Programming Projects on Design of Operating System

A. The Collatz conjecture concerns what happens when we take any positive integer *n* and apply the following algorithm:

$$n = \begin{cases} n/2, & \text{if n is even.} \\ \\ 3 \times n + 1, & \text{if n is odd.} \end{cases}$$

The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1. For example, if $n = 35$, the sequence is

35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Write a C program using the fork () system call that generates this sequence in the child process. The starting number will be provided from the command line. For example, if 8 is passed as a parameter on the command line, the child process will output 8, 4, 2, 1. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the wait() call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a positive integer is passed on the command line.

B. Write a program that creates three child processes and three pipes to communicate with each process. Each child process reads from different serial line (keyboard) and sends the characters read back to the parent process through a pipe. The parent process outputs all characters received on the console. A child terminates when two newline characters are received consecutively. The parent terminates after all three children have terminated. (hint: send-pipe and receive-pipe primitives can be used)

C. Multithreaded Sorting Application: Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term ***sorting threads***) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread—a ***merging thread*** — which merges the two sublists into a single sorted list.

Because global data are shared across all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured as in Figure 1.

This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to

begin sorting. The parent thread will output the sorted array once all sorting threads have exited.

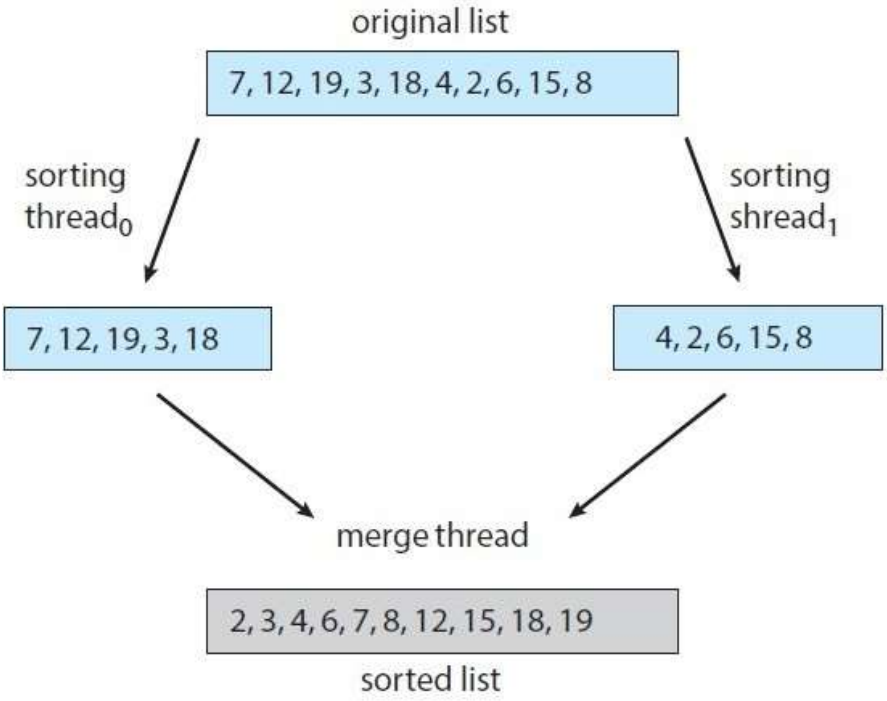

Figure 1: Multithreaded sorting

D. The reader writer problem:
A number of readers may simultaneously be reading from a file. Only one writer at a time may write to file, and no reader can be reading while a writer is writing.
Using semaphores, write program with solution to the reader writers problem that gives priority to writers, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

E. Modify the socket-based date server (Figure 3.27) in Chapter 3 so that the server services each client request in a separate thread.

F. The Producer – Consumer Problem

In Section 7.1.1, we presented a semaphore-based solution to the producer– consumer problem using a bounded buffer. In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in book Figures 5.9 and 5.10. The solution presented in Section 7.1.1 uses three semaphores: empty and full, which count the number of empty and full slots in the buffer, and mutex, which is a binary (or mutualexclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, you will use standard counting semaphores for empty and full and a mutex lock, rather than a binary semaphore, to represent mutex. The producer and consumer—running as separate threads —will move items to and from a buffer that is synchronized with the empty, full, and mutex structures. You can solve this problem using either Pthreads or Java API.

