Homework 1

CSE 4600 (01) – Operating Systems – Spring 2022

Submitted to

Department of Computer Science and Engineering

California State University, San Bernardino, California

by

Nathan Bush ID: 007463099

Date: March 4th, 2022

Email

- nathan.bush3099@coyote.csusb.edu

Code

- https://github.com/natebush707/CSE_4600_HW1.git
- All executables and screenshots are also included in a .zip file with this submission.

Part 1 (4%)

1. Process creation via fork()

Requirement:

- How many processes does the following piece of code create? Why?

```
Int main(){
  fork();
  fork();
  fork();
  return 0;
}
```

    - This piece of code creates processes. With each fork, an identical process is created that continues to run code from the point at which it was forked. So at the first fork, the main process creates a second process. At the second fork call, both of those processes create a second process for a total of 4. At the third fork call, all 4 process create another identical process for a total of 8 processes.
- Write a C/C++-program that creates a chain of 10 processes and prints out their process ids and relationships. For example, process 1 is the parent of process 2, process 2 is the parent of process 3, process 3 is the parent of 4 and so on. Each child must print out all her ancestors identified by the process IDs.

Report:

This program achieves the above requirements by keeping track of each process as it is created in an array for ancestor reporting. A FOR loop is initialized to create the required chain of 10 processes, with each parent waiting while each child reports its new PID, the parent PID, and the array of its ancestor's PIDs. The child then stores its PID in the array for following children to recall. Once the required chain depth is reached, each waiting process successively exits.

Code:

```cpp
// Nathan Bush
// CSE 4600 Spring 2022
// Homework 1, Part 1
//
//
// hw1_part1_1.cpp
// Creates a chain of 10 processes and prints out their process ids and relationships.
// Each child must print out all of her ancestors identified by the process IDs

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

using namespace std;


// helper function to print the contents of an array of PIDs
void print_array(int (&pids)[10])
{
    for (int i = 0; i < 10; i++) {
        if (pids[i] != 0) {
            printf("%d ", pids[i]);
        }
    }
    printf("\n\n");
    return;
}


int main()
{
    // initialize array to store pids
    int pids[10] = {0};

    // store and report PID from original main process
    pids[0] = getpid();
    printf("Original process: %d\n...no parent\n...no ancestors\n\n", pids[0]);

    for (int i = 0; i < 9; i++) {
        int current_pid = fork();

        // child process, store and report PID and ancestry
        if (current_pid == 0) {
            printf("Child process: %d\n...parent: %d\n...ancestors: ", getpid(), pids[i]);
            print_array(pids);
            pids[i+1] = getpid();
        }

        // parent process, wait for child to finish before terminating
        else {
            wait(NULL);
            exit(0);
        }
    }
    return 0;
}
```

Run the executable by extracting it from the code and either double clicking part1_1.exe or running it from your terminal by navigating to its directory and typing ./part1_1.exe

Console output:

```
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$ ./part1_1.exe
Original process: 14507
...no parent
...no ancestors

Child process: 14508
...parent: 14507
...ancestors: 14507

Child process: 14509
...parent: 14508
...ancestors: 14507 14508

Child process: 14510
...parent: 14509
...ancestors: 14507 14508 14509

Child process: 14511
...parent: 14510
...ancestors: 14507 14508 14509 14510

Child process: 14512
...parent: 14511
...ancestors: 14507 14508 14509 14510 14511

Child process: 14513
...parent: 14512
...ancestors: 14507 14508 14509 14510 14511 14512

Child process: 14514
...parent: 14513
...ancestors: 14507 14508 14509 14510 14511 14512 14513

Child process: 14515
...parent: 14514
...ancestors: 14507 14508 14509 14510 14511 14512 14513 14514

Child process: 14516
...parent: 14515
...ancestors: 14507 14508 14509 14510 14511 14512 14513 14514 14515

nate@NDB-LAP:~/Desktop/CSE_4600_HW1$
```

2. Replacing a Process Image

The **exec** family of functions replace the current process with another which is created according to the arguments given to the **exec** function.

Study this family of functions by the "man" command:

$ man execl

```
//test_exec.cpp
#include <unistd.h>
#include <iostream>

using namespace std;

int main()
{
  cout << "Running ps with execlp\n";
  execlp ("ps", "ps", "-ax", 0);

  cout << "The program is exiting!\n";

  return 0;
}
```

Requirement:

Note that when you execute this program, the message " The program is exiting!" is not displayed. Why? This is because when **ps** finishes, we get a new shell prompt. We do not return to **test_exec**, so the second message doesn't get printed.

Modify **test_exec** so that the function **execl** is used instead of using **execlp**. (Refer to "man" for the syntax of the functions)

Report:

> Not much modification of code was necessary to use execl instead of execlp. I had to provide the path to the ps binary, as well as change the 0 to NULL for the arg terminator. Running the file will result in a list of processes running on the OS.

Code:

```cpp
// Nathan Bush
// CSE 4600 Spring 2022
// Homework 1, Part 2
//
//
// hw1_part1_2.cpp
// modify test_exec (original provided by Dr. Bilal Khan)
// so that the function execl is used instead of using execlp

#include <unistd.h>
#include <iostream>

using namespace std;

int main()
{
    cout << "Running ps with execl\n";
    execl ("/bin/ps", "ps", "-ax", NULL);

    cout << "The program is exiting!\n";
}
```

Run the executable by extracting it from the code and either double clicking part1_2.exe or running it from your terminal by navigating to its directory and typing ./part1_2.exe

