Homework 1

CSE 4600 (Section 03) – Operating Systems – Fall 2023

Submitted to

Department of Computer Science and Engineering California State University, San Bernardino, California

by

Benjamin Harrity (007930709)
Fruzsina Ladanyi (008455051)
Purav Parab (008418118)

Date: September 27, 2023

Emails:
• 007930709@coyote.csusb.edu
• 008455051@coyote.csusb.edu
• purav.parab8118@coyote.csusb.edu

Part 1

1.1 Process creation via fork():

```
> make -s
> ./main
Hello from PID: 6696, PPID: 1
Hello from PID: 6697, PPID: 6696
Hello from PID: 6699, PPID: 6697
Hello from PID: 6698, PPID: 6696
```

```
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_1$ gcc process_creation.c
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_1$ ./a.out
Hello from PID: 3399, PPID: 2104
Hello from PID: 3401, PPID: 3399
Hello from PID: 3400, PPID: 3399
Hello from PID: 3402, PPID: 3400
Hello from PID: 3404, PPID: 3402
Hello from PID: 3403, PPID: 3400
```

The original program has a Process ID (PID) of 6696 and creates a child with PID 6697. This child, based on receiving a value of zero from the fork() function, further creates its own child, or grandson, with PID 6699. Additionally, the original program creates another direct child with PID 6698.

The potential second grandson is missing because the code checks the son's PID, 6697, to determine if it's even. Since 6697 is odd, this second grandson creation skipped, which is why there's no additional child with a PPID of 6697 in the output.

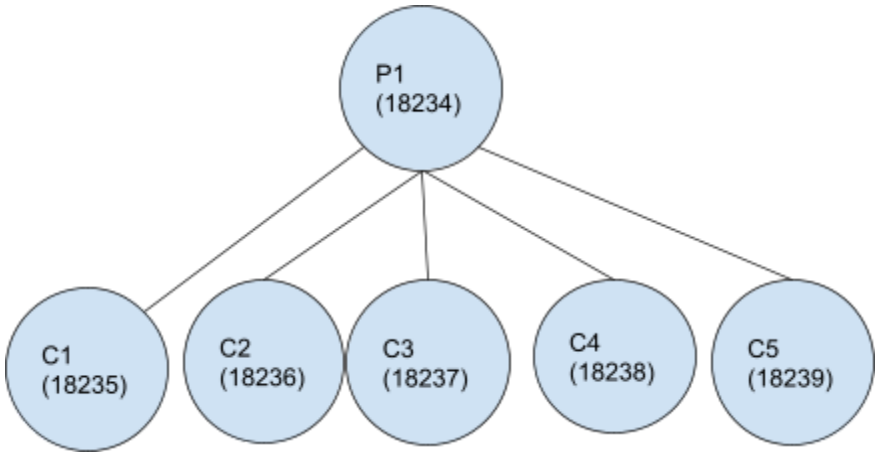
<u>Contributions:</u>

- Purav: Added console output
- Fruzsina: Added comments for code
- Benjamin: Wrote program description and added console output

1.2:

