`java.io.InputStream` class returns –1 when the client has closed its end of the socket connection.

3.26 Design a program using ordinary pipes in which one process sends a string message to a second process, and the second process reverses the case of each character in the message and sends it back to the first process. For example, if the first process sends the message `Hi There`, the second process will return `hI tHERE`. This will require using two pipes, one for sending the original message from the first to the second process and the other for sending the modified message from the second to the first process. You can write this program using either UNIX or Windows pipes.

3.27 Design a file-copying program named `filecopy.c` using ordinary pipes. This program will be passed two parameters: the name of the file to be copied and the name of the destination file. The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe. The child process will read this file from the pipe and write it to the destination file. For example, if we invoke the program as follows:

```
./filecopy input.txt copy.txt
```

the file `input.txt` will be written to the pipe. The child process will read the contents of this file and write it to the destination file `copy.txt`. You may write this program using either UNIX or Windows pipes.

Programming Projects

Project 1—UNIX Shell

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. Completing this project will involve using the UNIX `fork()`, `exec()`, `wait()`, `dup2()`, and `pipe()` system calls and can be completed on any Linux, UNIX, or macOS system.

I. Overview

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command: `cat prog.c`. (This command displays the file `prog.c` on the terminal using the UNIX `cat` command.)

```
osh>cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, `cat prog.c`) and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 3.9. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (`&`) at the end of the command. Thus, if we rewrite the above command as

```
osh>cat prog.c &
```

the parent and child processes will run concurrently.

The separate child process is created using the `fork()` system call, and the user's command is executed using one of the system calls in the `exec()` family (as described in Section 3.3.1).

A C program that provides the general operations of a command-line shell is supplied in Figure 3.36. The `main()` function presents the prompt `osh->` and outlines the steps to be taken after input from the user has been read. The `main()` function continually loops as long as `should_run` equals 1; when the user enters `exit` at the prompt, your program will set `should_run` to 0 and terminate.

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

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
char *args[MAX_LINE/2 + 1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */

   while (should_run) {
      printf("osh>");
      fflush(stdout);

      /**
       * After reading user input, the steps are:
       * (1) fork a child process using fork()
       * (2) the child process will invoke execvp()
       * (3) parent will invoke wait() unless command included &
       */
   }

    return 0;
}
```

Figure 3.36 Outline of simple shell.

This project is organized into several parts:

1. Creating the child process and executing the command in the child
2. Providing a history feature
3. Adding support of input and output redirection
4. Allowing the parent and child processes to communicate via a pipe

II. Executing Command in a Child Process

The first task is to modify the `main()` function in Figure 3.36 so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (`args` in Figure 3.36). For example, if the user enters the command `ps -ael` at the `osh>` prompt, the values stored in the `args` array are:

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

This `args` array will be passed to the `execvp()` function, which has the following prototype:

```
execvp(char *command, char *params[])
```

Here, `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as `execvp(args[0], args)`. Be sure to check whether the user included & to determine whether or not the parent process is to wait for the child to exit.

III. Creating a History Feature

The next task is to modify the shell interface program so that it provides a ***history*** feature to allow a user to execute the most recent command by entering `!!`. For example, if a user enters the command `ls -l`, she can then execute that command again by entering `!!` at the prompt. Any command executed in this fashion should be echoed on the user's screen, and the command should also be placed in the history buffer as the next command.

Your program should also manage basic error handling. If there is no recent command in the history, entering `!!` should result in a message "`No commands in history.`"

IV. Redirecting Input and Output

Your shell should then be modified to support the '`>`' and '`<`' redirection

operators, where '>' redirects the output of a command to a file and '<' redirects the input to a command from a file. For example, if a user enters

```
osh>ls > out.txt
```

the output from the `ls` command will be redirected to the file `out.txt`. Similarly, input can be redirected as well. For example, if the user enters

```
osh>sort < in.txt
```

the file `in.txt` will serve as input to the `sort` command.

Managing the redirection of both input and output will involve using the `dup2()` function, which duplicates an existing file descriptor to another file descriptor. For example, if `fd` is a file descriptor to the file `out.txt`, the call

```
dup2(fd, STDOUT_FILENO);
```

duplicates `fd` to standard output (the terminal). This means that any writes to standard output will in fact be sent to the `out.txt` file.