Console output:

```
nate@NDB-LAP:~/Desktop/part 1$ ./part1_2.exe
Running ps with execl
    PID TTY      STAT   TIME COMMAND
      1 ?        Ss     0:03 /sbin/init splash
      2 ?        S      0:00 [kthreadd]
      3 ?        I<     0:00 [rcu_gp]
      4 ?        I<     0:00 [rcu_par_gp]
      6 ?        I<     0:00 [kworker/0:0H-events_highpri]
      9 ?        I<     0:00 [mm_percpu_wq]
     10 ?        S      0:00 [rcu_tasks_rude_]
     11 ?        S      0:00 [rcu_tasks_trace]
     12 ?        S      0:00 [ksoftirqd/0]
     13 ?        I      0:03 [rcu_sched]
     14 ?        S      0:00 [migration/0]
     15 ?        S      0:00 [idle_inject/0]
     16 ?        S      0:00 [cpuhp/0]
     17 ?        S      0:00 [cpuhp/1]
     18 ?        S      0:00 [idle_inject/1]
     19 ?        S      0:00 [migration/1]
     20 ?        S      0:00 [ksoftirqd/1]
     22 ?        I<     0:00 [kworker/1:0H-events_highpri]
     23 ?        S      0:00 [cpuhp/2]
     24 ?        S      0:00 [idle_inject/2]
     25 ?        S      0:00 [migration/2]
     26 ?        S      0:00 [ksoftirqd/2]
     28 ?        I<     0:00 [kworker/2:0H-events_highpri]
     29 ?        S      0:00 [cpuhp/3]
     30 ?        S      0:00 [idle_inject/3]
     31 ?        S      0:00 [migration/3]
     32 ?        S      0:00 [ksoftirqd/3]
     34 ?        I<     0:00 [kworker/3:0H-events_highpri]
     35 ?        S      0:00 [cpuhp/4]
     36 ?        S      0:00 [idle_inject/4]
     37 ?        S      0:00 [migration/4]
     38 ?        S      0:00 [ksoftirqd/4]
     40 ?        I<     0:00 [kworker/4:0H-events_highpri]
     41 ?        S      0:00 [cpuhp/5]
     42 ?        S      0:00 [idle_inject/5]
     43 ?        S      0:00 [migration/5]
     44 ?        S      0:00 [ksoftirqd/5]
     46 ?        I<     0:00 [kworker/5:0H-events_highpri]
     47 ?        S      0:00 [cpuhp/6]
     48 ?        S      0:00 [idle_inject/6]
     49 ?        S      0:00 [migration/6]
     50 ?        S      0:00 [ksoftirqd/6]
     52 ?        I<     0:00 [kworker/6:0H-events_highpri]
     53 ?        S      0:00 [cpuhp/7]
```

3. Duplicating a Process Image

We can use **fork** to duplicate a process. The duplicated process will be run independently.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

The call to **fork** in the parent returns the PID (process ID) of the new child process. The new child process continues to execute just like the original, with the exception that in the child process the call to the **fork** returns 0. In this way, the parent and the child can determine which is which.

Try the following code implements **fork**.

```
//test_fork.cpp
#include <sys/types.h>
#include <unistd.h>
#include <iostream>

using namespace std;

int main()
{
  pid_t pid;           //process id
  char *message;
  int n;
  cout << "fork program starting\n";
  pid = fork();
  switch (pid) {
  case -1:
    cout << "Fork failure!\n";
    return 1;
  case 0:
    message = "This is the child\n";
    n = 5;
    break;
  default:
    message = "This is the parent\n";
    n = 3;
    break;
  }
  for (int i = 0; i < n; ++i) {
    cout << message;
    sleep (1);
  }

  return 0;
}
```

Requirement:

Try the "test_fork.cpp" program and explain what you see on the screen.

Example output:

```
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$ ./test_fork
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$ This is the child
```

Explanation:

In this program, a fork call is made at which point the parent process prints “This is the parent” 3 times to the console with a 1 second delay before continuing. The child process prints “This is the child” 5 times to the console with a 1 second delay before continuing. Since the parent process obviously finishes first and is not told to wait on the child, it reaches the end of the program, returns 0 from the main, and the program exits. However, the child process (now an orphan) continues to execute its console outputs, with the final output printing on the command prompt that appears when the parent process exits the program execution.

4. Waiting for a Process

We can make a parent process waiting for its child to finish before continuing by using **wait**:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *stat_loc);
```

The status information is written to **stat_loc**.

Try the following program:

```
//test_wait.cpp
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace std;

int main()
{
  pid_t pid;           //process id
  char *message;
  int n;
  int exit_code;

  cout << "fork program starting\n";
  pid = fork();
  switch (pid) {
  case -1:
    cout << "Fork failure!\n";
    return 1;
  case 0:
    message = "This is the child\n";
    n = 5;
    exit_code = 9;
    break;
  default:
    message = "This is the parent\n";
    n = 3;
    exit_code = 0;
    break;
  }
  for (int i = 0; i < n; ++i) {
    cout << message;
    sleep (1);
  }
```

```
  //waiting for child to finish
  if (pid != 0) {      //parent
    int stat_val;
    pid_t child_pid;

    child_pid = wait (&stat_val); //wait for child
    cout << "Child finished: PID = " << child_pid << endl;
    if (WIFEXITED (stat_val))
      cout << "child exited with code " << WEXITSTATUS (stat_val) <<
endl;
    else
      cout << "child terminated abnormally!" << endl;
  }
  exit (exit_code);
}
```

Requirement:

Run the program and explain what you have seen on the screen.

Example output:

```
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$ ./test_wait
fork program staring
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child finished: PID = 17156
child exited with code 9
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$
```

Explanation:

In this execution, the process is forked and similar to the previous example, the parent and child both print to the console identifying which process is printing, with a 1 second delay. Again, the parent finishes first, but this time is told to wait on the child to finish its execution. Once the child finishes its output loop, it prints an additional message with its PID and exits with a console

message containing a status code. Once the child process exits, the parent finishes its wait and terminates.

Modify the program so that the child process creates another child and wait for it. The grandchild prints out the IDs of itself, its parent and grandparent.

Report:

This program took me a bit longer to write given the requirement that the grandchild process be the one to report all the PIDs. In the example given, it is actually the parent process that reports the PID of the child process once it is finished waiting for the child to execute. I had to take a different approach to fulfill the requirement by tracking the PIDs in a shared array initialized in the main process before forking. Each process would enter its own array id using getpid(), so the grandchild would have the context in its own execution. Overall, I believe the exercise was more about having a chain of processes wait on each other, and that goal was achieved. Parent waits for the child, which waits for the grandchild.

Code:

```cpp
// Nathan Bush
// CSE 4600 Spring 2022
// Homework 1, Part 4
//
// hw1_part1_4.cpp
// modify test_wait (original provided by Dr. Bilal Khan) so:
//      1. Child process creates another child and waits for it.
//      2. Grandchild prints out the IDs of itself, its parent, and grandparent.


