~~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:~~