The Buffer

Internally, the buffer will consist of a fixed-size array of type buffer item (which will be defined using a typedef). The array of buffer item objects will be manipulated as a circular queue. The definition of buffer item, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef int buffer item;
#define BUFFER SIZE 5
```

The buffer will be manipulated with two functions, insert item() and remove item(), which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in Figure 2. The insert item() and remove item() functions will synchronize the producer and consumer using the algorithms outlined in Figure 7.1 and Figure 7.2. The buffer will also require an initialization function that initializes the mutual-exclusion object mutex along with the empty and full semaphores.

The main() function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the main() function will sleep for a period of time and, upon awakening, will terminate the application. The main() function will be passed three parameters on the command line:

1. How long to sleep before terminating
2. The number of producer threads
3. The number of consumer threads

A skeleton for this function appears in Figure 3.

```
#include "buffer.h"
  /* the buffer */
buffer item buffer[BUFFER SIZE];
int insert item(buffer item item) {
  /* insert item into buffer
return 0 if successful, otherwise
return -1 indicating an error condition */
}
int remove item(buffer item *item) {
  /* remove an object from buffer placing it in item
return 0 if successful, otherwise
return -1 indicating an error condition */
}
```

Figure 2 Outline of buffer operations.

The Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the rand() function, which produces random integers between 0 and RAND MAX. The

consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer.
An outline of the producer and consumer threads appears in Figure 4

```
#include "buffer.h"
int main(int argc, char *argv[]) {
/* 1. Get command line arguments argv[1],argv[2],argv[3] */
/* 2. Initialize buffer */
/* 3. Create producer thread(s) */
/* 4. Create consumer thread(s) */
/* 5. Sleep */
/* 6. Exit */
}
```

Figure 3 Outline of skeleton program.

```
#include <stdlib.h> /* required for rand() */
#include "buffer.h"
void *producer(void *param) {
buffer item item;
while (true) {
/* sleep for a random period of time */
sleep(...);
/* generate a random number */
item = rand();
if (insert item(item))
fprintf("report error condition");
else
printf("producer produced %d\n",item);
}
void *consumer(void *param) {
buffer item item;
while (true) {
/* sleep for a random period of time */
sleep(...);
if (remove item(&item))
fprintf("report error condition");
else
printf("consumer consumed %d\n",item);
}
```

Figure 4 An outline of the producer and consumer threads.

Programming Projects on Operating System (Optional)

A. Introduction to Linux Kernel Modules

In this project, you will learn how to create a kernel module and load it into the Linux kernel. You will then modify the kernel module so that it creates an entry in the /proc file system. The project can be completed using the Linux virtual machine that is available with this text. Although you may use any text editor to write these C programs, you will have to use the ***terminal*** application to compile the programs, and you will have to enter commands on the command line to manage the modules in the kernel.

As you'll discover, the advantage of developing kernel modules is that it is a relatively easy method of interacting with the kernel, thus allowing you to write programs that directly invoke kernel functions. It is important for you to keep in mind that you are indeed writing ***kernel code*** that directly interacts with the kernel. That normally means that any errors in the code could crash the system! However, since you will be using a virtual machine, any failures will at worst only require rebooting the system.

I. Kernel Modules Overview

The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel. You can list all kernel modules that are currently loaded by entering the command

lsmod

This command will list the current kernel modules in three columns:
name, size, and where the module is being used.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
/* This function is called when the module is loaded. */
int simple init(void)
{
printk(KERN INFO "Loading Kernel Module\n");
return 0;
}
/* This function is called when the module is removed. */
void simple exit(void)
{
printk(KERN INFO "Removing Kernel Module\n");
}
/* Macros for registering module entry and exit points. */
module init(simple init);
module exit(simple exit);
MODULE LICENSE("GPL");
MODULE DESCRIPTION("Simple Module");
MODULE AUTHOR("SGG");
```

Figure 1 Kernel module simple.c.

