Ravi’s Copy of Homework 4

Problem 1:

Question 1:

I will now provide two examples where multithreading provides better performance in a program than single threading and vice versa:

Example 1 where MultiThreading provides better performance in a program than single threading: The first example where MultiThreading provides better performance than single threading is when you are trying to calculate the sum of all the numbers in an Array Data Structure. In a single threaded program, you would simply write a for loop to loop through the entire array. As you were looping through the array, you would have a variable keep track of the sum of the elements that have been analyzed so far. Now, the single threaded approach doesn’t seem too bad at face value. But, when the array size is huge(i.e. Over 10 million), the single threaded approach can certainly incur a significantly large runtime. On the other hand, when we use a multi threaded approach, we can partition the input array into several components(e.g. 100 components). If we choose to partition the array into 100 components, we can create 100 threads and assign each component to one of the 100 threads. Then, each of the 100 threads will calculate the sum of the number in its assigned component of the original array. Once each of the 100 threads has finished computing the sum of the numbers in its assigned component of the original array, it will write the value to a shared variable(i.e. Preferably an array that is shared among all threads). Once all the threads are done executing, the parent just needs to loop through the shared variable(i.e. An array of size 100 storing the results from each thread) and then sum up all the values. The reason the Multithreaded approach could potential incur a smaller runtime is that this program is mostly CPU-bound(i.e. Doesn’t really need that many I/O operations and the threads won’t be blocking too much). Hence, in an architecture where there is simultaneous multithreading, some of the 100 threads will execute at the same time thus resulting in faster computation time

Example 2 where MultiThreading provides better performance in a program than single threading:

Another example of where MultiThreading could be used in a Word Processor Program like Google Documents or Microsoft Word. For example, a user can be writing text to the document and a background task can be executed to run a spell check program to detect errors in a user’s grammar/spelling. So, as we are typing into the document, we will be able to check spelling mistakes as we go rather than having to run a separate program at the end. If the word Processor program was implemented using a Single Threaded program, this would mean that we would have to type all of our text into the document and then run a spell check program. It would be quite cumbersome because we can certainly be typing quite a lengthy document. Hence, when we run the spell checker program, there is certainly a good chance that there are a larger number of mistakes.

Example 1 where Single Threading is a better approach than Multi Threading:
An example of where Single Threading is a better approach than Multi Threading is for I/O Bound Operations. Essentially, when our program is performing quite a lot of I/O Operations, having multiple threads may result in "blocking times" for some of the threads. So, when one thread is reading data from a file, another thread will have to block in order to wait for the first thread to finish reading data from that file. The cumulative "blocking time" across all the threads will result in a significant overhead for this program. To avoid this, it would be far more efficient to just have a single thread reading from that file

Example 2 where Single Threading is a Better Approach than Multi Threading
A case where single threading is obviously a better approach is when there are dependencies between different sections of our program. So, we can consider a program where we are processing an array and we are attempting to locate only one instance of a target number in the array. Since this target number can occur many times in the array, partitioning the array into multiple components and making this program multithreaded will result in the program locating multiple instances of the target number which is NOT what we want. We only want to locate 1 instance of the target number. Hence, the solution would be to clearly use a single threaded program and just loop through the array

Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

It would be better to use a multithreaded solution using multiple kernel threads when our program is CPU-Bound. What I mean by CPU-Bound is a program that relies more on computations rather than I/O Operations. The reason why multithreading is far more suited for an environment where we do CPU bound operations is that for CPU operations our program doesn't block that frequently. Hence, we won't have to worry about incurring significant overhead due to "blocking time". This will inevitably help us achieve much faster execution times in a multithreading environment

When the program that we have is non-parallelizable, it is obviously not practical to use a multithreaded solution since we must execute the program sequentially. Furthermore, when the program we have is I/O Bound(i.e. Requires a large amount of Input/Output Operations), it wouldn't be too wise to use a multithreaded solution. The reason is that threads can certainly block on an I/O Operation and this "blocking time" may propagate across all the thread and result in an increased latency time. We obviously don't want this I/O Latency time to impose too much of an overhead and incur a huge cost in runtime. Hence, it would just be better to use a single threaded program to execute a program that is I/O Bound

Question 2

1. In this problem, we are given the following information: (1) the number of user threads is greater than the number of processing cores, (2) we are using a many-to-many model to map user threads to kernel threads

   In this case, we are given information that the number of kernel threads being used is less than the number of processing cores. In terms of performance, we are not attaining our optimal level of performance. The reason for this is that the number of kernel threads is less than the number of processing cores. Hence, even though we are able to execute all of our kernel threads such that each kernel thread is executed on a separate processing core, there will remain idle processing cores. Hence, in this case, since we are not using all the processing cores and we are only using a subset of all the processing cores, the process will incur a larger runtime. Hence, it is clear that this configuration is not an ideal configuration for us to use.
2. In this problem, we are given the following information: (1) the number of user threads is greater than the number of processing cores, (2) we are using a many-to-many model to map user threads to kernel threads

   In this case, we are given information that the number of kernel threads being used is equal to the number of processing cores. In this case, the most likely thing that the Operating System will do is assign each kernel thread to each processing core. Since all processing cores are being used, our process will have a quick run time. However, we must analyze the case when one user thread blocks for an I/O Operation(e.g. Read from file). When this user thread blocks for an I/O, its corresponding kernel thread will block as well. We know that the number of kernel threads is equal to the number of processing cores and each kernel thread, most likely, is assigned to one processing core by the Operating System. Let's consider the case when a user thread, let's call it U, blocks. We know that the kernel thread, let's call it T, corresponding to this user thread will block as well. Let's denote the processing core of kernel thread T to be P. Since the number of kernel threads is equal to the number of processing cores and each kernel thread, most likely, is assigned to one processing core by the Operating System, there isn't another kernel thread that can be context switched onto processing core P when kernel thread T blocks. Hence, while kernel thread T blocks, processing core P will not be able to be used by any other kernel thread. This will result in an inefficient processor core utilization for our process because, in the event of a kernel thread blocking, a processing core will not be able to be used by any other process.

3.

In this problem, we are given the following information: (1) the number of user threads is greater than the number of processing cores, (2) we are using a many-to-many model to map user threads to kernel threads

In this case, we are given information that the number of kernel threads being used is greater than the number of processing cores. Since we are able to use all processing cores here, we will notice that our program has a faster execution time. Now, let's analyze the case where we have a blocking thread. When this user thread blocks for an I/O, its corresponding kernel thread will block as well. Let's consider the case when a user thread, let's call it U, blocks. We know that the kernel thread, let's call it T, corresponding to this user thread will block as well. Let's denote the processing core of kernel thread T to be P. However, since we have more kernel threads than processors, we can simply context switch out kernel thread T from Processor P and still continue to use processor P! This will enable us to have an efficient processor core utilization as well as having a faster execution time.

Problem 2C:

Problem 2C part 1
(i.e. Where parent thread waits for child thread to finish generating Fibonacci Sequence):

Github: https://github.com/Ravi-Raghavan/CompSysHW4/blob/main/2C/Main.c

Analysis:

I will start my explanation of the program with the main function

main() Function:

The first line in the main function is printing a line to the terminal asking the user to enter a number indicating the number of fibonacci numbers to generate in the sequence
The second line in the main function takes the number from the terminal that the user enters and then stores it in a variable. Let SIZE represent the number of Fibonacci numbers that the user wants to generate
In the third line in the main function, the program allocates a memory block whose size is equal to "SIZE". This memory block will serve as the array where the Fibonacci numbers will be stored
The fourth line creates a pthread_t data structure called threadID to represent the unique thread ID of the thread that I will soon generate
The fifth line creates a pthread_attr_t data structure called threadAttribute to store the attributes of the thread that I will soon generate
The sixth line initializes the pthread_attr_t data structure called threadAttribute to basically represent "default attributes" for the thread that I will soon create
The 7th line creates a new thread with the thread ID specified by the data structure threadID and tells the thread to execute the FibonacciIterative function with the argument SIZE
The 8th line in the main function calls pthread_join on the thread with ID specified by the threadID data structure. Hence, the main thread will wait until the child thread is finished executing the FibonacciIterative Function.

FibonacciIterative function analysis for thread:

The thread will execute the FibonacciIterative function. The input parameter for the FibonacciIterative function(i.e. ptr) is a pointer to a number that represents the total number of Fibonacci numbers that the user wants to generate. The next step is to do a simple for loop. If index == 0 or index == 1, these are base cases. The Fibonacci Number must be f0 = 0 when index == 0 and the fibonacci number must be f1 = 1 when index == 1. When index > 1, we follow the simple formula that fn = f_(n - 1) + f_(n - 2).

Once the child thread has finished executing the Fibonacci Iterative function it will terminate.
Once the child thread has terminated, the main thread will call displayFibonacci() and will display the Fibonacci numbers in the array FIBONACCI

Problem 2C Part 2(i.e. Where parent thread has to wait on child thread):
Github:
https://github.com/Ravi-Raghavan/CompSysHW4/blob/main/2C/MutexLocksWithoutCondition

What changes were made from Part 1?