You can assume that commands will contain either one input or one output redirection and will not contain both. In other words, you do not have to be concerned with command sequences such as `sort < in.txt > out.txt`.

V. Communication via a Pipe

The final modification to your shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

```
osh>ls -l | less
```

has the output of the command `ls -l` serve as the input to the `less` command. Both the `ls` and `less` commands will run as separate processes and will communicate using the UNIX `pipe()` function described in Section 3.7.4. Perhaps the easiest way to create these separate processes is to have the parent process create the child process (which will execute `ls -l`). This child will also create another child process (which will execute `less`) and will establish a pipe between itself and the child process it creates. Implementing pipe functionality will also require using the `dup2()` function as described in the previous section. Finally, although several commands can be chained together using multiple pipes, you can assume that commands will contain only one pipe character and will not be combined with any redirection operators.

Project 2 — Linux Kernel Module for Task Information

In this project, you will write a Linux kernel module that uses the `/proc` file system for displaying a task's information based on its process identifier value `pid`. Before beginning this project, be sure you have completed the Linux kernel module programming project in Chapter 2, which involves creating an entry in the `/proc` file system. This project will involve writing a process identifier to

the file /proc/pid. Once a pid has been written to the /proc file, subsequent reads from /proc/pid will report (1) the command the task is running, (2) the value of the task's pid, and (3) the current state of the task. An example of how your kernel module will be accessed once loaded into the system is as follows:

```
echo "1395" > /proc/pid
cat /proc/pid
command = [bash] pid = [1395] state = [1]
```

The echo command writes the characters "1395" to the /proc/pid file. Your kernel module will read this value and store its integer equivalent as it represents a process identifier. The cat command reads from /proc/pid, where your kernel module will retrieve the three fields from the task_struct associated with the task whose pid value is 1395.

```
ssize_t proc_write(struct file *file, char __user *usr_buf,
   size_t count, loff_t *pos)
{
   int rv = 0;
   char *k_mem;

   /* allocate kernel memory */
   k_mem = kmalloc(count, GFP_KERNEL);

   /* copies user space usr_buf to kernel memory */
   copy_from_user(k_mem, usr_buf, count);

   printk(KERN_INFO "%s\n", k_mem);

   /* return kernel memory */
   kfree(k_mem);

   return count;
}
```

Figure 3.37 The proc_write() function.

I. Writing to the /proc File System

In the kernel module project in Chapter 2, you learned how to read from the /proc file system. We now cover how to write to /proc. Setting the field .write in struct file_operations to

```
.write = proc_write
```

causes the proc_write() function of Figure 3.37 to be called when a write operation is made to /proc/pid

The kmalloc() function is the kernel equivalent of the user-level malloc() function for allocating memory, except that kernel memory is being allocated. The GFP_KERNEL flag indicates routine kernel memory allocation. The copy_from_user() function copies the contents of usr_buf (which contains what has been written to /proc/pid) to the recently allocated kernel memory. Your kernel module will have to obtain the integer equivalent of this value using the kernel function kstrtol(), which has the signature

```
int kstrtol(const char *str, unsigned int base, long *res)
```

This stores the character equivalent of str, which is expressed as a base into res.

Finally, note that we return memory that was previously allocated with kmalloc() back to the kernel with the call to kfree(). Careful memory management—which includes releasing memory to prevent ***memory leaks***—is crucial when developing kernel-level code.

II. Reading from the /proc File System

Once the process identifier has been stored, any reads from /proc/pid will return the name of the command, its process identifier, and its state. As illustrated in Section 3.1, the PCB in Linux is represented by the structure task_struct, which is found in the <linux/sched.h> include file. Given a process identifier, the function pid_task() returns the associated task_struct. The signature of this function appears as follows:

```
struct task_struct pid_task(struct pid *pid,
   enum pid_type type)
```

The kernel function find_vpid(int pid) can be used to obtain the struct pid, and PIDTYPE_PID can be used as the pid_type.

For a valid pid in the system, pid_task will return its task_struct. You can then display the values of the command, pid, and state. (You will probably have to read through the task_struct structure in <linux/sched.h> to obtain the names of these fields.)