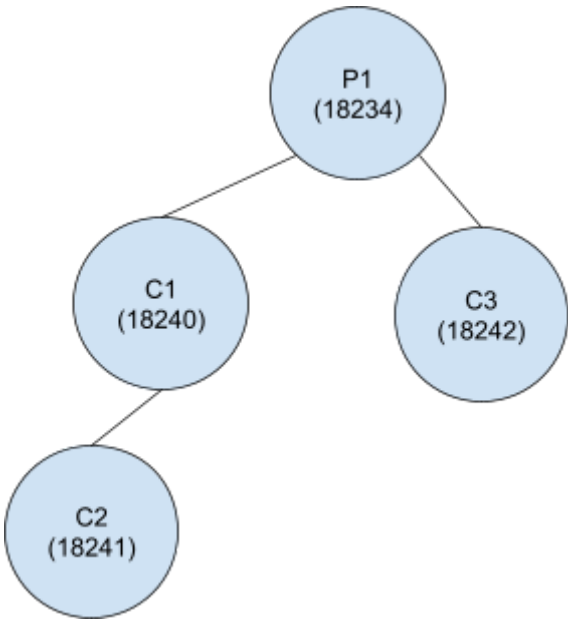
```
Create 3 processes from the same parent process:
Child Process (ID): 18235, parent process (ID): 18234
Child Process (ID): 18236, parent process (ID): 18234
Child Process (ID): 18237, parent process (ID): 18234
Child Process (ID): 18238, parent process (ID): 18234
Child Process (ID): 18239, parent process (ID): 18234

Create a binary tree of the processes:
1. Child Process (ID): 18240, parent process (ID): 18234
2. Child Process (ID): 18240, parent process (ID): 18234
2. Child Process (ID): 18241, parent process (ID): 18240
1. Child Process (ID): 18242, parent process (ID): 18234

Star topology of processes:
Child Process (ID): 18247, parent process (ID): 18234
Child Process (ID): 18246, parent process (ID): 18234
Child Process (ID): 18245, parent process (ID): 18234
Child Process (ID): 18244, parent process (ID): 18234
Child Process (ID): 18243, parent process (ID): 18234
```

Linear topology of processes



The first part of the code creates a linear topology of the processes. Each time the loop runs, a child process is created. The algorithm checks if it is working with a child process and outputs the Child Process ID and the parent process ID. The parent (18234) waits for each child to exit before moving on with the loop, creating the next child process.

Binary Tree of the processes

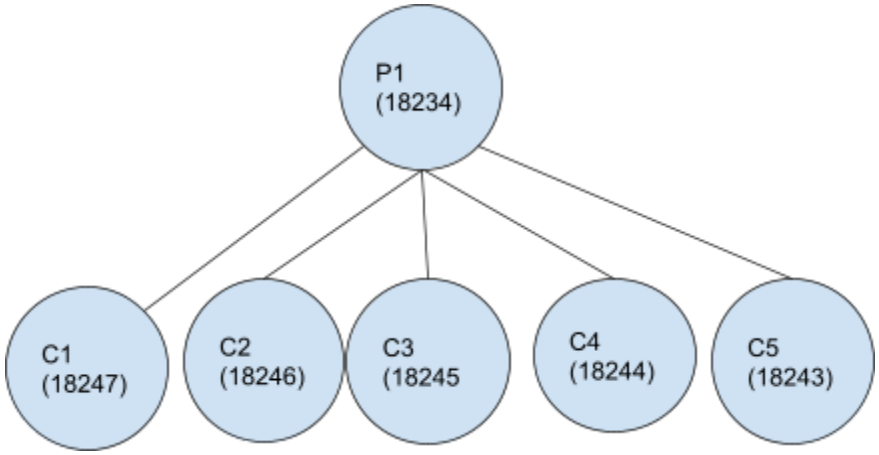


The second part of the code creates a binary tree of processes. For the first time through the loop, a child process is created (18240). As this is a child process, it prints out its own ID (18240) and its parent's ID (18234).

After this, we enter the if statement, as this is our first time through the loop (i==0). Another child process (18241) is created from the first child process (18240). The algorithm now runs for 18240 and prints out its ID (18240) and the parent's ID (18234). It also runs for the grandchild (18241) so it prints out its ID (18241) and the parent ID (18240). The parent process waits as the child process exits.

After this, we go through the loop one more time. This is when the next child process (18242) is created directly from the parent. The ID of the child process (18242) and its parent process (18234) is printed. We don't enter the if statement for the second time as i>0, and the child process exits.

Star topology of the processes



The third part of the code produces a star topology of the processes. Although the graph looks similar to the linear topology, the order of execution is different. Similarly to the linear topology, each time through the loop a child process is created. The algorithm checks if it's working with a child process and if so, it outputs its ID and the parent's ID. The parent however, might proceed to execute the code and create children before the other children exit. The parent waits at the very end of the algorithm for all children to exit.