In this program, according to the problem description, the program had to be changed to enable the parent process to access the Fibonacci values as soon as they have been computed rather than waiting for the child thread to finish computing ALL of the Fibonacci numbers. Hence, the changes I made to the original program was to incorporate mutex locks. I considered the FIBONACCI array as a “shared resource” between the main thread(i.e. The parent thread) and the child thread. I had a count variable in my program. Whenever the count variable was even, I gave the child thread the mutex lock to be able to write the value to the FIBONACCI array. Once the child thread wrote the value to the FIBONACCI array, it “unlocked” the mutex lock, thus releasing it, and also incremented the value of count. When the count variable was odd, the parent thread was given the mutex lock to read data from the FIBONACCI array. Once the parent thread was finished reading data from the FIBONACCI array, it “unlocked” the mutex lock thus releasing it and also incremented the value of count.

This ensures that the child thread would write a value to the FIBONACCI array, the parent would read this value, then the child thread would write another value to the FIBONACCI array, then the parent would read this value, and so on and so forth until all values are written to the FIBONACCI array by the child thread and until all values are read from the FIBONACCI array by the parent thread.

As shown above, this is achieved by giving the child thread the mutex lock, having that thread perform its operations(i.e. The critical section of this program), release the mutex lock, the parent thread will then acquire the mutex lock, the parent thread will then perform its operations(i.e. The critical section of this program), release the mutex lock, and the child thread will then reclaim the mutex lock. This sequence will continue until all values are written to the FIBONACCI array by the child thread and until all values are read from the FIBONACCI array by the parent thread.

I have used the count variable to control the order as shown above. The reason I chose to do this is because in class, we learned that we don’t have control over the order in which threads are context switched onto the processor. So, if I just used mutex locks, this may present a weird scenario. There is a possibility that the child thread remains on the processor for a period of time, retains control over the mutex locks during this period of time, and subsequently writes quite a few values to the FIBONACCI Array. Since the programmer has no direct control over the Operating System scheduling algorithms, we have no way of controlling how long each thread remains on the processor before it is context switched out. In this problem, the requirement states that the parent thread MUST read the values from the FIBONACCI Array immediately after the child thread has written it to the array. Hence, I used the count variable to control the sequence of operations(i.e. giving the child thread the mutex lock, having that thread perform its operations(i.e. The critical section of this program), release the mutex lock, the parent thread will then acquire the mutex lock, the parent thread will then perform its operations(i.e. The critical section of this program), release the mutex lock, and the child thread will then re-claim the mutex lock)

Problem 2C Part 3(i.e. Where Parent Thread Has to Wait on Child Thread):
Github:
https://github.com/Ravi-Raghavan/CompSysHW4/blob/main/2C/MutexLocksWithCondition

What changes were made from Part 1?

In addition to the previous program, I also wrote an alternative program that used condition variables in conjunction with mutex locks.

In this program, according to the problem description, the program had to be changed to enable the parent process to access the Fibonacci values as soon as they have been computed rather than waiting for the child thread to finish computing ALL of the Fibonacci numbers.
The logic of the actual Fibonacci Method was the exact same and the Logic of the Parent Thread Displaying the Fibonacci Numbers were the exact same.

I used Mutex Locks in conjunction with Condition Variables. I had two condition variables in my program(i.e. readDone and writeDone). Essentially, the value of readDone would be 0 when the parent thread is still reading a value from the FIBONACCI array. The value of readDone would be 1 when the parent thread is finished reading a value from the FIBONACCI Array. The value of writeDone would be 0 when the child thread is in the process of writing a data value to the FIBONACCI array. The value of writeDone would be 1 when the child thread is finished writing a value to the FIBONACCI array. Essentially, I had my child thread call pthread_cond_wait and put that in a while loop to ensure that the child thread would wait until the parent thread is done reading from the FIBONACCI array. The reason I used a while loop is because the Linux Manual States that "Spurious wakeups from the pthread_cond_timedwait() or pthread_cond_wait() functions may occur. Since the return from pthread_cond_timedwait() or pthread_cond_wait() does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return". Hence, I didn't want the pthread_cond_time to return incorrectly so I just put it in a while loop and the condition of the while loop was readDone == 0. So, the child thread will only return from the while loop only when the parent thread is done reading data from the FIBONACCI Array

Likewise, in the parent thread, I had a while loop containing the statement pthread_cond_wait and the condition of the while loop was writeDone == 0. This ensures that the we will return out of the while loop only when the child thread has written data to the FIBONACCI array.

Another detail to note is that whenever the child thread writes data to the array, it uses the pthread_cond_signal() system call to unblock the parent thread that is blocked on the condition variable. In my program, I just used a uniform condition variable "c".

Likewise, whenever the parent thread writes data to the array, it uses the pthread_cond_signal() system call to unblock the child thread that is blocked on the condition variable. In my program, I just used a uniform condition variable "c".

Whenever the child thread is writing data to the FIBONACCI Array, I used mutex locks to ensure that the child thread is the only thread in my process that has access to the FIBONACCI Array.

Likewise, whenever the parent thread is reading data from the FIBONACCI Array, I used mutex locks to ensure that the parent thread is the only thread in my process that has access to the FIBONACCI Array when it is reading data from the array. To implement this, in the displayFibonacci() method(i.e. The function that the parent thread is calling when iterating through the FIBONACCI array to print out all the

Fibonacci numbers), the first instruction of each loop iteration is pthread_mutex_lock(). What this does is that if the mutex lock is available then it enables the parent thread to claim it and to begin doing its read operations on the array without worrying about having the child process doing simultaneous operations on the array at the same time. If the mutex lock is not available, then this indicates that the child thread possesses the mutex lock and is doing some operations on the array. Then, in this case, the parent thread will block(i.e. wait) for the child thread to release this lock so that it can perform read operations on the array.

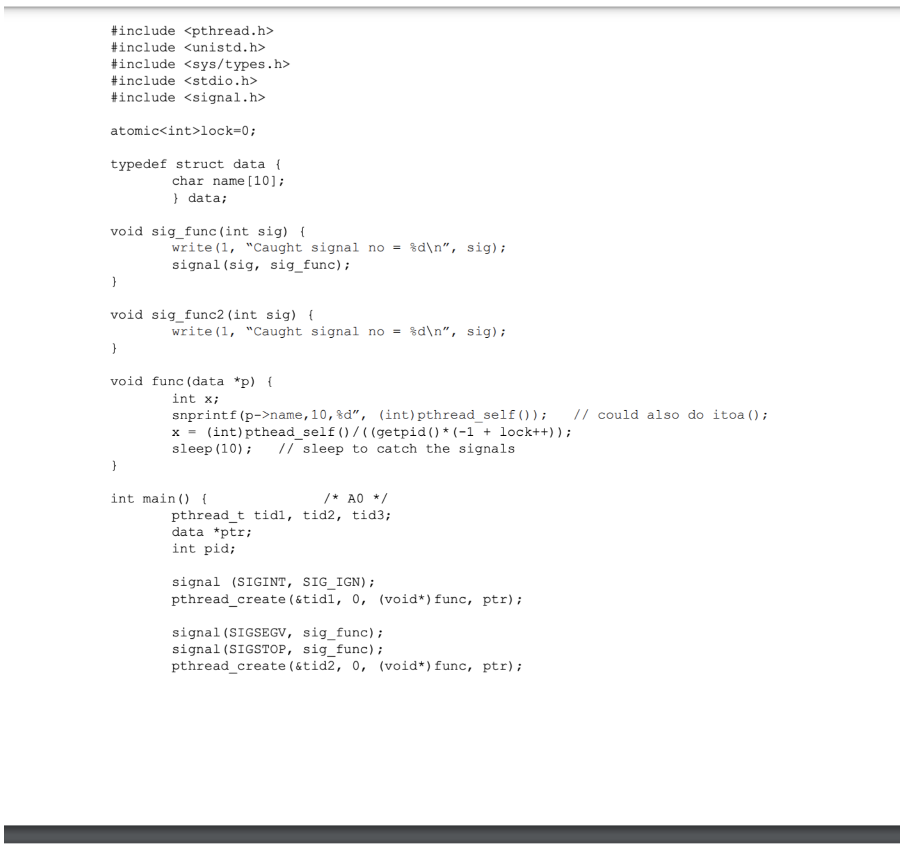
In the case of the child thread, whenever the child thread is writing data to the FIBONACCI Array, I used mutex locks to ensure that the child thread is the only thread in my process that has access to the FIBONACCI array when it is writing data to the array. To implement this, in the FibonacciIterative method(i.e. The function that the child thread is calling when iterating through the FIBONACCI Array to write all the Fibonacci Numbers), the first instruction of each loop iteration is pthread_mutex_lock(). What this does is that if the mutex lock is available then it enables the child thread to claim it and to begin doing its write operations on the array without worrying about having the parent process doing simultaneous operations on the array at the same time. If the mutex lock is not available, then this indicates that the parent thread possesses the mutex lock and is doing some operations on the array. Then, in this case, the child thread will block(i.e. wait) for the parent thread to release this lock so that it can perform write operations on the array.