If pid_task() is not passed a valid pid, it returns NULL. Be sure to perform appropriate error checking to check for this condition. If this situation occurs, the kernel module function associated with reading from /proc/pid should return 0.

In the source code download, we give the C program pid.c, which provides some of the basic building blocks for beginning this project.

Project 3—Linux Kernel Module for Listing Tasks

In this project, you will write a kernel module that lists all current tasks in a Linux system. You will iterate through the tasks both linearly and depth first.

Part I—Iterating over Tasks Linearly

In the Linux kernel, the for_each_process() macro easily allows iteration over all current tasks in the system:

```
#include <linux/sched.h>

struct task_struct *task;

for_each_process(task) {
   /* on each iteration task points to the next task */
}
```

The various fields in `task_struct` can then be displayed as the program loops through the `for_each_process()` macro.

Assignment

Design a kernel module that iterates through all tasks in the system using the `for_each_process()` macro. In particular, output the task command, state, and process id of each task. (You will probably have to read through the `task_struct` structure in `<linux/sched.h>` to obtain the names of these fields.) Write this code in the module entry point so that its contents will appear in the kernel log buffer, which can be viewed using the `dmesg` command. To verify that your code is working correctly, compare the contents of the kernel log buffer with the output of the following command, which lists all tasks in the system:

```
ps -el
```

The two values should be very similar. Because tasks are dynamic, however, it is possible that a few tasks may appear in one listing but not the other.

Part II—Iterating over Tasks with a Depth-First Search Tree

The second portion of this project involves iterating over all tasks in the system using a depth-first search (DFS) tree. (As an example: the DFS iteration of the processes in Figure 3.7 is 1, 8415, 8416, 9298, 9204, 2808, 3028, 3610, 4005.)

Linux maintains its process tree as a series of lists. Examining the `task_struct` in `<linux/sched.h>`, we see two `struct list_head` objects:

```
children
```

and

```
sibling
```

These objects are pointers to a list of the task's children, as well as its siblings. Linux also maintains a reference to the initial task in the system — `init_task` — which is of type `task_struct`. Using this information as well as macro operations on lists, we can iterate over the children of `init_task` as follows:

```
struct task_struct *task;
struct list_head *list;
```

```
list_for_each(list, &init_task->children) {
   task = list_entry(list, struct task_struct, sibling);
   /* task points to the next child in the list */
}
```

The `list_for_each()` macro is passed two parameters, both of type `struct list_head`:

- A pointer to the head of the list to be traversed
- A pointer to the head node of the list to be traversed

At each iteration of `list_for_each()`, the first parameter is set to the `list` structure of the next child. We then use this value to obtain each structure in the list using the `list_entry()` macro.

Assignment

Beginning from `init_task` task, design a kernel module that iterates over all tasks in the system using a DFS tree. Just as in the first part of this project, output the name, state, and pid of each task. Perform this iteration in the kernel entry module so that its output appears in the kernel log buffer.

If you output all tasks in the system, you may see many more tasks than appear with the `ps -ael` command. This is because some threads appear as children but do not show up as ordinary processes. Therefore, to check the output of the DFS tree, use the command

```
ps -eLf
```

This command lists all tasks—including threads—in the system. To verify that you have indeed performed an appropriate DFS iteration, you will have to examine the relationships among the various tasks output by the `ps` command.

Project 4—Kernel Data Structures

In Section 1.9, we covered various data structures that are common in operating systems. The Linux kernel provides several of these structures. Here, we explore using the circular, doubly linked list that is available to kernel developers. Much of what we discuss is available in the Linux source code—in this instance, the include file `<linux/list.h>`—and we recommend that you examine this file as you proceed through the following steps.

Initially, you must define a `struct` containing the elements that are to be inserted in the linked list. The following C `struct` defines a color as a mixture of red, blue, and green:

```
struct color {
   int red;
   int blue;
   int green;
```

```
    struct list_head list;
};
```

Notice the member `struct list_head list`. The `list_head` structure is defined in the include file `<linux/types.h>`, and its intention is to embed the linked list within the nodes that comprise the list. This `list_head` structure is quite simple—it merely holds two members, `next` and `prev`, that point to the next and previous entries in the list. By embedding the linked list within the structure, Linux makes it possible to manage the data structure with a series of *macro* functions.