#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace std;


int main()
{
    pid_t pid, child_pid, grandchild_pid;           //process_id
    const char *message;
    int n;
    int exit_code;
    int pid_array[3];


    cout << "fork program staring \n";
    pid_array[0] = getpid();
    pid = fork();
    switch (pid) {
        case -1:
            cout <<"Fork failure!\n";
            return 1;
        case 0:
            message = "This is the child\n";
            n = 5;
            exit_code = 9;
            pid_array[1] = getpid();
            child_pid = fork();
            switch(child_pid) {     //nested switch to handle second fork
                case -1:
                    cout <<"Fork failure!\n";
                    return 1;
                case 0:
                    pid_array[2] = getpid();
                    grandchild_pid = getpid();
                    message = "This is the grandchild\n";
                    n = 7;
                    exit_code = 7;
                default:
                    break;
```

```
            }
            break;
        default:
            message = "This is the parent\n";
            n = 3;
            exit_code = 0;
            break;
    }

    //console output identifying process
    for (int i = 0; i < n; ++i){
        cout << message;
        sleep(1);
    }

    //waiting for child to finish
    if (pid != 0) {             //parent
        int stat_val;
        pid_t child_pid;

        child_pid = wait (&stat_val);   //wait for child
        cout << "\nChild finished: PID = " << child_pid << endl;
        if (WIFEXITED(stat_val))
            cout << "child exited with code " << WEXITSTATUS (stat_val) << endl;
        else
            cout << "child terminated abnormally!" << endl;
    }

    //waiting for grandchild to finish
    else if (child_pid != 0) {  //child
        int stat_val2;
        pid_t grandchild_pid;

        grandchild_pid = wait (&stat_val2);     //wait for grandchild
        cout <<"\nGrandchild finished: PID = " << grandchild_pid << endl;
        if (WIFEXITED(stat_val2))
            cout << "grandchild exited with code " << WEXITSTATUS (stat_val2) << endl;
        else
            cout << "grandchild terminated abnormally!" << endl;
    }

    //grandchild, report PIDs
    else if (child_pid == 0) {
        cout << "\nGrandchild nearly finished, reporting PIDs:\n"
             << "My(grandchild) PID = " << pid_array[2] << endl
             << "My parent's PID = " << pid_array[1] << endl
             << "My grandparent's PID = " << pid_array[0] << endl
             << "... all done, grandchild finishing up now.\n";
    }

    exit (exit_code);
}
```

Run the executable by extracting it from the code and either double clicking part1_4.exe or running it from your terminal by navigating to its directory and typing ./part1_4.exe

Console output:

```
nate@NDB-LAP:~/Desktop/part 1$ ./part1_4.exe
fork program staring
This is the parent
This is the child
This is the grandchild
This is the parent
This is the child
This is the grandchild
This is the parent
This is the child
This is the grandchild
This is the child
This is the grandchild
This is the child
This is the grandchild
This is the grandchild
This is the grandchild

Grandchild nearly finished, reporting PIDs:
My(grandchild) PID = 19208
My parent's PID = 19207
My grandparent's PID = 19206
... all done, grandchild finishing up now.

Grandchild finished: PID = 19208
grandchild exited with code 7

Child finished: PID = 19207
child exited with code 9
```

5. Signals

Signals are software interrupts. We can handle signals using the **signal** library function:

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

This rather complex prototype means that **signal()** is a function that takes two parameters: *sig* and *func*. The function to be caught or ignored is specified by argument *sig* and *func* specifies the function that will receive the signal; this function must be one that takes a single **int** argument

and is of type **void**. The **signal()** function itself returns a function of the same type, which is the previous value of the function set up to handle this signal, or one of the two special values

SIG_IGN		Ignore the signal.
SIG_DFL		Restore default behavior.

```
//test_signal.cpp
#include <signal.h>
#include <unistd.h>
#include <iostream>

using namespace std;

void func (int sig)
{
  cout << "Oops! -- I got a signal " << sig << endl;
}

int main()
{
  (void) signal (SIGINT, func);   //catch terminal interrupts

  for (int i = 0; i < 20; ++i) {
    cout << "CSUSB CSE 4600 Operating Systems. Homework on signals" <<
endl;
    sleep (1);
  }
  return 0;
}
```

Requirement:

Now try the following scripts. Run the program and hit ^C for a few times. What do you see? Why?

Explanation:

As shown in the output below, the signal_test program printed a message 20 times on a 1 second delay. Each time I preempted the program by hitting ^C, the signal was caught by the function passed into the signal function, which printed a message to the console that an interrupt signal had been received. The program then continued on after handling the terminal interrupt as defined.

Example Output:

```
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$ ./test_signal
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
^COops! -- I got a signal 2
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
^COops! -- I got a signal 2
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
^COops! -- I got a signal 2
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
^COops! -- I got a signal 2
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
^COops! -- I got a signal 2
CSUSB CSE 4600 Operating Systems. Homework on signals
CSUSB CSE 4600 Operating Systems. Homework on signals
```

Sending Signals

A process can send a signal to itself by calling **raise**. It can also send a signal to another process (including itself) by calling **kill**:

```
#include<signal.h>
int raise (int sig);
```

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
```

Another useful signal function is **alarm**, which can be used to schedule a **SIGALARM** signal sometime in the future.

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

The **alarm** function schedules the delivery of a **SIGALARM** signal in *seconds* seconds. Use "man" to study more on **raise, kill, alarm**.

```
//test_alarm.cpp
#include <signal.h>
#include <unistd.h>
#include <iostream>

using namespace std;

//simulates an alarm clock
void alarm_off(int sig)
{
  cout << "Alarm has gone off " << endl;
}

//tell child process to wait for 5 seconds before sending
//a SIGALRM signal to its parent.

int main()
{
  int pid;

  cout << "Alarm testing!" << endl;

  if ((pid = fork()) == 0) {//child
    sleep (5);
    /*
      Get parent process id, send SIGALARM signal to it.
     */
    kill (getppid(), SIGALRM);
    return 1;
  }

  //parent process arranges to catch SIGALRM with a call
  //to signal and then waits for the inevitable.

  cout << "Waiting for alarm to go off!" << endl;
  (void) signal (SIGALRM, alarm_off);
```

```
  pause(); //process suspended, waiting for signals to wake up
  cout << "Done!" << endl;

  return 1;
}
```

Requirement:

Try the following "test_alarm.cpp" program. Run "test_alarm.cpp". What do you see? Why?