If the parent acquires the mutex locks prior to the child being able to write to the array, this is where pthread_cond_wait comes in handy. One thing to notice is that the function argument list is like this: int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex). So, the 2nd argument is supposed to be a mutex. The reason for this is that the thread calling pthread_cond_wait, according to the Linux Manual, "These functions atomically release mutex and cause the calling thread to block on the condition variable cond". It is clear here that, when a thread acquires the mutex lock and then calls pthread_cond_wait, it will release that mutex lock for the other thread to require. Hence, even if the parent thread acquires the mutex lock first, my program structure has ensured that it will atomically release the mutex lock for the child thread to acquire and enter the while loop waiting for writeDone to be set to 1.

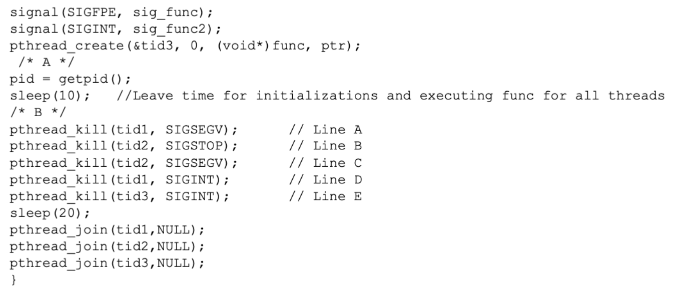
On the other hand, if the child thread acquires the mutex lock first, my program structure has ensured that it will atomically release the mutex lock for the parent thread to acquire and enter the while loop waiting for readDone to be set to 1.

I have used the condition variables to essentially control the order in which the parent and child thread receive the mutex locks. In class, we learned that we don't have control over the order in which threads are context switched onto the processor. Hence, if I use mutex locks and don't use condition variables, this may present a weird scenario. There is a possibility that the child thread remains on the processor for a period of time, retains control over the mutex locks during this period of time, and subsequently writes quite a few values to the FIBONACCI Array. Since the programmer has no direct control over the Operating System scheduling algorithms, we have no way of controlling how long each thread remains on the processor before it is context switched out. In this problem, the requirement states that the parent thread MUST read the values from the FIBONACCI Array immediately after the child thread has written it to the array. Hence, I used condition variables to generate this sequence of operations: child writes data to array first, parent then reads data from array, child writes data to array, then parent reads data from array.

Problem 4A



```
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <signal.h>

atomic<int>lock=0;

typedef struct data {
        char name[10];
        } data;

void sig_func(int sig) {
        write(1, "Caught signal no = %d\n", sig);
        signal(sig, sig_func);
}

void sig_func2(int sig) {
        write(1, "Caught signal no = %d\n", sig);
}

void func(data *p) {
        int x;
        snprintf(p->name,10,%d", (int)pthread_self());   // could also do itoa();
        x = (int)pthead_self()/((getpid()*(-1 + lock++));
        sleep(10);   // sleep to catch the signals
}

int main() {               /* A0 */
        pthread_t tid1, tid2, tid3;
        data *ptr;
        int pid;

        signal (SIGINT, SIG_IGN);
        pthread_create(&tid1, 0, (void*)func, ptr);

        signal(SIGSEGV, sig_func);
        signal(SIGSTOP, sig_func);
        pthread_create(&tid2, 0, (void*)func, ptr);
```



```
        signal(SIGFPE, sig_func);
        signal(SIGINT, sig_func2);
        pthread_create(&tid3, 0, (void*)func, ptr);
         /* A */
        pid = getpid();
        sleep(10);   //Leave time for initializations and executing func for all threads
        /* B */
        pthread_kill(tid1, SIGSEGV);      // Line A
        pthread_kill(tid2, SIGSTOP);      // Line B
        pthread_kill(tid2, SIGSEGV);      // Line C
        pthread_kill(tid1, SIGINT);       // Line D
        pthread_kill(tid3, SIGINT);       // Line E
        sleep(20);
        pthread_join(tid1,NULL);
        pthread_join(tid2,NULL);
        pthread_join(tid3,NULL);
        }
```

Part 1:

Question1 :

We cannot make a determination as to which thread will execute first. This solely depends on the Operating System Scheduling Algorithms. Just because we create a thread first DOES NOT mean that it will execute on the CPU first. Thread tid1, tid2, or tid3 could execute on the processor first. The Operating System scheduling algorithm will determine this.

The thread that executes function “func” first will print the value of the name field in the data structure “p” which is passed as a parameter to func when the thread was created(i.e. This is basically the ptr variable that was passed to the thread when pthread_create was called). The thread Thread will also print its thread ID. Finally, since this thread is the first thread to execute func, the value of “lock” for this thread will be 0. Hence, this thread will execute the statement (int)pthead_self()/((getpid()*(-1 + 0)). Let y represent the thread ID of this thread. Hence, the statement x = y / (-1) will be executed and x will be set equal to -y.

An important thing to note is that since the lock variable is atomic, the instruction streams for each thread that are accessing/setting the value of the lock variable will not be intertwined. Hence. When this thread is reading the value of lock and then subsequently setting the value of lock at the end, another thread won’t try to change the value of lock or access it.

Hence, this thread will increment the value of lock by 1. After this thread is done executing the function, the value of lock will be 1.

The thread that executes function “func” second will print the value of the name field in the data structure “p” which is passed as a parameter to func when the thread was created(i.e. This is basically the ptr variable that was passed to the thread when pthread_create was called). The thread will also print its thread ID. Finally, since this thread is the second thread to execute func, the value of “lock” for this thread will be 1. Hence, this thread will execute the statement (int)pthead_self()/((getpid()*(-1 + 1)). Let y represent the thread ID of this thread. Hence, the statement x = y / (0) will be executed.

Dividing by Zero clearly will produce an error. Hence, a SIGFPE Signal will be generated synchronously for this thread. There are two cases to observe here:

1st case: If this thread is thread tid 1 or thread tid 2:

Prior to calling pid_create for thread tid2, we have not defined a signal handler for this thread. Furthermore, this is the first time that SIGFPE has been generated in our program. Hence, when the SIGFPE signal is generated synchronously for this thread, this thread will execute the default signal handler for the SIGFPE signal which terminates the program. Hence, the SIGFPE signal will cause this thread to be terminated.

this thread will increment the value of lock by 1. After this thread is done executing the function, the value of lock will be 2.

2nd case: if this thread is thread tid 3

When thread tid3 executes the function “func”, it will print the value of the name field in the data structure “p” which is passed as a parameter to func when tid3 was created(i.e. This is basically the ptr variable that was passed to tid3 when pthread_create was called). Thread tid3 will also print its thread ID. Finally, since thread tid3 is the second thread to execute func, the value of “lock” for tid3 will be 1. Hence, thread tid3 will execute the statement (int)pthead_self()/((getpid()*(-1 + 1)). Let y represent the thread ID of tid3. Hence, the statement x = y / (0) will be executed.

Dividing by Zero clearly will produce an error. Hence, a SIGFPE Signal will be generated synchronously for thread ID tid3. Furthermore, this is the first time that SIGFPE has been generated in our program. Hence, when the SIGFPE signal is generated synchronously for thread tid3, thread tid3 will execute the sigfunc function for the SIGFPE signal. This will result in thread tid3 will print “Caught signal no = 8” since SIGFPE has a signal number 8. Another thing to notice is that sigfunc uses the signal() system call to re-install sigfunc as a signal handler for the SIGFPE signal. Hence, after thread tid3 is done executing the sigfunc function, it will still retain sigfunc as a handler for the SIGFPE signal.

Thread tid3 will increment the value of lock by 1. After thread tid3 is done executing the function, the value of lock will be 2.

The thread that executes function “func” third will print the value of the name field in the data structure “p” which is passed as a parameter to func when the thread was created(i.e. This is basically the ptr variable that was passed to the thread when pthread_create was called). The thread will also print its thread ID. Finally, since this thread is the third thread to execute func, the value of “lock” for this thread will be 2. Hence, this thread will execute the statement (int)pthead_self()/((getpid()*(-1 + 2)). Let y represent the thread ID of this thread. Hence, the statement x = y / (1) will be executed and x will be set equal to y.

An important thing to note is that since the lock variable is atomic, the instruction streams for each thread that are accessing/setting the value of the lock variable will not be intertwined. Hence. When this thread is reading the value of lock and then subsequently setting the value of lock at the end, another thread won’t try to change the value of lock or access it.

Hence, this thread will increment the value of lock by 1. After this thread is done executing the function, the value of lock will be 3.

