000000004005e0 T module3_fun
  00000000004006b0 r msg
                   U printf@@GLIBC_2.2.5
  ```

- Best practices related to scoping and linking, as discussed in lecture, were followed in this mini project.

Reconstruct `module1.c`, `module2.c`, `module3.c`, and `headerfile.h`!

If there are multiple reconstructions that meet the conditions described, any of them will be accepted. The addresses of the symbols do not need to match the ones shown, but the type and scope must. All facts that are listed in the problem must hold true for your reconstruction, including the observations about compilation.

[Solution.] A possible reconstruction is given below:

headerfile.h

```
void module2_fun(void);
void module3_fun(void);
extern int module1_flag;
```

module1.c

```
#include "headerfile.h"

int module1_flag = 0;

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

module2.c

```
#include "headerfile.h"

static void helper(const char *msg) {
}

static char * const msg = "message";

void
module2_fun()
{
    helper(msg);
    module3_fun();
}
```

module3.c

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

static void helper() {
    module1_flag++;
}

void
module3_fun()
{
    helper();
    printf("module3_fun %d\n", module1_flag);
}
```

Note that `helper` cannot be in `headerfile.h` since it is not appearing in the symbol table of module1. To prevent the compiler from replacing `printf` with `puts` it should be invoked with a format string containing format modifiers.

5 Submission Requirements

Submit a tar file that contains the following files:

- A file `osfacts.txt` with your answers to question 1.1.
- A file with basename `blockedrunning` and a suitable suffix in a language of your choice for question 1.2.1.a).
- A file with basename `readyrunning` and a suitable suffix in a language of your choice for question 1.2.2.a).
- A file `zombie.c` with your answer to Question 3.1 part 3.
- A file `recvsig.c` with your answer to Question 3.2. (If your file relies on `signal_support.c` from p1, you do not need to include this file.)
- A file `mysterytool.c` with your solution to question 3.3.
- Files `ub1.c`, `ub2.c`, and `ub3.c` with the answers to Question 4.1.
- `module1.c`, `module2.c`, `module3.c`, and `headerfile.h` for question 4.2. These source files should meet all requirements stated in the problem.
- A UTF-8 encoded text file `answers.txt` with your answers for all remaining questions. Good answers are precise, brief, and cite relevant material as necessary.