Explanation:

In this program, the parent forks the child and waits for a signal alarm. Upon receiving that signal, the parent process fires off the alarm_off function before exiting from the kill signal. The child sleeps for 5 seconds, then sends the kill signal to the parent and ends program execution. So, instead of waiting for the child to finish executing before ending the program, the parent waits for a kill signal and the child ends the program after sending that signal. The parent is actually waiting for an alarm signal, which when received will make the alarm_off function call. The child sends a kill signal, with an alarm signal. The parent receives and executes the alarm signal before finishing the kill command.

Example output:

```
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$ ./test_alarm
Alarm testing!
Waiting for alarm to go off!
Alarm has gone off
Done!
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$
```

More Robust Signals Interface

sigaction is function that provides more robust interface to signals. Use "man" to study this function:

Before calling **sigaction**(), you should first use **sigemptyset**() to empty the signal set. (Use '$ man sigemptyset' to find out the details.). Modify your *test_signal.cpp* program above by using **sigaction ()** to intercept **SIGINT**; replace the "for" loop with "while (1); you should be able to quit the program by entering "^\". (Need to intercept **SIGQUIT**.)

Report:

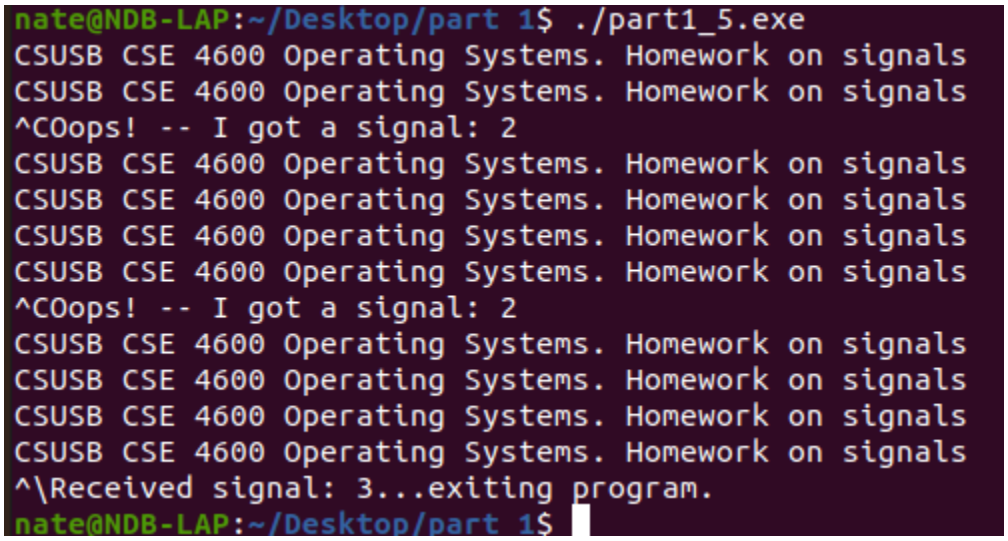
I added a function to handle the intercept SIGQUIT signal. Through documentation I found out about the sigaction structure initialization and defining how the handler reroutes the intercepted signal. I left the original functionality of the ^C intercept function but changed it to a sigaction intercept. I added a separate sigaction statement to handle the SIGQUIT signal so that ^\ would exit the program.

Code:

```cpp
// Nathan Bush
// CSE 4600 Spring 2022
// Homework 1, Part 5
//
// hw1_part1_5.cpp
// modify signal (original provided by Dr. Bilal Khan):
//    1. use sigaction() to intercept SIGINT
//    2. replace the "for" loop with while(1)
//    3. be able to quit the program by entering "^\". (Need to intercept SIGQUIT.)

#include <signal.h>
#include <unistd.h>
#include <iostream>

using namespace std;

void func (int sig)
{
  cout << "Oops! -- I got a signal: " << sig << endl;
}

void quit (int sig)
{
  cout << "Received signal: " << sig << "...exiting program." << endl;
  exit(1);
}

int main()
{
  struct sigaction handle_signal, handle_quit;

  handle_signal.sa_handler = func;
  handle_quit.sa_handler = quit;

  sigemptyset(&handle_signal.sa_mask);
  sigemptyset(&handle_quit.sa_mask);

  sigaction(SIGQUIT, &handle_quit, NULL);
  sigaction(SIGINT, &handle_signal, NULL);

  while(1) {
    cout << "CSUSB CSE 4600 Operating Systems. Homework on signals" << endl;
    sleep (1);
  }
  return 0;
}
```

Run the executable by extracting it from the code and either double clicking part1_5.exe or running it from your terminal by navigating to its directory and typing ./part1_5.exe

Console output:



Part 2 (3%)

1. Pipes

We have covered inter-process communication (IPC) via *message passing* and *shared memory* mechanism in the class. Another way of IPC is a pipe that is a pseudo file used for IPC, allowing data flow from one process to another. In UNIX, you can use pipes to link shell commands:

```
$ command1 | command2 | command2
```

The shell arranges the standard input and output of the commands, so that

1. Standard input to *command1* comes from the keyboard.
2. The standard output from *command1* is fed to *command2*.
3. The standard output from *command2* is fed to *command3*.
4. The standard output from *command3* is fed to the terminal screen.

2. Process Pipes

We can use **popen** and **pclose** to pass data between two processes:

```
#include <stdio.h>

FILE * popen(const char *command, const char *type);
int  pclose(FILE *stream);
```

A pipe is unidirectional, so the *type* argument may specify only reading or writing, not both; the resulting stream is correspondingly read-only or write-only.

Use "man" to study **popen()** and **pclose()**.

You can read data from a pipe which "connects" to a command. Try the following program:

```cpp
//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 ("ps -auxw", "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;
}
```

Requirement:

- What do you see when you execute "pipe1"? Why?