Question2:

When thread tid1 executes the function “func” first, it will print the value of the name field in the data structure “p” which is passed as a parameter to func when tid1 was created(i.e. This is basically the ptr variable that was passed to tid1 when pthread_create was called). Thread tid1 will also print its thread ID. Finally, since thread tid1 is the first thread to execute func, the value of “lock” for tid1 will be 0. Hence, thread tid1 will execute the statement (int)pthead_self()/((getpid()*(-1 + 0)). Let y represent the thread ID of tid1. Hence, the statement x = y / (-1) will be executed and x will be set equal to -y.

An important thing to note is that since the lock variable is atomic, the instruction streams for each thread that are accessing/setting the value of the lock variable will not be intertwined. Hence. When thread tid1 is reading the value of lock and then subsequently setting the value of lock at the end, another thread won't try to change the value of lock or access it.

Hence, Thread tid1 will increment the value of lock by 1. After thread tid1 is done executing the function, the value of lock will be 1.

Since we are only given information that thread tid1 is the first to execute and the Operating System runs scheduling algorithms that run behind the scenes(i.e. We don't really have access to these scheduling algorithms nor do we know how the OS schedules the threads to run on the CPU).

Hence, thread with tid2 can be the second thread to run or thread tid3 can be the second thread to run

Case 1: Thread tid2 executes second

When thread tid2 executes the function "func", it will print the value of the name field in the data structure "p" which is passed as a parameter to func when tid2 was created(i.e. This is basically the ptr variable that was passed to tid2 when pthread_create was called). Thread tid2 will also print its thread ID. Finally, since thread tid2 is the second thread to execute func, the value of "lock" for tid2 will be 1. Hence, thread tid2 will execute the statement (int)pthead_self()/((getpid()*(-1 + 1)). Let y represent the thread ID of tid2. Hence, the statement x = y / (0) will be executed.

Dividing by Zero clearly will produce an error. Hence, a SIGFPE Signal will be generated synchronously for thread ID tid2. Prior to calling pid_create for thread tid2, we have not defined a signal handler for thread tid2. Furthermore, this is the first time that SIGFPE has been generated in our program. Hence, when the SIGFPE signal is generated synchronously for thread tid2, thread tid2 will execute the default signal handler for the SIGFPE signal which terminates the program. Hence, the SIGFPE signal will cause thread tid2 to be terminated.

Thread tid2 will increment the value of lock by 1 . After thread tid2 is done executing the function, the value of lock will be 2.

When thread tid3 executes the function "func", it will print the value of the name field in the data structure "p" which is passed as a parameter to func when tid3 was created(i.e. This is basically the ptr variable that was passed to tid3 when pthread_create was called). Thread tid3 will also print its thread ID. Finally, since thread tid3 is the third thread to execute func, the value of "lock" for tid3 will be 2. Hence, thread tid3 will execute the statement (int)pthead_self()/((getpid()*(-1 + 2)). Let y represent the thread ID of tid3. Hence, the statement x = y / (1) will be executed and x will be set equal to y.

An important thing to note is that since the lock variable is atomic, the instruction streams for each thread that are accessing/setting the value of the lock variable will not be intertwined. Hence. When thread tid3 is reading the value of lock and then subsequently setting the value of lock at the end, another thread won't try to change the value of lock or access it.

Thread tid3 will increment the value of lock by 1 and, after thread tid3 is finished executing, the value of lock will be 3.

Case 2: Thread tid3 executes second

When thread tid3 executes the function "func", it will print the value of the name field in the data structure "p" which is passed as a parameter to func when tid3 was created(i.e. This is basically the ptr variable that was passed to tid3 when pthread_create was called). Thread tid3 will also print its thread ID. Finally, since thread tid3 is the second thread to execute func, the value of "lock" for tid3 will be 1. Hence, thread tid3 will execute the statement (int)pthead_self()/((getpid()*(-1 + 1)). Let y represent the thread ID of tid3. Hence, the statement x = y / (0) will be executed.

Dividing by Zero clearly will produce an error. Hence, a SIGFPE Signal will be generated synchronously for thread ID tid3. Furthermore, this is the first time that SIGFPE has been generated in our program. Hence, when the SIGFPE signal is generated synchronously for thread tid3, thread tid3 will execute the sigfunc function for the SIGFPE signal. This will result in thread tid3 will print "Caught signal no = 8" since SIGFPE has a signal number 8. Another thing to notice is that sigfunc uses the signal() system call to re-install sigfunc as a signal handler for the SIGFPE signal. Hence, after thread tid3 is done executing the sigfunc function, it will still retain sigfunc as a handler for the SIGFPE signal.

Thread tid3 will increment the value of lock by 1. After thread tid3 is done executing the function, the value of lock will be 2.

When thread tid2 executes the function "func" first, it will print the value of the name field in the data structure "p" which is passed as a parameter to func when tid2 was created(i.e. This is basically the ptr variable that was passed to tid2 when pthread_create was called). Thread tid2 will also print its thread ID. Finally, since thread tid2 is the third thread to execute func, the value of "lock" for tid2 will be 2. Hence, thread tid2 will execute the statement (int)pthead_self()/((getpid()*(-1 + 2)). Let y represent the thread ID of tid2. Hence, the statement x = y / (1) will be executed and x will be set equal to y.

An important thing to note is that since the lock variable is atomic, the instruction streams for each thread that are accessing/setting the value of the lock variable will not be intertwined. Hence. When thread tid2 is reading the value of lock and then subsequently setting the value of lock at the end, another thread won't try to change the value of lock or access it.

Thread tid2 will increment the value of lock by 1 and, after thread tid2 is finished executing, the value of lock will be 3.

Question 3:

When thread tid3 executes the function "func" first, it will print the value of the name field in the data structure "p" which is passed as a parameter to func when tid3 was created(i.e. This is basically the ptr variable that was passed to tid3 when pthread_create was called). Thread tid3 will also print its thread ID. Finally, since thread tid3 is the first thread to execute func, the value of "lock" for tid3 will be 0. Hence, thread tid3 will execute the statement (int)pthead_self()/((getpid()*(-1 + 0)). Let y represent the thread ID of tid3. Hence, the statement x = y / (-1) will be executed and x will be set equal to -y.

An important thing to note is that since the lock variable is atomic, the instruction streams for each thread that are accessing/setting the value of the lock variable will not be intertwined. Hence. When thread tid3 is reading the value of lock and then subsequently setting the value of lock at the end, another thread won't try to change the value of lock or access it.

Thread tid3 will increment the value of lock by 1. After thread tid3 is done executing the function, the value of lock will be 1.

Since we are only given information that thread tid3 is the first to execute and the Operating System runs scheduling algorithms that run behind the scenes(i.e. We don't really have access to these scheduling algorithms nor do we know how the OS schedules the threads to run on the CPU).

Hence, thread with tid2 can be the second thread to run or thread tid1 can be the second thread to run

Case 1: Thread tid2 executes second

When thread tid2 executes the function "func", it will print the value of the name field in the data structure "p" which is passed as a parameter to func when tid2 was created(i.e. This is basically the ptr variable that was passed to tid2 when pthread_create was called). Thread tid2 will also print its thread ID. Finally, since thread tid2 is the second thread to execute func, the value of "lock" for tid2 will be 1. Hence, thread tid2 will execute the statement (int)pthead_self()/((getpid()*(-1 + 1)). Let y represent the thread ID of tid2. Hence, the statement x = y / (0) will be executed. Dividing by Zero clearly will produce an error. Hence, a SIGFPE Signal will be generated synchronously for thread ID tid2. Prior to calling pid_create for thread tid2, we have not defined a signal handler for thread tid2. Furthermore, this is the first time that SIGFPE has been generated in our program. Hence, when the SIGFPE signal is generated synchronously for thread tid2, thread tid2 will execute the default signal handler for the SIGFPE signal which terminates the program. Hence, the SIGFPE signal will cause thread tid2 to be terminated.

Thread tid2 will increment the value of lock by . After thread tid2 is done executing the function, the value of lock will be 2.

When thread tid1 executes the function “func”, it will print the value of the name field in the data structure “p” which is passed as a parameter to func when tid1 was created(i.e. This is basically the ptr variable that was passed to tid1 when pthread_create was called). Thread tid1 will also print its thread ID. Finally, since thread tid1 is the third thread to execute func, the value of “lock” for tid1 will be 2. Hence, thread tid1 will execute the statement (int)pthead_self()/((getpid()*(-1 + 2)). Let y represent the thread ID of tid1. Hence, the statement x = y / (1) will be executed and x will be set equal to y.

An important thing to note is that since the lock variable is atomic, the instruction streams for each thread that are accessing/setting the value of the lock variable will not be intertwined. Hence. When thread tid1 is reading the value of lock and then subsequently setting the value of lock at the end, another thread won’t try to change the value of lock or access it.

Thread tid1 will increment the value of lock by 1 and, after thread tid1 is finished executing, the value of lock will be 3.