The program in Figure 1 (named simple.c and available with the source code for this text) illustrates a

very basic kernel module that prints appropriate messages when it is loaded and unloaded. The function simple init() is the **module entry point**, which represents the function that is invoked when the module is loaded into the kernel. Similarly, the simple exit() function is the **module exit point**—the function that
is called when the module is removed from the kernel.
The module entry point function must return an integer value, with 0 representing success and any other value representing failure. The module exit point function returns void. Neither the module entry point nor the module exit point is passed any parameters. The two following macros are used for registering the module entry and exit points with the kernel:

```
module init(simple init)
module exit(simple exit)
```

Notice in the figure how the module entry and exit point functions make calls to the printk() function. printk() is the kernel equivalent of printf(), but its output is sent to a kernel log buffer whose contents can be read by
the dmesg command. One difference between printf() and printk() is that printk() allows us to specify a priority flag, whose values are given in the <linux/printk.h> include file. In this instance, the priority is KERN INFO, which is defined as an ***informational*** message.
The final lines—MODULE LICENSE(), MODULE DESCRIPTION(), and MODULE AUTHOR()—represent details regarding the software license, description of the module, and author. For our purposes, we do not require this information, but we include it because it is standard practice in developing kernel modules.
This kernel module simple.c is compiled using the Makefile accompanying the source code with this project. To compile the module, enter the following on the command line:

```
make
```

The compilation produces several files. The file simple.ko represents the compiled kernel module. The following step illustrates inserting this module into the Linux kernel.

II. Loading and Removing Kernel Modules

Kernel modules are loaded using the insmod command, which is run as follows:

```
sudo insmod simple.ko
```

To checkwhether themodule has loaded, enter the lsmod command and search for the module simple. Recall that the module entry point is invoked when the module is inserted into the kernel. To check the contents of this message in the kernel log buffer, enter the command

```
dmesg
```

You should see the message "Loading Module." Removing the kernel module involves invoking the rmmod command (notice that the .ko suffix is unnecessary):

```
sudo rmmod simple
```

Be sure to check with the dmesg command to ensure the module has been removed. Because the kernel log buffer can fill up quickly, it often makes sense to clear the buffer periodically. This can be accomplished as follows:

```
sudo dmesg -c
```

Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the contents of the kernel log buffer using dmesg to ensure that you have followed the steps properly.
As kernel modules are running within the kernel, it is possible to obtain values and call functions that are available only in the kernel and not to regular user applications. For example, the Linux include file <linux/hash.h> defines several hashing functions for use within the kernel. This file also defines the
constant value GOLDEN RATIO PRIME (which is defined as an unsigned long).

This value can be printed out as follows:

```
printk(KERN INFO "%lu\n", GOLDEN RATIO PRIME);
```

As another example, the include file <linux/gcd.h> defines the following function

```
unsigned long gcd(unsigned long a, unsigned b);
```

which returns the greatest common divisor of the parameters a and b.
Once you are able to correctly load and unload your module, complete the following additional steps:

1. Print out the value of GOLDEN RATIO PRIME in the simple init() function.
2. Print out the greatest common divisor of 3,700 and 24 in the simple exit() function.

As compiler errors are not often helpful when performing kernel development, it is important to compile your program often by running make regularly. Be sure to load and remove the kernel module and check the kernel log buffer using dmesg to ensure that your changes to simple.c are working properly.

B. The Dining-Philosophers Problem

In Section 7.1.3, an outline of a solution to the dining-philosophers problem using monitors is presented. This project involves implementing a solution to this problem using either POSIX mutex locks and condition variables or Java condition variables. Solutions will be based on the algorithm illustrated in Figure 7.7.
The implementation will require creating five philosophers, each identified by a number 0...4. Each philosopher will run as a separate thread. Philosophers alternate between thinking and eating. To simulate both activities, have each thread sleep for a random period between one and three seconds.

Java

When a philosopher wishes to eat, she invokes the method take Forks(philosopherNumber), where philosopherNumber identifies themnumber of the philosopher wishing to eat. When a philosopher finishes eating, she invokes returnForks(philosopherNumber).

Your solution will implement the following interface:

```
public interface DiningServer
{
/* Called by a philosopher when it wishes to eat */
public void takeForks(int philosopherNumber);
/* Called by a philosopher when it is finished eating */
public void returnForks(int philosopherNumber);
}
```

It will require the use of Java condition variables.