Example Output:

```
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$ ./pipe1
Output from pipe: USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 169276 13228 ?        Ss   13:01   0:04 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    13:01   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   13:01   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   13:01   0:00 [rcu_par_gp]
root           6  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/0:0H-events_highpri]
root           9  0.0  0.0      0     0 ?        I<   13:01   0:00 [mm_percpu_wq]
root          10  0.0  0.0      0     0 ?        S    13:01   0:00 [rcu_tasks_rude_]
root          11  0.0  0.0      0     0 ?        S    13:01   0:00 [rcu_tasks_trace]
root          12  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/0]
root          13  0.0  0.0      0     0 ?        I    13:01   0:11 [rcu_sched]
root          14  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/0]
root          15  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/0]
root          16  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/0]
root          17  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/1]
root          18  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/1]
root          19  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/1]
root          20  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/1]
root          22  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/1:0H-events_highpri]
root          23  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/2]
root          24  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/2]
root          25  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/2]
root          26  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/2]
root          28  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/2:0H-events_highpri]
root          29  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/3]
root          30  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/3]
root          31  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/3]
root          32  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/3]
root          34  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/3:0H-events_highpri]
root          35  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/4]
root          36  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/4]
root          37  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/4]
root          38  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/4]
root          40  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/4:0H-events_highpri]
root          41  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/5]
root          42  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/5]
root          43  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/5]
root          44  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/5]
root          46  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/5:0H-events_highpri]
root          47  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/6]
root          48  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/6]
root          49  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/6]
root          50  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/6]
root          52  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/6:0H-events_highpri]
root          53  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/7]
root          54  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/7]
root          55  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/7]
```

Explanation

The example code pipes the console output of the ps auxw command (listing current process details) to an initialized FILE object. Then, the length of the piped output is checked to make sure there is data in the file object. If so, the file object is read into the buffer. After passing a check to make sure that characters were actually read into the buffer, the buffer is outputted to the console. The result is an identical output than what would have been displayed on the console

by just running ps auxw, except for the header that was outputted after the pipe operation “Output from pipe: “

- Modify the program **pipe1.cpp** to **pipe1a.cpp** so that it accepts a command (e.g. "ls -l") from the keyboard. For example, when you execute "./pipe1a ps -auxw", it should give you the same output as **pipe1.cpp**.
  (Hint: Use string functions **strcpy**() and **strcat**() to store the commands in a buffer. Your **main** function should be like: *int main(int argc, char *argv[])*)

Report:

The trick with this exercise was to handle the argc and argv command line parameters passed to the program. argc indicates the number of commands passed, with the first one being the executable file. argv is an array of character strings holding each command. I looped through the argv array using the argc value in the loop initialization and concatenated each argv[i] value into a command string using strcat. Everything else in the program is the same, aside from replacing the hardcoded command in the popen call with the constructed string composed of argv parameters. The output is identical, as required.

Code:

```cpp
// Nathan Bush
// CSE 4600 Spring 2022
// Homework 1, Part 2_2
//
// pipe1a.cpp
// modify pipe1.cpp (original provided by Dr. Bilal Khan) so that it accepts a command:
//    1. (e.g. "ls -l") from the keyboard. For example, when you execute "./pipe1a ps -auxw"
//       it should give you the same output as pipe1.cpp.
//    2. Your main function should be like: int main(int argc, char *argv[])

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
  char command[50];

  for (int i = 1; i < argc; ++i) {
    strcat(command, argv[i]);
    strcat(command, " ");
  }

  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 (command, "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;
}
```

For this example run the executable from your terminal by navigating to its directory and typing ./pipe1a ps -auxw

Console Output:

```
nate@NDB-LAP:~/Desktop/part 2$ ./pipe1a ps auxw
Output from pipe: USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 169276 13228 ?        Ss   13:01   0:04 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    13:01   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   13:01   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   13:01   0:00 [rcu_par_gp]
root           6  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/0:0H-events_highpri]
root           9  0.0  0.0      0     0 ?        I<   13:01   0:00 [mm_percpu_wq]
root          10  0.0  0.0      0     0 ?        S    13:01   0:00 [rcu_tasks_rude_]
root          11  0.0  0.0      0     0 ?        S    13:01   0:00 [rcu_tasks_trace]
root          12  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/0]
root          13  0.0  0.0      0     0 ?        I    13:01   0:13 [rcu_sched]
root          14  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/0]
root          15  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/0]
root          16  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/0]
root          17  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/1]
root          18  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/1]
root          19  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/1]
root          20  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/1]
root          22  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/1:0H-events_highpri]
root          23  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/2]
root          24  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/2]
root          25  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/2]
root          26  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/2]
root          28  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/2:0H-events_highpri]
root          29  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/3]
root          30  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/3]
root          31  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/3]
root          32  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/3]
root          34  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/3:0H-events_highpri]
root          35  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/4]
root          36  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/4]
root          37  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/4]
root          38  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/4]
root          40  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/4:0H-events_highpri]
root          41  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/5]
root          42  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/5]
root          43  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/5]
root          44  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/5]
root          46  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/5:0H-events_highpri]
root          47  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/6]
root          48  0.0  0.0      0     0 ?        S    13:01   0:00 [idle_inject/6]
root          49  0.0  0.0      0     0 ?        S    13:01   0:00 [migration/6]
root          50  0.0  0.0      0     0 ?        S    13:01   0:00 [ksoftirqd/6]
root          52  0.0  0.0      0     0 ?        I<   13:01   0:00 [kworker/6:0H-events_highpri]
root          53  0.0  0.0      0     0 ?        S    13:01   0:00 [cpuhp/7]
```

3. The pipe Call

The lower-level **pipe()** function provides a means of passing data between two processes, without the overhead of invoking a shell to interpret the requested command.

```
#include <unistd.h>

int pipe (int
fd[2]);
```

Use "man" to study more of **pipe()**. The function **pipe()** takes an array address of two integer file descriptors; it fills the array with two new file descriptors and returns 0 if successful. The two file descriptors returned are connected in a special way. Any data written to *fd[1]* can be read back from *fd[0]*. The data are processed on a first in, first out (FIFO) basis. Try the following program that illustrates this concept.

```
//pipe3.cpp
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <iostream>

using namespace std;

int main()
{
  int nbytes;
  int fd[2];           //file descriptors for pipe
  const char s[] = "CSUSB";
  char buffer[BUFSIZ+1];

  memset (buffer, 0, sizeof(buffer));   //clear buffer

  if (pipe(fd) == 0) { //create a pipe
    nbytes = write(fd[1], s, strlen(s));      //send data to pipe
    cout << "Sent " << nbytes << " bytes to pipe." << endl;
    nbytes = read (fd[0], buffer, BUFSIZ);    //read data from pipe
    cout << "Read " << nbytes << " from pipe: " << buffer << endl;
    return 0;
  }
  return 1;
}
```

Requirement:

What do you see when you execute "pipe3" ? Why?