Case 2: Thread tid1 executes second

When thread tid1 executes the function “func”, it will print the value of the name field in the data structure “p” which is passed as a parameter to func when tid1 was created(i.e. This is basically the ptr variable that was passed to tid1 when pthread_create was called). Thread tid1 will also print its thread ID. Finally, since thread tid1 is the second thread to execute func, the value of “lock” for tid1 will be 1. Hence, thread tid1 will execute the statement (int)pthead_self()/((getpid()*(-1 + 1)). Let y represent the thread ID of tid1. Hence, the statement x = y / (0) will be executed.

Dividing by Zero clearly will produce an error. Hence, a SIGFPE Signal will be generated synchronously for thread ID tid1. Prior to calling pid_create for thread tid1, we have not defined a signal handler for thread tid1. Furthermore, this is the first time that SIGFPE has been generated in our program. Hence, when the SIGFPE signal is generated synchronously for thread tid1, thread tid1 will execute the default signal handler for the SIGFPE signal which terminates the program. Hence, the SIGFPE signal will cause thread tid1 to be terminated.

Thread tid1 will increment the value of lock by 1. After thread tid1 is done executing the function, the value of lock will be 2.

When thread tid2 executes the function "func" first, it will print the value of the name field in the data structure "p" which is passed as a parameter to func when tid2 was created(i.e. This is basically the ptr variable that was passed to tid2 when pthread_create was called). Thread tid2 will also print its thread ID. Finally, since thread tid2 is the third thread to execute func, the value of "lock" for tid2 will be 2. Hence, thread tid2 will execute the statement (int)pthead_self()/((getpid()*(-1 + 2)). Let y represent the thread ID of tid2. Hence, the statement x = y / (1) will be executed and x will be set equal to y.

An important thing to note is that since the lock variable is atomic, the instruction streams for each thread that are accessing/setting the value of the lock variable will not be intertwined. Hence. When thread tid2 is reading the value of lock and then subsequently setting the value of lock at the end, another thread won't try to change the value of lock or access it.

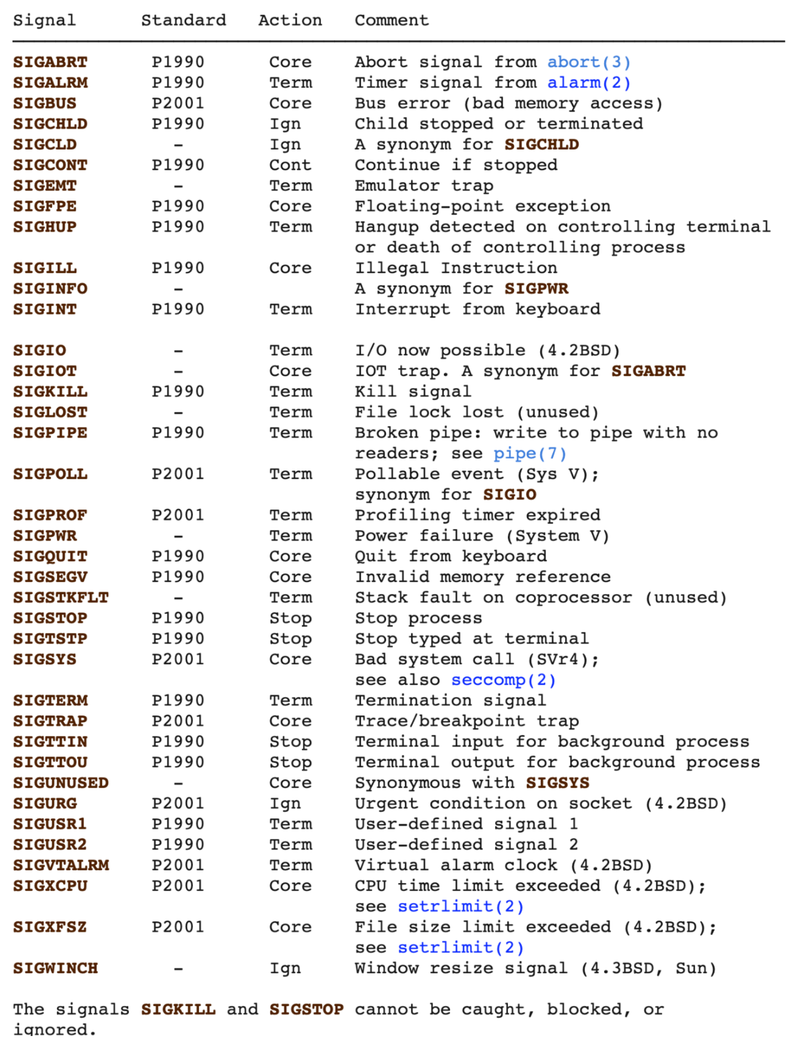
Thread tid2 will increment the value of lock by 1 and, after thread tid2 is finished executing, the value of lock will be 3.

Part 2:

Assumptions as per Problem Statement:

- each thread inherits the signal handlers set by the main process just before the call to pthread_create
- the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued
- a terminating signal that is sent to a thread alone terminates only that particular thread.

To determine the default handlers for each Signal, I referred to the Linux Manual and used this table:



Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from abort(3)
SIGALRM	P1990	Term	Timer signal from alarm(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	-	Ign	A synonym for **SIGCHLD**
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	-	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	-		A synonym for **SIGPWR**
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	-	Term	I/O now possible (4.2BSD)
SIGIOT	-	Core	IOT trap. A synonym for **SIGABRT**
SIGKILL	P1990	Term	Kill signal
SIGLOST	-	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGPOLL	P2001	Term	Pollable event (Sys V); synonym for **SIGIO**
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also seccomp(2)
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	-	Core	Synonymous with **SIGSYS**
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD); see setrlimit(2)
SIGWINCH	-	Ign	Window resize signal (4.3BSD, Sun)

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

As shown, the default action for SIGSEGV is “Core” and a “Core Dump” essentially results in the termination of the program

As shown, the default action for SIGINT is “Term” which means terminated!!

Question 1: For this question, there are 3 cases to analyze.

Case 1: Every time we execute Line C, the sig_func handler has had a chance to re-install
In this question, we are saying that we attempt to execute Line C 3 times in a row. An important thing to notice, given our assumption in the class that a particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued, is that for thread tid2 the signal handler for SIGSEGV is the sig_func function. The reason is that in our main function, we have registered the sig_func function as a handler for SIGSEGV just prior to calling pthread_create for the thread with tid2. Hence, when we execute Line C for the first time, thread tid2 will execute the sig_func handler. It will print the statement “Caught signal no = 11” to the terminal. The sig_func handler re-installs the sig_func handler for this thread. Hence, when we execute Line C for the

second time, thread tid2 will execute the sig_func handler. It will print the statement "Caught signal no = 11" to the terminal. The sig_func handler re-installs the sig_func handler for this thread. Hence, when we execute Line C for the third time, thread tid2 will execute the sig_func handler. It will print the statement "Caught signal no = 11" to the terminal. The sig_func handler re-installs the sig_func handler for this thread.

Case 2: The 2nd execution of Line C is done during the execution of the sig_handler function(i.e. The sig_handler function is handling the first execution of Line C) just prior to it reinstalling itself
In this question, we are saying that we attempt to execute Line C 3 times in a row. An important thing to notice, given our assumption in the class that a particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued, is that for thread tid2 the signal handler for SIGSEGV is the sig_func function. The reason is that in our main function, we have registered the sig_func function as a handler for SIGSEGV just prior to calling pthread_create for the thread with tid2. Hence, when we execute Line C for the first time, thread tid2 will execute the sig_func handler. It will print the statement "Caught signal no = 11" to the terminal. At this point, let's say we execute Line C for the second time. We are given information in this problem that we are using an "outdated Linux Operating System" According to the Linux Manual, "In the original UNIX systems, when a handler that was established using signal() was invoked by the delivery of a signal, the disposition of the signal would be reset to SIG_DFL, and the system **did not block delivery** of further instances of the signal." The most important thing to take away from this is that, when the sig_func handler is executing, if we send a SIGSEGV signal to thread tid2, the system will NOT block this signal. Hence, during the execution of the sig_func handler(i.e. Prior to reinstallation), when we execute Line C for a second time, we will be sending another SIGSEGV signal to thread tid2. However, we must notice that, since the signal handler has started to execute, the signal handler for the SIGSEGV signal in thread tid2 will be reset to the default handler. In our Linux Operating System, the default handler for SIGSEGV essentially will terminate the thread. Hence, when we execute Line C for a second time in this case, we will terminate thread tid2. Furthermore, when we execute line C for a third time, since thread tid2 is already terminated, it won't really have much of an effect. Furthermore, at the point when we execute Line C, we haven't yet called pthread_join. Hence, the thread resources will still be in the system despite the fact that is has been terminated after the second execution of Line C. So, sending a terminating signal to a "zombie thread" is essentially of no use and won't do anything notable. According to POSIX Standards, "POSIX.1-2008 recommends that if an implementation detects the use of a thread ID after the end of its lifetime, pthread_kill() should return the error ESRCH." So, depending on which Operating System we have, when we call pthread_kill and send a terminating signal to thread tid2 after it has been terminated, pthread_kill may return the error ESRCH.