Contributions:

- Fruzsina: Added graphs, description, and console output

2. Replacing a Process Image:

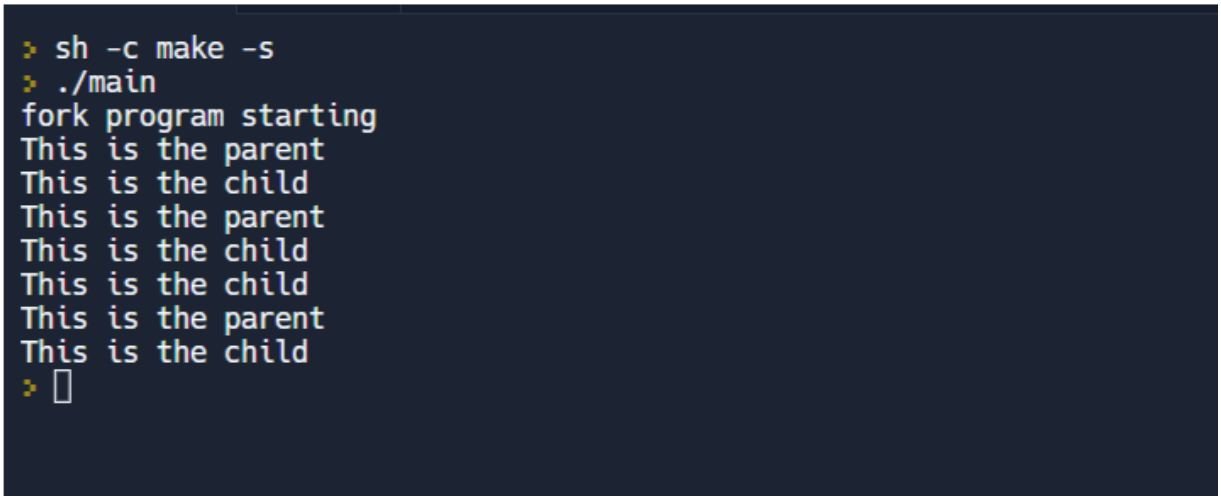
Original code can be found in test_exec.cpp.

Modified codes can be found in test_exec1.cpp, test_exec2.cpp, test_exec3.cpp and test_exec4.cpp.

Contributions:

- Fruzsina: Modified codes

3. Duplicating a Process Image:

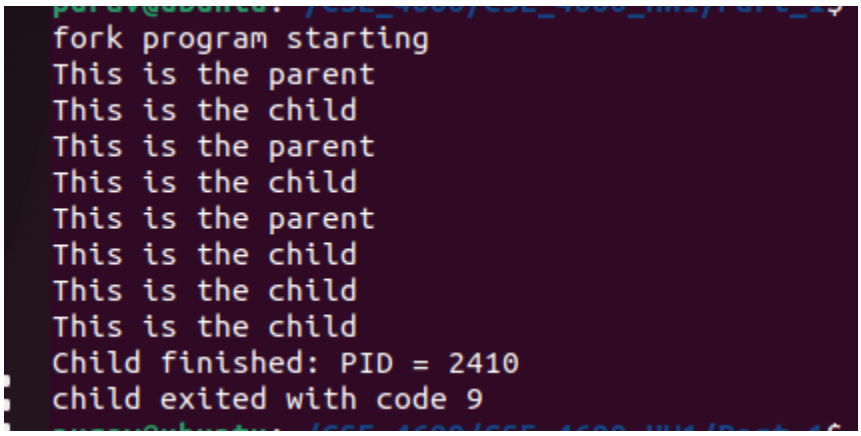


When the code is executed, both the parent and child processes initiate tasks simultaneously following the fork. The parent is set to display its message three times, while the child aims for the five. Due to the system's scheduling, the sequence of their messages can vary with each run. But the child's message appears only four times; it suggests that the parent finished early, potentially interrupting the child's output. To fix this, a wait function can help the child finish its output.

Contributions:

- Benjamin: Worked on this section

4. Waiting for a Process:



When the code is executed, the parent and child both print out messages simultaneously. The parent prints out its message 3 times and the child prints out its message 5 times. At the end, the child ends its process normally and the parent prints the child's PID and exit code 9.