Example Output:

```
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$ ./pipe3
Sent 5 bytes to pipe.
Read 5 from pipe: CSUSB
```

Explanation:

This program establishes a pipe between two file descriptors. It then reads in a hardcoded character array into one of those file descriptors. Next, the contents of the other file descriptor are read into the buffer. Finally, the buffer is outputted to the console. Along the way, the return value for read and write (the number of bytes) is outputted to show that no bytes were lost. This is an example of a pipeline being established between two file objects where reading/writing into one changes the other as well. The character array is ‘piped’ from one file object to the other and into a buffer for output.

4. Parent and Child Processes

We can use **exec()** to create a child process running a different program. After an **exec** call, the old process has been replaced by the new child process. The following two programs demonstrate the concept. The first is the "data producer", which creates the pipe and then invokes the child, the "data consumer".

```c
//pipe4.cpp  (data producer)
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {   //creates pipe
        fork_result = fork();
        if (fork_result == (pid_t)-1) {  //fork fails
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {    //child
            sprintf(buffer, "%d", file_pipes[0]);
            (void)execl("pipe5", "pipe5", buffer, (char *)0);
            exit(EXIT_FAILURE);
        }
        else {                     //parent
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

```
// The 'consumer' program, pipe5.cpp, that reads the data is much
simpler.

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

Compile and execute:

```
$ g++ -o pipe4 pipe4.cpp
$ g++ -o pipe5 pipe5.cpp
$ ./pipe4
```

You should see output similar to the following.

```
6403 - wrote 3 bytes
6404 - read 3 bytes:
123
```

Confirmed: (although I added a 1 second delay on the producer so it wouldn't return control to the command prompt prior to the consumer outputting to console).

```
nate@NDB-LAP:~/Desktop/CSE_4600_HW1$ ./pipe4
35491 - wrote 3 bytes
35492 - read 3 bytes: 123
```

Requirement:

Modify pipe4.cpp so that it accepts a message from the keyboard and sends it to **pipe5**.

Report:

Modifications made: changed character array so that it is no longer constant and initialized to a length of 64 bytes maximum. Added console output/input to retrieve user-entered character array. All other code is identical to what was provided to us. When the pipe4a.exe is run, the user is asked to enter a message to be piped. Upon entering the message and hitting enter, the producer forks a consumer with execl. The consumer is told to run the program compiled in pipe5. The producer pipes the message into its file object, the consumer retrieves that data from its file object on the other side of the pipeline and outputs the message to console.

Code:

```cpp
// Nathan Bush
// CSE 4600 Spring 2022
// Homework 1, Part 2_4
//
// pipe4a.cpp
// Modify pipe4.cpp so that it accepts a message from the keyboard and sends it to pipe5.cpp
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <iostream>

int main()
{
    int data_processed;
    int file_pipes[2];
    char some_data[64];
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    std::cout << "Please enter a message to be piped to the consumer (max 64 chars): ";
    std::cin.getline(some_data, 64);

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {   //creates pipe
        fork_result = fork();
        if (fork_result == (pid_t)-1) {  //fork fails
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {    //child
            sprintf(buffer, "%d", file_pipes[0]);
            (void)execl("pipe5", "pipe5", buffer, (char *)0);
            exit(EXIT_FAILURE);
        }
        else {                     //parent
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    sleep(1);
    exit(EXIT_SUCCESS);
}
```

Run the executable by extracting it from the code and either double clicking pipe4a.exe or running it from your terminal by navigating to its directory and typing ./pipe4a.exe

Console output:

```
nate@NDB-LAP:~/Desktop/part 2$ ./pipe4a.exe
Please enter a message to be piped to the consumer (max 64 chars): CSE 4600 is fun!
38262 - wrote 16 bytes
38263 - read 16 bytes: CSE 4600 is fun!
nate@NDB-LAP:~/Desktop/part 2$
```

Part 3 (3%)

1. Pthreads

IEEE defines a POSIX standard API, referred to as **Pthreads** (IEEE 1003.1c), for thread creation and synchronization. Many contemporary systems implement Pthreads, including Linux, Solaris, and Mac OS X. To use **Pthreads** in your program, you must include **pthread.h** and link with **-l pthread**. The following code shows how to use **Pthreads**.

```
/*
  pthreads_demo.cpp
  A very simple example demonstrating the usage of pthreads.
  Compile: g++ -o pthreads_demo pthreads_demo.cpp -lpthread
  Execute: ./pthreads_demo
*/

#include <pthread.h>
#include <stdio.h>

using namespace std;

//The thread
void * thread_func (void *data)
{
  char *tname = (char *) data;

  printf("My thread identifier is %s\n", tname);

  pthread_exit (0);
}

int main ()
{
  pthread_t id1, id2;           //thread identifiers
  pthread_attr_t attr1, attr2;  //set of thread attributes
  char *tnames[2] = { "Thread 1", "Thread 2" }; //names of threads
  //get the default attributes
  pthread_attr_init (&attr1);
  pthread_attr_init (&attr2);

  //create the threads
  pthread_create (&id1, &attr1, thread_func, tnames[0]);
  pthread_create (&id2, &attr2, thread_func, tnames[1]);
```

```
  //wait for the threads to exit
  pthread_join (id1, NULL);
  pthread_join (id2, NULL);

  return 0;
}
```

In the above example, **pthread_tid** is used to declare the identifiers for the threads being created. Each thread has a set of attributes containing information about the thread such as stack size and scheduling information. We use **pthread_attr_t** to declare the attributes of the threads and set the attributes in the function call by **pthread_attr_init()**. You may notice that no thread attributes were explicitly in this example. In that case the default attributes will be used. The function **pthread_create()** is used to create a separate thread. In addition to passing the thread identifier and the attributes to the thread, the name of the function, **thread_func**, is also passed where the new thread will begin execution. The last argument passed to **pthread_create()** is the argument passed to the thread which in this case is a string parameter containing the predefined name of the thread. At this point, the program has three threads: the initial parent thread in **main()** and two children threads in **runner()**. After creating children threads, the **main()** thread will wait for the **thread_func()** threads to complete by calling **pthread_join()** function.