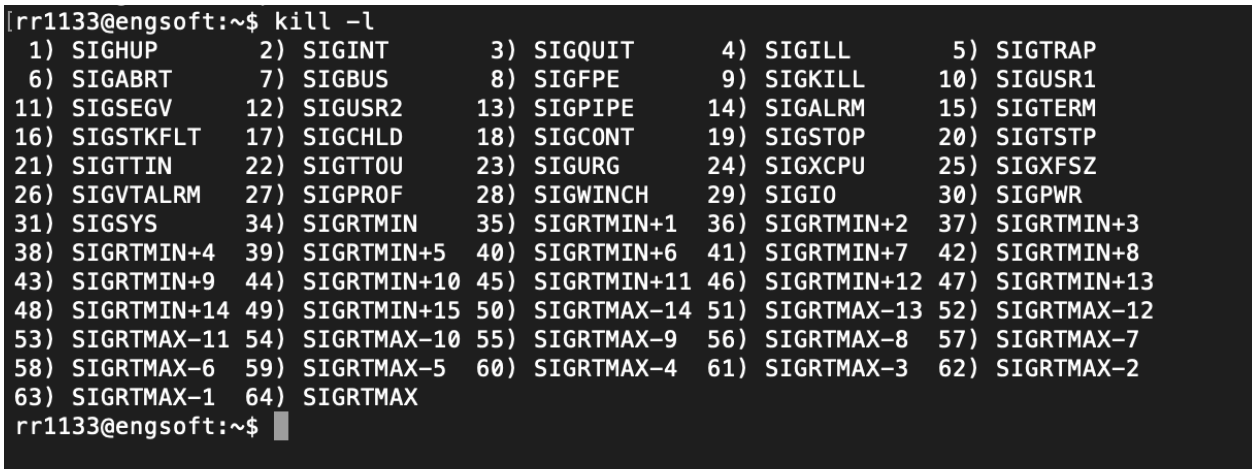
Case 3: The 3rd execution of Line C is done during the execution of the sig_handler function(i.e. The function is handling the first execution of Line C) just prior to it reinstalling itself

In this question, we are saying that we attempt to execute Line C 3 times in a row. An important thing to notice, given our assumption in the class that a particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued, is that for thread tid2 the signal handler for SIGSEGV is the sig_func function. The reason is that in our main function, we have registered the sig_func function as a handler for SIGSEGV just prior to calling pthread_create for the thread with tid2. Hence, when we execute Line C for the first time, thread tid2 will execute the sig_func handler. It will print the statement "Caught signal no = 11" to the terminal. The sig_func handler re-installs the sig_func handler for this thread. Hence, when we execute Line C for the second time, thread tid2 will execute the sig_func handler. It will print the statement "Caught signal no = 11" to the terminal. At this point, let's say we execute Line C for the 3rd time. According to the Linux Manual, "In the original UNIX systems, when a handler that was established using signal() was invoked by the delivery of a signal, the disposition of the signal would be reset to SIG_DFL, and the system did not block delivery of further instances of the signal." The most important thing to take away from this is that, when the sig_func handler is executing, if we send a SIGSEGV signal to thread tid2, the system will NOT block this signal. Hence, during the execution of the sig_func handler(i.e. Prior to reinstallation), when we execute Line C for a 3rd time, we will be sending another SIGSEGV signal to thread tid2. However, we must notice that, since the signal handler has started to execute, the signal handler for the SIGSEGV signal in thread tid2 will be reset to the default handler. In our Linux Operating System, the default handler for SIGSEGV essentially will terminate the thread. Hence, when we execute Line C for a 3rd time in this case, we will terminate thread 2.

For Question 2, since our signal handler never re-installs itself, there is only 1 case to analyze here
Question 2: In this question, we are saying that we attempt to execute Line E 3 times in a row. An important thing to notice, given our assumption in the class that a particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued, is that for thread tid3 the signal handler for SIGINT is the sig_func2 function. The reason is that in our main function, we have registered the sig_func2 function as a handler for SIGINT just prior to calling pthread_create for the thread with tid3. Hence, when we execute Line E for the first time, thread tid3 will execute the sig_func2 handler. It will print the statement "Caught signal no = 2" to the terminal. However, something to notice here is that sig_func2 does NOT reinstall itself after the first invocation. Since we are using an Older Linux OS, upon the first invocation of sig_func2, the handler for that particular signal will be reset back to the default handler. Hence, when we execute Line E for a second time, since the default action for SIGINT is to terminate, thread tid3 will be terminated. Furthermore, when we execute line E for a third time, since thread tid3 is already terminated, it won't really have much of an effect. Furthermore, at the point when we execute Line E, we haven't yet called pthread_join. Hence, the thread resources will still be in the system despite the fact that is has been terminated after the first execution of Line E. So, sending a terminating signal to a "zombie thread" is essentially of no use and won't do anything notable. According to POSIX Standards, "POSIX.1-2008 recommends that if an implementation detects the use of a thread ID after the end of its lifetime, pthread_kill() should return the error ESRCH." So, depending on which Operating System we have, when we call pthread_kill and send a terminating signal to thread tid3 after it has been terminated, pthread_kill may return the error ESRCH.

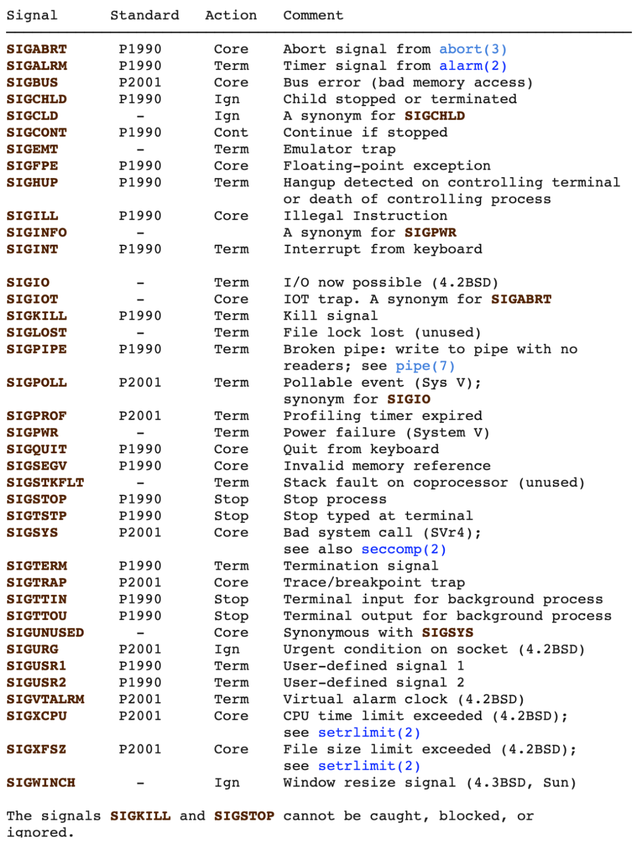
Part 3:

To get a list of Signals and the numbers they correspond to, I executed the “kill -l” statement in the terminal:



```
[rr1133@engsoft:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
rr1133@engsoft:~$ 
```

To determine the default handlers for each Signal, I referred to the Linux Manual and used this table:



Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from abort(3)
SIGALRM	P1990	Term	Timer signal from alarm(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	-	Ign	A synonym for **SIGCHLD**
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	-	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	-		A synonym for **SIGPWR**
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	-	Term	I/O now possible (4.2BSD)
SIGIOT	-	Core	IOT trap. A synonym for **SIGABRT**
SIGKILL	P1990	Term	Kill signal
SIGLOST	-	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGPOLL	P2001	Term	Pollable event (Sys V); synonym for **SIGIO**
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also seccomp(2)
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	-	Core	Synonymous with **SIGSYS**
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD); see setrlimit(2)
SIGWINCH	-	Ign	Window resize signal (4.3BSD, Sun)

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Question 1: In this question, we are sending 3 SIGINT signals to the entire process. When we send a signal to the entire process, we must first look to see which threads are blocking this signal and which threads are NOT blocking this signal. However, in our example, no thread is blocking the SIGINT signal. In this case, every time we send the SIGINT signal to our process, our operating system will pick an arbitrary thread to deliver this SIGINT Signal to. Since none of the three threads are blocking this signal, there are multiple cases for us to analyze here.

Note: Since thread tid1 and tid2 both have SIG_IGN(i.e. ignore) as the handler for SIGINT, I will consolidate both of these threads into 1 case.

Let's start by looking at the first time we send the SIGINT Signal to the entire process:

Case 1: The SIGINT signal is delivered to thread tid1 or tid2. In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid1 and tid2 for the SIGINT signal is to ignore the signal. Another thing to remember here is that, upon invocation of the handler for the first time, the signal handler will be reset to the default handler since we are using an Older Linux Operating System.