After program is modified:

Contributions:

- Purav: Worked on this section

5. Signals:

```
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_1$ ./a.out
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
^CReceived a signal from your program 2
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
^CReceived a signal from your program 2
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
CSUSB CSE 4600 Operating Systems Fall 2023
Received a signal from your program 2
CSUSB CSE 4600 Operating Systems Fall 2023
```

Every time ^C is pressed, it triggers a signal interrupt that displays a message with the signal number.

Sending Signals:

```
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_1$ g++ test_alarm.cpp
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_1$ ./a.out
Alarm testing!
Waiting for alarm to go off!
Alarm has gone off
Done!
```

The child process waits for 5 secs, then sends a kill call to its parent process with SIGALRM which triggers the signal alarm.

Contributions:

- Purav: Worked on the first two questions.
- Benjamin: Wrote the code for more robust signals interface.

Part 2

2. Process pipes:

```
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_2$ g++ pipe1.cpp
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_2$ ./a.out
Output from pipe: //pipe1.cpp
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <iostream>

using namespace std;

int main(){
        FILE *fpi; //for reading a pipe

        char buffer[BUFSIZ+1]; //BUFSIZ defined in <stdio.h>

        int chars_read;
        memset (buffer, 0,sizeof(buffer)); //clear buffer
        fpi = popen ("cat pipe1.cpp", "r"); //pipe to command "ps -auxw"
        if (fpi != NULL) {
                //read data from pipe into buffer
                chars_read = fread(buffer, sizeof(char), BUFSIZ, fpi);
                if (chars_read > 0){
                        cout << "Output from pipe: " << buffer << endl;
                }
                pclose (fpi); //close the pipe
                return 0;
        }
        return 1;
}
```

Executing pipe1.cpp outputs the code inside pipe1.cpp. This happens because the program uses popen() to read the output of “cat pipe1.cpp” into the buffer. This is then printed on the console.

After modifying the code:

```
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_2$ g++ pipe1a.cpp
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_2$ ./a.out cat pipe1.cpp
Output from pipe: //pipe1.cpp
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <iostream>

using namespace std;

int main(){
        FILE *fpi; //for reading a pipe

        char buffer[BUFSIZ+1]; //BUFSIZ defined in <stdio.h>

        int chars_read;
        memset (buffer, 0,sizeof(buffer)); //clear buffer
        fpi = popen ("cat pipe1.cpp", "r"); //pipe to command "ps -auxw"
        if (fpi != NULL) {
                //read data from pipe into buffer
                chars_read = fread(buffer, sizeof(char), BUFSIZ, fpi);
                if (chars_read > 0){
                        cout << "Output from pipe: " << buffer << endl;
                }
                pclose (fpi); //close the pipe
                return 0;
        }
        return 1;
}
```

pipe1a.cpp takes in a command such as cat pipe1.cpp and prints out the output to the console.

Contributions:

- Purav: Worked on this section.

3. The pipe call:

```
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_2$ g++ pipe3.cpp
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_2$ ./a.out
Sent 18 bytes to pipe.
Read 18 from pipe: CSE 4600 Fall 2023
```

Pipe3.cpp outputs two strings indicating the amount of bits written to the pipe and read from the pipe. When the pipe is read from, the program also prints out the string s.

After modifying the code:

```
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_2$ g++ pipe3.cpp
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_2$ ./a.out 10 5
Sent the numbers 10 and 5 to pipe for their summation.
The sum of the two numbers is: 15
```

The code is modified to take in two input integers and write and read its sum using a pipe.

Contributions:

- Purav: Worked on this section.

4. Parent and Child Processes:

The updated code sends a message you type from one part of the program to another. First, it writes the message you put in when you start the program. Then a new 'child' part of the program reads this message, adds a welcome message to it, and then displays it. If there's a problem making the 'child' part or the special channel (pipe) it uses, or if you forgot to type a message when you started the program, you'll see an error message.

Contributions:

- Benjamin: Worked on this section.

5. Sum of a large array via pipes:

Output from original code when array size is 48:

```
Partial sum is: 222Partial sum is: 366Partial sum is: 510Partial sum is: 78Total sum: 664
```

Since the wait() call can only receive signal values of up to 255 the program isn't able to handle partial sums 366 and 510. Therefore, the Total sum value is incorrect.