I. Inserting Elements into the Linked List

We can declare a `list_head` object, which we use as a reference to the head of the list by using the `LIST_HEAD()` macro:

```
static LIST_HEAD(color_list);
```

This macro defines and initializes the variable `color_list`, which is of type `struct list_head`.

We create and initialize instances of `struct color` as follows:

```
struct color *violet;

violet = kmalloc(sizeof(*violet), GFP_KERNEL);
violet->red = 138;
violet->blue = 43;
violet->green = 226;
INIT_LIST_HEAD(&violet->list);
```

The `kmalloc()` function is the kernel equivalent of the user-level `malloc()` function for allocating memory, except that kernel memory is being allocated. The `GFP_KERNEL` flag indicates routine kernel memory allocation. The macro `INIT_LIST_HEAD()` initializes the `list` member in `struct color`. We can then add this instance to the end of the linked list using the `list_add_tail()` macro:

```
list_add_tail(&violet->list, &color_list);
```

II. Traversing the Linked List

Traversing the list involves using the `list_for_each_entry()` macro, which accepts three parameters:

- A pointer to the structure being iterated over
- A pointer to the head of the list being iterated over
- The name of the variable containing the `list_head` structure

The following code illustrates this macro:

```
struct color *ptr;

list_for_each_entry(ptr, &color_list, list) {
   /* on each iteration ptr points */
   /* to the next struct color */
}
```

III. Removing Elements from the Linked List

Removing elements from the list involves using the `list_del()` macro, which is passed a pointer to `struct list_head`:

```
list_del(struct list_head *element);
```

This removes `element` from the list while maintaining the structure of the remainder of the list.

Perhaps the simplest approach for removing all elements from a linked list is to remove each element as you traverse the list. The macro `list_for_each_entry_safe()` behaves much like `list_for_each_entry()` except that it is passed an additional argument that maintains the value of the `next` pointer of the item being deleted. (This is necessary for preserving the structure of the list.) The following code example illustrates this macro:

```
struct color *ptr, *next;

list_for_each_entry_safe(ptr,next,&color_list,list) {
   /* on each iteration ptr points */
   /* to the next struct color */
   list_del(&ptr->list);
   kfree(ptr);
}
```

Notice that after deleting each element, we return memory that was previously allocated with `kmalloc()` back to the kernel with the call to `kfree()`.

Part I—Assignment

In the module entry point, create a linked list containing four `struct color` elements. Traverse the linked list and output its contents to the kernel log buffer. Invoke the `dmesg` command to ensure that the list is properly constructed once the kernel module has been loaded.

In the module exit point, delete the elements from the linked list and return the free memory back to the kernel. Again, invoke the `dmesg` command to check that the list has been removed once the kernel module has been unloaded.

Part II—Parameter Passing

This portion of the project will involve passing a parameter to a kernel module. The module will use this parameter as an initial value and generate the Collatz sequence as described in Exercise 3.21.

Passing a Parameter to a Kernel Module

Parameters may be passed to kernel modules when they are loaded. For example, if the name of the kernel module is `collatz`, we can pass the initial value of 15 to the kernel parameter `start` as follows:

```
sudo insmod collatz.ko start=15
```

Within the kernel module, we declare `start` as a parameter using the following code:

```
#include<linux/moduleparam.h>

static int start = 25;

module_param(start, int, 0);
```

The `module_param()` macro is used to establish variables as parameters to kernel modules. `module_param()` is provided three arguments: (1) the name of the parameter, (2) its type, and (3) file permissions. Since we are not using a file system for accessing the parameter, we are not concerned with permissions and use a default value of 0. Note that the name of the parameter used with the `insmod` command must match the name of the associated kernel parameter. Finally, if we do not provide a value to the module parameter during loading with `insmod`, the default value (which in this case is 25) is used.

Part II—Assignment

Design a kernel module named `collatz` that is passed an initial value as a module parameter. Your module will then generate and store the sequence in a kernel linked list when the module is loaded. Once the sequence has been stored, your module will traverse the list and output its contents to the kernel log buffer. Use the `dmesg` command to ensure that the sequence is properly generated once the module has been loaded.

In the module exit point, delete the contents of the list and return the free memory back to the kernel. Again, use `dmesg` to check that the list has been removed once the kernel module has been unloaded.