34 MARCH, 2023

LABORATORY 2: C

OPERATING SYSTEMS

FRANCISCO SECCHI - ALEX HERNANDEZ
TECNOCAMPUS MARESME-MATARÓ

Index

Introduction

Repository: https://github.com/FranSecchi/Lab2-C

In this report we are going to be explaining the second C laboratory.

This laboratory also consists in 3 activities, overall we are going to show how the libraries of the GNU/Linux operating system to create, use and destroy processes and threads, pipes and shared memory and semaphores and mutexes.

Compiling C-files

Note that to compile our code each activity has a Makefile file, which will compile every c-program on the activity folder. We do it so by using the *make* command on our prompt, as:

"make <file-to-compile> -f act#.Makefile".

Every Makefile will follow a similar structure as:

We can also compile every file and get our executables by manually doing so, inputing in our prompt in the project directory:

"gcc -o <exe-name> <c-file>".

```
devasc@fsecchi:~/labs/devnet-src/c/lab2/activity2$ gcc activity2.c -o activity2
devasc@fsecchi:~/labs/devnet-src/c/lab2/activity2$ ./activity2 3
```

Although, when using threads we need to specify in the command prompt that we are using the -pthread library (activities 1 and 3):

"gcc <exe-name> -o <c-file> -pthread".

```
devasc@fsecchi:~/labs/devnet-src/c/lab2/activity1$ gcc activity1.c -o activity1 -lrt -pthread
devasc@fsecchi:~/labs/devnet-src/c/lab2/activity1$ ./activity1
```

The -ltr option specifies that the program should be linked against the *librt* library, which provides support for real-time signals, clocks, and timers.

Activity 1

The goal of activity1 is to implement a program that creates two processes and demonstrates how they can communicate using pipes.

We implemented all the logic in a single *main* method as follows:

```c
#define SHM_NAME "/activity1_shm"
#define SEM1_NAME "/activity1_sem1"
#define SEM2_NAME "/activity1_sem2"

int main(int argc, char** argv) {
    srand(time(NULL));
    // Create and initialize semaphores
    sem_t* sem1 = sem_open(SEM1_NAME, O_CREAT, 0644, 1);
    if (sem1 == SEM_FAILED) {
        perror("sem_open");
        exit(1);
    }
    sem_t* sem2 = sem_open(SEM2_NAME, O_CREAT, 0644, 0);
    if (sem2 == SEM_FAILED) {
        perror("sem_open");
        exit(1);
    }
```

We first define the shared memory and the semaphores to be created later.

Once in the *main* method, we start our seed for creating random values and we initialize our 2 semaphores.

```c
// Create and initialize shared memory
int fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0644);
if (fd == -1) {
    perror("shm_open");
    exit(1);
}
if (ftruncate(fd, sizeof(int)) == -1) {
    perror("ftruncate");
    exit(1);
}
int* shared_num = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (shared_num == MAP_FAILED) {
    perror("mmap");
    exit(1);
}
*shared_num = rand() % 20 + 11; // Generate a random number between 11 and 20
```

Then we initialize the shared memory using the method shm_open() and mmap() to reset the memory values. And we set it as a random number between 11 and 20 for our little bouncing game.

```c
// Create child process
pid_t pid = fork();
if (pid == -1) {
    perror("fork");
    exit(1);
}
```

Using the method fork() we instantiate a child process, if the PID is -1, an error has occurred.

```c
if (pid == 0) { // Child process
    for (;;) {
        sem_wait(sem2); // Wait for parent's turn
        if (*shared_num == 0) {
            break;
        }
        (*shared_num)--;
        printf("child (pid = %d) bounce %d.\n", getpid(), *shared_num);
        sem_post(sem1); // Notify parent's turn
    }
    printf("child (pid = %d) ends.\n", getpid());
    sem_close(sem1);
    sem_close(sem2);
    exit(0);
```

The child process will enter an endless loop, waiting for the parent's turn to end (semaphore 2 signal), then decreasing the shared number by 1, print it, and notify the parent's turn with a signal on the first semaphore. Once the number has reached 0, we can end the process and close the semaphores releasing the resources.

```c
} else { // Parent process
    printf("parent (pid = %d) begins.\n", getpid());
    for (;;) {
        if (*shared_num == 0) {
            break;
        }
        (*shared_num)--;
        printf("parent (pid = %d) bounce %d.\n", getpid(), *shared_num);
        sem_post(sem2); // Notify child's turn
        sem_wait(sem1); // Wait for child's turn
    }

    printf("parent (pid = %d) ends.\n", getpid());
    sem_post(sem2);
    sem_close(sem1);
    sem_close(sem2);
    munmap(shared_num, sizeof(int));
    close(fd);
    shm_unlink(SHM_NAME);
}
```

The parent process works just the same, only that we notify the child's turn after the first decreasing of the shared number and then, we wait for the child's turn to be over to start again the loop.