Hence, the first time we execute the kill(pid, 2) line, if the Operating System chooses to send this signal to either thread tid1 or thread tid2, it will be ignored

Case 2: The SIGINT signal is delivered to the thread tid3. In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid3 for the SIGINT signal is to execute the sig_func2 signal. It will print the statement "Caught signal no = 2" to the terminal. Another thing to remember here is that, since sig_func2 doesn't reinstall itself, the signal handler will be reset to the default handler since we are using an Older Linux Operating System.
Let's now look at the second time we send the SIGINT Signal to the entire process:

Case 1: the thread that handled the first SIGINT signal ALSO handles the second signal

As shown previously, after the first SIGINT signal is sent to the process, regardless of which thread the SIGINT signal is delivered to, that thread will have its handler for SIGINT reset to the default handler. Hence, for this case, the SIGINT signal, when delivered to this particular thread, will have a terminating flavor. Since we used the kill() system call here, we are saying that this termination will be propagated to all threads and will terminate our entire process as a result.

Case 2: thread tid1 handled the first SIGINT signal and thread tid2 handles the second SIGINT signal

In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid2 for the SIGINT signal is to ignore the signal. Another thing to remember here is that, upon invocation of the handler for the first time, the signal handler will be reset to the default handler since we are using an Older Linux Operating System.

Hence, the second time we execute the kill(pid, 2) line, if the Operating System chooses to send this signal to thread tid2, it will be ignored

Case 3: thread tid1 handled the first SIGINT signal and thread tid3 handles the second SIGINT signal

In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid3 for the SIGINT signal is to execute the sig_func2 signal. It will print the statement “Caught signal no = 2” to the terminal. Another thing to remember here is that, since sig_func2 doesn’t reinstall itself, the signal handler will be reset to the default handler since we are using an Older Linux Operating System.

Case 4: thread tid2 handled the first SIGINT signal and thread tid1 handles the second SIGINT signal

In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid1 for the SIGINT signal is to ignore the signal. Another thing to remember here is that, upon invocation of the handler for the first time, the signal handler will be reset to the default handler since we are using an Older Linux Operating System.

Hence, the second time we execute the kill(pid, 2) line, if the Operating System chooses to send this signal to thread tid1, it will be ignored

Case 5: thread tid2 handled the first SIGINT signal and thread tid3 handles the second SIGINT signal

In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid3 for the SIGINT signal is to execute the sig_func2 signal. It will print the statement “Caught signal no = 2” to the terminal. Another thing to remember here is that, since sig_func2 doesn’t reinstall itself, the signal handler will be reset to the default handler since we are using an Older Linux Operating System.

Case 6: thread tid3 handled the first SIGINT signal and thread tid2 handles the second SIGINT signal

In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid2 for the SIGINT signal is to ignore the signal. Another thing to remember here is that, upon invocation of the handler for the first time, the signal handler will be reset to the default handler since we are using an Older Linux Operating System.

Hence, the second time we execute the kill(pid, 2) line, if the Operating System chooses to send this signal to thread tid2, it will be ignored

Case 7: thread tid3 handled the first SIGINT signal and thread tid1 handles the second SIGINT signal

In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid1 for the SIGINT signal is to ignore the signal. Another thing to remember here is that, upon invocation of the handler for the first time, the signal handler will be reset to the default handler since we are using an Older Linux Operating System.

Hence, the second time we execute the kill(pid, 2) line, if the Operating System chooses to send this signal to thread tid1, it will be ignored

Let's now look at what happens the third time we send the SIGINT (assuming the entire process hasn't been terminated by this point)

Case 1: The Signal is sent to thread tid1 to be handled and the handler for SIGINT for thread tid1 has already been reset during a prior invocation

Thread tid1 will have its handler for SIGINT reset to the default handler. Hence, for this case, the SIGINT signal, when delivered to this particular thread, will have a terminating flavor. Since we used the kill() system call here, we are saying that this termination will be propagated to all threads and will terminate our entire process as a result.

Case 2: The Signal is sent to thread tid1 to be handled and the handler for SIGINT for thread tid1 but HAS NOT been reset during a prior invocation

In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid1 for the SIGINT signal is to ignore the signal.

Case 3: The Signal is sent to thread tid2 to be handled and the handler for SIGINT for thread tid2 has already been reset during a prior invocation

Thread tid2 will have its handler for SIGINT reset to the default handler. Hence, for this case, the SIGINT signal, when delivered to this particular thread, will have a terminating flavor. Since we used the kill() system call here, we are saying that this termination will be propagated to all threads and will terminate our entire process as a result.

Case 4: The Signal is sent to thread tid2 to be handled and the handler for SIGINT for thread tid2 but HAS NOT been reset during a prior invocation

In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid2 for the SIGINT signal is to ignore the signal.

Case 5: The Signal is sent to thread tid3 to be handled and the handler for SIGINT for thread tid3 has already been reset during a prior invocation

Thread tid3 will have its handler for SIGINT reset to the default handler. Hence, for this case, the SIGINT signal, when delivered to this particular thread, will have a terminating flavor. Since we used the kill() system call here, we are saying that this termination will be propagated to all threads and will terminate our entire process as a result.

Case 6: The Signal is sent to thread tid3 to be handled and the handler for SIGINT for thread tid3 but HAS NOT been reset during a prior invocation

In this problem, we are also assuming that each thread inherits the signal handlers set by the main process just before the call to pthread_create, and also assume that that the particular thread is impervious to the changes of signal handlers from other threads or the main process once the call to pthread_create has been issued. Hence, the signal handler for thread tid3 for the SIGINT signal is to execute the sig_func2 signal. It will print the statement "Caught signal no = 2" to the terminal.

Question 2:

In this question, we are sending 3 SIGSEGV signals to the entire process. When we send a signal to the entire process, we must first look to see which threads are blocking this signal and which threads are NOT blocking this signal. However, in our example, no thread is blocking the SIGSEGV signal. In this case, every time we send the SIGSEGV signal to our process, our operating system will pick an arbitrary thread to deliver this SIGSEGV Signal to. Since none of the three threads are blocking this signal, there are multiple cases for us to analyze here.

Note: Since thread tid2 and thread tid3 have the same handler for SIGSEGV Signal, I have consolidated the analysis of thread tid2 and thread tid3 into one case.

Let's look at the first time we send the SIGSEGV Signal
Case 1: It is delivered to thread tid1
As shown by our program, thread tid1 has the default handler for SIGSEGV which is just to terminate. Since the response to the signal is of a terminating flavor and we are using the kill() system call here to send a signal to the entire process, the SIGSEGV will terminate the process

Case 2: It is delivered to thread tid2 or tid3
The SIGSEGV handler for thread tid2 and tid3 is the sig_func function. This function does reinstall itself after first invocation. So, when the SIGSEGV signal is delivered to thread tid2 or thread tid3, the thread will just print "Caught signal no = 11". However, since sig_func does reinstall itself after the first invocation, the next time either one of these threads handle the SIGSEGV signal, they will use the sig_func handler again!

Let's look at the second time we send the SIGSEGV Signal(assuming our process is still alive at this point. For this to happen, this means that thread tid2 or thread tid3 handled this signal during the first time we sent the SIGSEGV signal):

Case 1: It is delivered to thread tid1
As shown by our program, thread tid1 has the default handler for SIGSEGV which is just to terminate. Since the response to the signal is of a terminating flavor and we are using the kill() system call here to send a signal to the entire process, the SIGSEGV will terminate the process

Case 2: thread tid2 handled the first SIGSEGV signal and thread tid3 will handle the second SIGSEGV signal

The SIGSEGV handler for thread tid3 is the sig_func function. This function does reinstall itself after first invocation. So, when the SIGSEGV signal is delivered to thread tid3, the thread will just print "Caught signal no = 11". However, since sig_func does reinstall itself after the first invocation, the next time thread tid3 handles the SIGSEGV signal, they will use the sig_func handler

Case 3: thread tid3 handled the first SIGSEGV signal and thread tid2 will handle the second SIGSEGV signal

The SIGSEGV handler for thread tid2 is the sig_func function. This function does reinstall itself after first invocation. So, when the SIGSEGV signal is delivered to thread tid2, the thread will just print "Caught signal no = 11". However, since sig_func does reinstall itself after the first invocation, the next time thread tid2 handles the SIGSEGV signal, they will use the sig_func handler

Case 4: The thread that handled the first SIGSEGV signal(i.e. Thread tid2 or thread tid3) also handles the second SIGSEGV signal

Essentially, in other words, this case handles the following possibilities of our program:

- Thread tid2 handled the first SIGSEGV signal and is also going to handle the second SIGSEGV signal
- Thread tid3 handled the first SIGSEGV signal and is also going to handle the second SIGSEGV signal

The SIGSEGV handler for this thread is the sig_func function. This function does reinstall itself after first invocation. So, when the SIGSEGV signal is delivered to this thread, the thread will just print "Caught signal no = 11". However, since sig_func does reinstall itself after the first invocation, the next time this thread handles the SIGSEGV signal, they will use the sig_func handler

Let's look at the third time we send the SIGSEGV signal(assuming our process is still alive at this point(For this to happen, this means that thread tid2 and thread tid3 handled the first two SIGSEGV signals).