Output after modifying the code using pipes:

```
Partial sum is: 78
Partial sum is: 300
Partial sum is: 666
Total sum is: 1176
```

Contributions:

- Fruzsina: Worked on this section.

Part 3

1. Pthreads:

```
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_3$ ./pthreads_demo
Sum of the numbers from 1 to 10: 55
Product of the numbers from 1 to 10: 3628800
```

pthreads_demo.cpp runs two threads (sum and product) and outputs their result. The output is determined by what thread the CPU scheduler decides to run first.

```
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_3$ g++ -o pthreads_demo pthreads_demo.cpp -lpthread
purav@ubuntu:~/CSE_4600/CSE_4600_HW1/Part_3$ ./pthreads_demo
Product of the numbers from 1 to 10: 3628800
Sum of the numbers from 1 to 10: 55
Average of the numbers from 1 to 10: 5.5
```

After modifying the code, the program outputs the results of three threads, including the average thread. Once again, the order of the output depends on the CPU scheduler.

Contributions:

- Purav: Worked on this section

2. Print message from the parent and child thread

1. In pthreads_message_ demo.cpp, the console may unpredictably display one of two messages of either: “You are in thread function” or “You are in the main function.” This happens because the program starts two threads that try to work on the same piece of data at the same time. Which message gets shown depends on how the CPU scheduler picks the order of the threads.

2. To consistently display the message from the thread function, you need to rearrange the sequence of operations in the main function. To fix this, you can relocate the printf statement below the pthread_join function call. This change makes sure that the main part of the program, which prints the message, waits until the new part is done doing its job. This ensures that the message “You are in thread function!” gets changed as needed before the program shows on screen.

Contributions:

- Benjamin: Worked on this section

3. Synchronization using Pthreads mutex

```
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$ ./sha
red_resource_mutex
Shared resource value: 26273388
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$ ./sha
red_resource_mutex
Shared resource value: 4236666
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$ ./sha
red_resource_mutex
Shared resource value: 15897123
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$ ./sha
red_resource_mutex
Shared resource value: -30799410
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$ ./sha
red_resource_mutex
Shared resource value: 36952430
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$
```

1. The code has two threads changing the same shared resource at the same time without taking turns. This messes up the final number because the threads are overwriting each other's work. To fix it, they need to take turns using something in the code that makes one wait for the other to finish.

```
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$ time
./shared_resource_mutex
Shared resource value: 2511702

real    0m1.123s
user    0m1.095s
sys     0m0.000s
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$
```

2. After running the time command I received three different times: real, user, and sys.

3. Switching the order of pthread_join commands does not matter. Both ways, your program will wait for two tasks to finish, without caring which finishes first. The order doesn't cause randomness; that's the result of different issues in the code like missing synchronization when accessing the same data from different threads. Basically, the order of pthread_join doesn't influence the outcome of the results.

4. Modified code

```
void* inc_dec_resource(void* arg){
        //get the pointer from main thread and dereference it to put the v
        int resource_value = *(int *) arg;
        for(int i=0; i < iterations; i++){
                pthread_mutex_lock(&mutex);
                shared_resource += resource_value;
                pthread_mutex_unlock(&mutex);
        }
        pthread_exit(NULL);
}
```

Executed Code

```
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$ time
./shared_resource_mutex
Shared resource value: 0

real    0m7.661s
user    0m7.555s
sys     0m0.022s
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$ time
./shared_resource_mutex
Shared resource value: 0

real    0m7.324s
user    0m7.241s
sys     0m0.000s
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$ time
./shared_resource_mutex
Shared resource value: 0

real    0m7.487s
user    0m7.369s
sys     0m0.041s
benjamin@benjamin-VirtualBox:~/Documents/new_project/CSE_4600_HW1/Part_3$
```

Contributions:

- Benjamin: Worked on this section

4. Synchronization using Pthreads mutex

Code can be found in Part_3/partial_sum_threads.cpp

Contributions:

- Fruzsina: Worked on this section.