Testing

This piece of code makes 2 processes, parent and child, bounce a number by decreasing it by one in turns, using shared memory variables:

```
devasc@fsecchi:~/labs/devnet-src/c/lab2/activity1$ ./activity1
parent (pid = 3235) begins.
parent (pid = 3235) bounce 15.
child (pid = 3236) bounce 14.
parent (pid = 3235) bounce 13.
child (pid = 3236) bounce 12.
parent (pid = 3235) bounce 11.
child (pid = 3236) bounce 10.
parent (pid = 3235) bounce 9.
child (pid = 3236) bounce 8.
parent (pid = 3235) bounce 7.
child (pid = 3236) bounce 6.
parent (pid = 3235) bounce 5.
child (pid = 3236) bounce 4.
parent (pid = 3235) bounce 3.
child (pid = 3236) bounce 2.
parent (pid = 3235) bounce 1.
child (pid = 3236) bounce 0.
parent (pid = 3235) ends.
devasc@fsecchi:~/labs/devnet-src/c/lab2/activity1$ child (pid = 3236) ends.
```

Activity 2

The goal of activity2 is to demonstrate interprocess communication using named pipes in a C program. The program creates two processes, parent and child, and establishes a named pipe between them for communication.

This time we implemented this activity with a couple of useful methods. But first we defined some necessary global variables, both operands, the operation as a char and the pipe_fd array for the file descriptors:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <time.h>
#include <stdint.h>

uint8_t operand1, operand2;
char operation;
int pipe_fd[2];
```

Once in our *main* we want to get the number of iterations from the prompt's first argument, and initialize our named pipe using the function pipe() to get the file descriptor. Then we fork to get the child process:

```c
int num_iterations = atoi(argv[1]);
pid_t pid;

if (pipe(pipe_fd) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
printf("main: created pipe.\n");

pid = fork();
if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}
```

We will handle both, parent and child in separated methods:

```c
if (pid > 0) { // parent process
    parent_process(num_iterations);
    exit(EXIT_SUCCESS);
}
else { // child process
    child_process(num_iterations);
    exit(EXIT_SUCCESS);
}
return 0;
```

The parent process will generate random operands and operation, write them in our pipe and print it in the prompt. Then it sleeps for 1 second giving time for the child to answer:

```c
void parent_process(int iterations) {
    close(pipe_fd[0]); // close unused read end of the pipe
    printf("parent (pid = %d) begins.\n", getpid());

    for (int i = 0; i < iterations; i++) {
        generate_random_values();
        printf("parent (pid = %d): iteration %d.\n", getpid(), i);

        // write operand1, operation and operand2 to the pipe
        write(pipe_fd[1], &operand1, sizeof(operand1));
        write(pipe_fd[1], &operation, sizeof(operation));
        write(pipe_fd[1], &operand2, sizeof(operand2));

        if (operation == '+') {
            printf("parent (pid = %d): %d + %d = ?\n", getpid(), operand1, operand2);
        }
        else if (operation == '-') {
            printf("parent (pid = %d): %d - %d = ?\n", getpid(), operand1, operand2);
        }
        else if (operation == '*') {
            printf("parent (pid = %d): %d * %d = ?\n", getpid(), operand1, operand2);
        }
        else if (operation == '/') {
            printf("parent (pid = %d): %d / %d = ?\n", getpid(), operand1, operand2);
        }
        else {
            printf("parent (pid = %d): Invalid operation %c\n", getpid(), operation);
        }

        sleep(1);
    }

    close(pipe_fd[1]); // close write end of the pipe
    printf("parent (pid=%d) ends.\n", getpid());
}
```

It will repeat this logic until it has reached the desired number of iterations, then close the write end of the pipe.

In the other hand, the child process closes the unused write end of the pipe and reads the operations and operation into a variable:

```c
void child_process(int iterations) {
    close(pipe_fd[1]); // close unused write end of the pipe
    printf("child (pid = %d) begins.\n", getpid());

    uint8_t a, b;
    char op;

    for (int i = 0; i < iterations; i++) {
        read(pipe_fd[0], &a, sizeof(a));
        read(pipe_fd[0], &op, sizeof(op));
        read(pipe_fd[0], &b, sizeof(b));

        int result;
        switch (op) {
            case '+': result = a + b; break;
            case '-': result = a - b; break;
            case '*': result = a * b; break;
            case '/': result = a / b; break;
        }
         printf("child (pid = %d): %d %c %d = %d\n", getpid(), a, op, b, result);
    }
    close(pipe_fd[0]); // close read end of pipe
    printf("child (pid=%d) ends.\n", getpid());
}
```

Then, it performs the operation and prints the result and repeats for every iteration. Once done, closes the read end of the pipe.

Testing

As we can see next, we input 3 iterations and the parent and child process interact three times, parent asking for a random operation and child answering them:

```
devasc@fsecchi:~/labs/devnet-src/c/lab2/activity2$ gcc activity2.c -o activity2
devasc@fsecchi:~/labs/devnet-src/c/lab2/activity2$ ./activity2 3
main: created pipe.
parent (pid = 4992) begins.
parent (pid = 4992): iteration 0.
parent (pid = 4992): 100 + 92 = ?
child (pid = 4993) begins.
child (pid = 4993): 100 + 92 = 192
parent (pid = 4992): iteration 1.
parent (pid = 4992): 78 / 1 = ?
child (pid = 4993): 78 / 1 = 78
parent (pid = 4992): iteration 2.
parent (pid = 4992): 4 + 14 = ?
child (pid = 4993): 4 + 14 = 18
child (pid=4993) ends.
parent (pid=4992) ends.
```

Activity 3

The goal of activity3 to implement the bouncing ball game from activity 1 using threads instead of processes, and mutexes instead of semaphores, to synchronize access to the shared variable.

We use the following data structure for the creation of two threads, whom will execute a thread_bouncing function that contains the bouncing logic from activity 1:

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

typedef struct {
    uint32_t * data_ptr;
    pthread_mutex_t * mutex1;
    pthread_mutex_t * mutex2;
} thread_data_t;
```

In our *main* we basically define and initialize two mutexes and threads, and the times we bounce will be random between 11 and 20:

```c
int main() {
    uint32_t data = rand()%20 + 11;
    pthread_mutex_t mutex1, mutex2;
    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);
    pthread_t thread1, thread2;
    thread_data_t data1 = {&data, &mutex1, &mutex2};
    thread_data_t data2 = {&data, &mutex2, &mutex1};
    printf("main: bouncing for %d times.\n", data);
```

Note that each thread data saves both mutexes, but in different order.

Then we create two new threads of execution, wait for them to terminate with pthread_join() and then destroy the mutexes to release the resources they are holding:

```c
pthread_create(&thread1, NULL, thread_func1, &data1);
pthread_create(&thread2, NULL, thread_func2, &data2);
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_mutex_destroy(&mutex1);
pthread_mutex_destroy(&mutex2);
exit(0);
```

Both threads of execution are 2 methods that receive a pointer to a thread_data_t structure as its argument, which contains a pointer to the shared value to be decremented, and two

mutexes to synchronize the threads and ensure mutual exclusion when accessing the shared value.

```
void * thread_func1(void * arg) {
    thread_data_t * data = (thread_data_t *) arg;
    uint32_t * val = data->data_ptr;
    pthread_mutex_t * mutex1 = data->mutex1;
    pthread_mutex_t * mutex2 = data->mutex2;
    bounce(data, 1);
    pthread_exit(NULL);
}

void * thread_func2(void * arg) {
    thread_data_t * data = (thread_data_t *) arg;
    uint32_t * val = data->data_ptr;
    pthread_mutex_t * mutex1 = data->mutex1;
    pthread_mutex_t * mutex2 = data->mutex2;
    bounce(data, 2);
    pthread_exit(NULL);
}
```

We will have the bouncing logic in a separated method.

The function first retrieves the shared value and enters a loop that continues until the shared value reaches zero. Inside the loop, the function first acquires mutex1 to ensure mutual exclusion when accessing the shared value. It then checks if the value is greater than zero, decrements it, and prints the current value along with the thread ID.

```
void* bounce(void* arg, int thread_id) {
    thread_data_t* data = (thread_data_t*) arg;
    uint32_t* count = data->data_ptr;
    pthread_mutex_t* mutex1 = data->mutex1;
    pthread_mutex_t* mutex2 = data->mutex2;

    uint32_t bounce_count = *count;
    printf("thread%d begins, %d.\n", thread_id, bounce_count);
    while (bounce_count > 0) {
        pthread_mutex_lock(mutex1);
        if (*count == 0) {
            pthread_mutex_unlock(mutex2);
            break;
        }
        (*count)--;
        printf("thread%d bounce %d.\n", thread_id, *count);
        pthread_mutex_unlock(mutex2);
    }
    printf("thread%d ends.\n", thread_id);
    pthread_mutex_unlock(mutex1);
    pthread_exit(NULL);
}
```

After releasing mutex1, the function acquires mutex2, which allows the other thread to access the shared value. It then checks again if the value is greater than zero, decrements it, and prints the current value along with the thread ID.

Finally, the function releases mutex2, allowing the other thread to acquire it and repeat the same process. The loop continues until the shared value reaches zero, at which point the function ends and the thread terminates.

Testing

```
devasc@fsecchi:~/labs/devnet-src/c/lab2/activity3$ ./activity3
main: bouncing for 14 times.
thread1 begins, 14.
thread1 bounce 13.
thread2 begins, 13.
thread2 bounce 12.
thread1 bounce 11.
thread2 bounce 10.
thread1 bounce 9.
thread2 bounce 8.
thread1 bounce 7.
thread2 bounce 6.
thread1 bounce 5.
thread2 bounce 4.
thread1 bounce 3.
thread2 bounce 2.
thread1 bounce 1.
thread2 bounce 0.
thread1 ends.
thread2 ends.
```