Pthreads specification has a rich set of functions, allowing users to develop very sophisticated multithreaded programs. In your homework, you do not need to use many of the Pthread functions. Besides POSIX threads, different crucial thread programming schemes exist in the market. MS Windows has its own threading interface which is very different from POSIX threads. Though Sun's Solaris supports Pthreads, it also has its own thread API. Other UNIX systems may also have their own thread APIs.

Requirement:

Try **pthreads.cpp**. Modify it so that they run 3 threads (instead of two) and each thread runs a different function, displaying a different message. Copy-and-paste the source code and the outputs in your report.

Output from unmodified pthreads_demo:

```
nate@NDB-LAP:~/Desktop/part 3$ ./pthreads_demo
My thread identifier is Thread 1
My thread identifier is Thread 2
```

Report:

I had trouble compiling the demo as written. I had to change the pointer to char array for storing thread identifiers into just a char array, as well as initialize it with the length of the array and the byte length for memory allocation. Such as:

char tnames[3][10] = { "Thread 1", "Thread 2", "Thread 3" };

After that, code would compile just fine. So, aside from adding a third thread identifier, I created 3 separate functions that were each called by their respective threads upon creation. From there it was just a matter of making sure I had three thread attribute statements and three join statements. The code and console output are displayed below.

Code:

```
// Nathan Bush
// CSE 4600 Spring 2022
// Homework 1, Part 3_1
//
// pthreads_mod.cpp
// modify pthreads_demo.cpp (original provided by Dr. Bilal Khan) so:
//    1. 3 threads are run instead of 2
//    2. Each thread runs its own function, displaying a different message

#include <pthread.h>
#include <stdio.h>

using namespace std;

// first thread function
void * thread_func1 (void *data)
{
    char *tname = (char *) data;

    printf("First thread, reporting in: my thread identifier is %s.\n", tname);

    pthread_exit (0);
}

// second thread function
void * thread_func2 (void *data)
{
    char *tname = (char *) data;

    printf("Second thread, reporting in: my thread identifier is %s.\n", tname);

    pthread_exit (0);
}

// third thread function
void * thread_func3 (void *data)
{
    char *tname = (char *) data;

    printf("Third thread, reporting in: my thread identifier is %s.\n", tname);

    pthread_exit (0);
}
```

```
45  int main ()
46  {
47      pthread_t id1, id2, id3;               //thread identifiers
48      pthread_attr_t attr1, attr2, attr3; //set of thread attributes
49      char tnames[3][10] = { "Thread 1", "Thread 2", "Thread 3" };    //names of threads
50      //get the default attributes
51      pthread_attr_init (&attr1);
52      pthread_attr_init (&attr2);
53      pthread_attr_init (&attr3);
54
55      //create the threads
56      pthread_create (&id1, &attr1, thread_func1, tnames[0]);
57      pthread_create (&id2, &attr2, thread_func2, tnames[1]);
58      pthread_create (&id3, &attr3, thread_func3, tnames[2]);
59
60      //wait for the threads to exit
61      pthread_join (id1, NULL);
62      pthread_join (id2, NULL);
63      pthread_join (id3, NULL);
64
65      return 0;
66  }
```

Compile: g++ -o pthreads_mod.exe pthreads_mod.cpp -lpthread
Run: ./pthreads_mod.exe

Console output:

```
nate@NDB-LAP:~/Desktop/part 3$ g++ -o pthreads_mod.exe pthreads_mod.cpp -lpthread
nate@NDB-LAP:~/Desktop/part 3$ ./pthreads_mod.exe
First thread, reporting in: my thread identifier is Thread 1.
Second thread, reporting in: my thread identifier is Thread 2.
Third thread, reporting in: my thread identifier is Thread 3.
nate@NDB-LAP:~/Desktop/part 3$
```

2. Synchronization using Pthreads mutex

The program below is used by two threads to modify/update a shared resource (a long integer) at the same time. There is no synchronization performed in the code below. You may see unexpected/unmatching results when you execute the program multiple times. The shared resource is a long integer where first thread takes its value as 1 and increments it to a number 300 million, where the second thread decrements one from the incremented resource to all the way to 0 (but adding a -1 in the for loop). You will implement mutual exclusion in the code below to make sure that the increments and decrements of the shared resource are done without any interruption by the other thread.

```c
/*
Compile: gcc -o shared_resouce_mutex shared_resource_mutex.c -lpthread
Execute: ./shared_resource_mutex
*/

#include <stdio.h>
#include <pthread.h>

#define iterations 300000000
long long shared_resource = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// Thread function to modify shared resource
void* inc_dec_resource(void* arg){
    //get the pointer from main thread and dereference it to put the value in resource_value
    int resource_value = *(int *) arg;
    for(int i=0; i < iterations; i++){
        shared_resource += resource_value;
    }
    pthread_exit(NULL);
}
```

```c
int main(void){

    // Thread 1 to increment shared resource
    pthread_t tid1, tid2;
    int value1 = 1;
    pthread_create(&tid1, NULL, inc_dec_resource, &value1);
    // Thread 2 to increment shared resource
    int value2 = -1;
    pthread_create(&tid2, NULL, inc_dec_resource, &value2);
    pthread_join(tid1, NULL);
```

```
    pthread_join(tid2, NULL);
    printf("Shared resource value: %lld\n", shared_resource);

    return 0;
}
```

Requirement:

- Compile and run the above program. Execute it 5 times in the command prompt. What do you see in the result?

Example output:

```
nate@NDB-LAP:~/Desktop/part 3$ gcc -o shared_resource_mutex shared_resource_mutex.c -lpthread
nate@NDB-LAP:~/Desktop/part 3$ ./shared_resource_mutex
Shared resource value: -28312801
nate@NDB-LAP:~/Desktop/part 3$ ./shared_resource_mutex
Shared resource value: -61299915
nate@NDB-LAP:~/Desktop/part 3$ ./shared_resource_mutex
Shared resource value: 217400381
nate@NDB-LAP:~/Desktop/part 3$ ./shared_resource_mutex
Shared resource value: -169882878
nate@NDB-LAP:~/Desktop/part 3$ ./shared_resource_mutex
Shared resource value: -13936541
nate@NDB-LAP:~/Desktop/part 3$
```

Explanation:

The shared resource value reported appears to be random. This is due to the race condition on the shared resource. Both threads are trying to increment and decrement the shared resource simultaneously. Depending on which thread wins the race condition in each iteration, that thread will perform its operation. Once one of the threads exhausts its operations, the other will dump the rest of its increments/decrements without racing. The resulting value in the shared resource will always be random. Without thread synchronization.

- Try executing the command with $ time ./shared_resource_mutex. What do you see other than printf statement? Copy and paste and report how long did the threads take to run.

  Example output:

```
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: 200666473

real    0m1.014s
user    0m1.848s
sys     0m0.001s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: -38217353

real    0m1.648s
user    0m3.282s
sys     0m0.001s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: -108449291

real    0m1.305s
user    0m2.585s
sys     0m0.005s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: -128054482

real    0m1.345s
user    0m2.667s
sys     0m0.005s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: -37295338

real    0m1.620s
user    0m3.221s
sys     0m0.001s
nate@NDB-LAP:~/Desktop/part 3$
```

Explanation:

The time to execute this program also varies in a random fashion, much like the shared resource value. This depends on the number of times the threads race and interrupt each other before one of them exhausts its iterations and the other can proceed with dumping the rest of its operations. The run time will also be random like this until thread synchronization is implemented.

- Try switching the joins
  pthread_join(tid2, NULL);

  pthread_join(tid1, NULL);

  Do you see any difference? Please report if and why you do or do not see any difference in terms of the randomness in the result of the shared resource.

Example output:

```
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: 102322916

real    0m1.653s
user    0m3.287s
sys     0m0.000s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: -58761950

real    0m1.465s
user    0m2.919s
sys     0m0.005s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: -98470755

real    0m1.542s
user    0m3.071s
sys     0m0.001s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: 8970700

real    0m1.706s
user    0m3.408s
sys     0m0.001s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex
Shared resource value: 49435412

real    0m1.564s
user    0m3.067s
sys     0m0.005s
nate@NDB-LAP:~/Desktop/part 3$
```

Explanation:

I did not see any difference in the randomness of the shared resource value or the run time, both still exhibited the same level of randomness after the pthread join statements were switched. This makes sense, because the order shouldn't matter in this case. The main function has to wait on both threads to finish, and the fact that they finish at random times is also a function of the race condition. Whether thread2 finishes first and is waiting to join until after thread1 finishes, or whether thread1 finishes first, joins, and then the program is waiting on thread2 to finish is irrelevant to the level of randomness exhibited.

- In the inc_dec_resource() function, implement mutual exclusion (pthread_mutex_lock) to ensure that the result becomes 0 every time when you execute your program. Put your updated code in the report (highlighted) and show your screenshot of the execution by running the script three time using $ time ./shared_resource_mutex. Hint: Your loop is incrementing/decrementing the resource which should be protected by each thread while it is executing that portion.

Report:

I added a mutex lock and mutex unlock statement around the for loop in the incrementing/decrementing function. As a result, whichever thread reaches the mutex lock first will secure the shared resource, perform its iterations, then unlock. The other thread will then be free to acquire the lock, perform its iterations, unlock and join. The only difference I made in the code was simplifying the mutex declaration in the global namespace, and wrapping the for loop in mutex lock and mutex unlock statements. In the console output screenshot shown below, the resulting shared resource value was 0 for all 5 test runs, and the run time was also very stable and much faster than the code with race conditions unchecked.

Code:

```c
// Nathan Bush
// CSE 4600 Spring 2022
// Homework 1, Part 4_2
//
// shared_resource_mutex_mod.cpp
// Modify share_resource_mutex.cpp so that it implements a mutex lock on the shared resource
// Resulting shared resource should always be 0 after running

#include <stdio.h>
#include <pthread.h>

#define iterations 300000000
long long shared_resource = 0;

pthread_mutex_t mutex;

// Thread function to modify shared resource
void* inc_dec_resource(void* arg){
    //get the pointer from main thread and derefernce it to put the value in resource_value
    int resource_value = *(int *)arg;
    pthread_mutex_lock(&mutex);
    for(int i = 0; i < iterations; i++){
        shared_resource += resource_value;
    }
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

int main(void){

    //Thread 1 to increment shared resource
    pthread_t tid1, tid2;
    int value1 = 1;
    pthread_create(&tid1, NULL, inc_dec_resource, &value1);

    //Thread 2 to decrement shared resource
    int value2 = -1;
    pthread_create(&tid2, NULL, inc_dec_resource, &value2);
    pthread_join(tid2, NULL);
    pthread_join(tid1, NULL);
    printf("Shared resource value: %lld\n", shared_resource);

    return 0;
}
```

Console Output:

```
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex_mod.exe
Shared resource value: 0

real    0m0.757s
user    0m0.757s
sys     0m0.000s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex_mod.exe
Shared resource value: 0

real    0m0.751s
user    0m0.746s
sys     0m0.004s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex_mod.exe
Shared resource value: 0

real    0m0.746s
user    0m0.746s
sys     0m0.000s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex_mod.exe
Shared resource value: 0

real    0m0.748s
user    0m0.748s
sys     0m0.000s
nate@NDB-LAP:~/Desktop/part 3$ time ./shared_resource_mutex_mod.exe
Shared resource value: 0

real    0m0.749s
user    0m0.748s
sys     0m0.001s
nate@NDB-LAP:~/Desktop/part 3$
```