In other words, for the process to still be alive at this point, as shown by my previous analysis, either of the following must be true:

- Thread tid2 handled the first SIGSEGV signal and Thread tid3 handled the second SIGSEGV signal
- Thread tid3 handled the first SIGSEGV signal and thread tid2 handled the second SIGSEGV signal

Case 1: It is delivered to thread tid1
As shown by our program, thread tid1 has the default handler for SIGSEGV which is just to terminate. Since the response to the signal is of a terminating flavor, the SIGSEGV will terminate the process

Case 2: thread tid2 or thread tid3 will handle the 3rd SIGSEGV signal

The SIGSEGV handler for thread tid2 and tid3 is the sig_func function. This function does reinstall itself after first invocation. So, when the SIGSEGV signal is delivered to thread tid2 or thread tid3, the thread will just print "Caught signal no = 11". However, since sig_func does reinstall itself after the first invocation, the next time either one of these threads handle the SIGSEGV signal, they will use the sig_func handler

Question 3:
In this question, we are sending 3 SIGFPE signals to the entire process. When we send a signal to the entire process, we must first look to see which threads are blocking this signal and which threads are NOT blocking this signal. However, in our example, no thread is blocking the SIGFPE signal. In this case, every time we send the SIGFPE signal to our process, our operating system will pick an arbitrary thread to deliver this SIGFPE Signal to. Since none of the three threads are blocking this signal, there are multiple cases for us to analyze here.

Let's look at the first time we send the SIGFPE Signal
Note: Since thread tid1 and tid2 have the same handler for SIGFPE, I have consolidated their analysis into the same case.
Case 1: It is delivered to thread tid1 or tid2
As shown by our program, thread tid1 and tid2 have the default handler for SIGFPE which is just to terminate. Since the response to the signal is of a terminating flavor, the SIGFPE will terminate the process

Case 2: It is delivered to thread tid3

It is clear in our program that thread tid3 has the sig_func handler for the SIGFPE Signal. Hence, it will print "Caught signal no = 8". The sig_func handler reinstalls itself so, the next time thread tid3 is delivered a SIGFPE signal, it can re-use the sig_func handler

Let's look at the second time we send the SIGFPE Signal(i.e. Assuming that the process hasn't been terminated at this point. For the process not to have been terminated at this point, we know, as per my previous analysis, that the first SIGFPE signal was delivered to and handled by thread tid3)
Case 1: It is delivered to thread tid1 or tid2
As shown by our program, thread tid1 and tid2 have the default handler for SIGFPE which is just to terminate. Since the response to the signal is of a terminating flavor, the SIGFPE will terminate the process

Case 2: It is delivered to thread tid3

It is clear in our program that thread tid3 has the sig_func handler for the SIGFPE Signal. Hence, it will print "Caught signal no = 8". The sig_func handler reinstalls itself so, the next time thread tid3 is delivered a SIGFPE signal, it can re-use the sig_func handler

Let's look at the 3rd time we send the SIGFPE Signal(i.e. Assuming that the process hasn't been terminated at this point. For the process not to have been terminated at this point, we know, as per my previous analysis, that the first SIGFPE signal was delivered to and handled by thread tid3)
Case 1: It is delivered to thread tid1 or tid2
As shown by our program, thread tid1 and tid2 have the default handler for SIGFPE which is just to terminate. Since the response to the signal is of a terminating flavor, the SIGFPE will terminate the process

Case 2: It is delivered to thread tid3

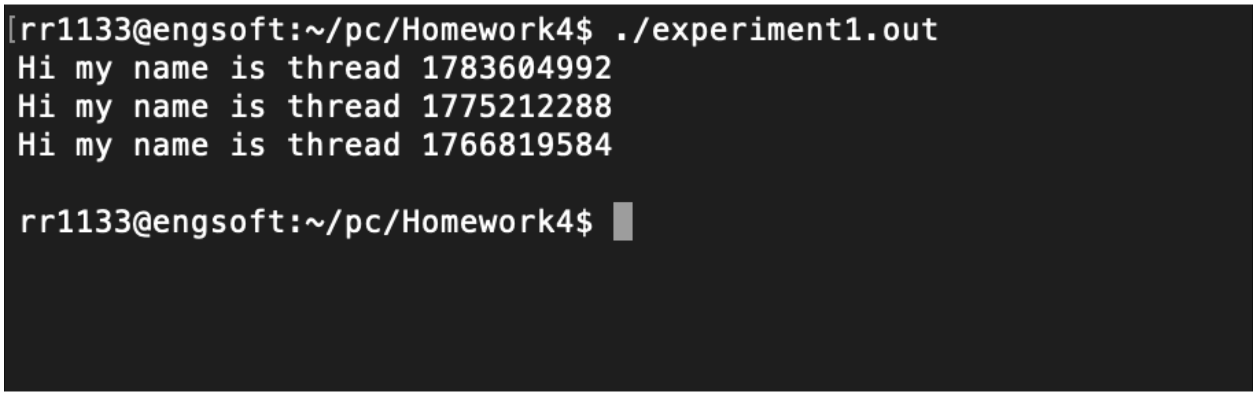
It is clear in our program that thread tid3 has the sig_func handler for the SIGFPE Signal. Hence, it will print "Caught signal no = 8". The sig_func handler reinstalls itself so, the next time thread tid3 is delivered a SIGFPE signal, it can re-use the sig_func handler

Question 4: In this question, we are sending 3 SIGTERM signals to the entire process. When we send a signal to the entire process, we must first look to see which threads are blocking this signal and which threads are NOT blocking this signal. However, in our example, no thread is blocking the SIGTERM signal. In this case, every time we send the SIGTERM signal to our process, our operating system will pick an arbitrary thread to deliver this SIGTERM Signal to. However, no thread in our program has defined a handler for SIGTERM. Hence, no matter which thread the OS delivers this signal to, it will terminate that thread, and since this response is a terminating flavor, it will terminate the process after the first SIGTERM signal is delivered. Hence, the remaining two SIGTERM signals have no effect.

Part 2:

**NOTE: All Experiments were run on the Engsoft Machine with 20.0.4 Ubuntu Operating System

<u>The first experiment that was conducted was to test what the case on my OS is regarding sending a terminating signal to only one of the threads</u>



```
rr1133@engsoft:~/pc/Homework4$ ./experiment1.out
Hi my name is thread 1783604992
Hi my name is thread 1775212288
Hi my name is thread 1766819584

rr1133@engsoft:~/pc/Homework4$ 
```

Output of Experiment

Code:

```
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdatomic.h>
```

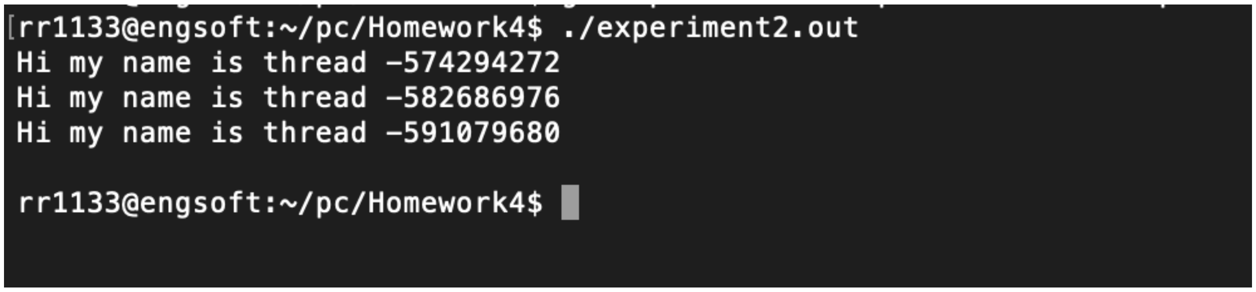
```c
//experiment 1
void * firstThread(void * ptr){
   printf("Hi my name is thread %d\n", (int)(pthread_self()));
   sleep(30);  //long sleep to give time for main thread to send SIGINT signals
   printf("Hi this thread %d is exiting\n", (int)(pthread_self()));
   return NULL;
}
void * otherThreads(void * ptr){
   printf("Hi my name is thread %d\n", (int)(pthread_self()));
   sleep(10);
   printf("Hi this thread %d is exiting\n", (int)(pthread_self()));
   return NULL;
}

int main(){

   pthread_t threadID[3];
   pthread_attr_t threadAttributes[3];

   void * ptr = NULL;

   for(int i = 0; i < 3; i ++){
       pthread_attr_init(&threadAttributes[i]);
       if(i == 0){
           pthread_create(&threadID[i],&threadAttributes[i], firstThread, ptr);
       }
       else{
           pthread_create(&threadID[i],&threadAttributes[i], otherThreads, ptr);
       }
   }

   sleep(5); //Give some time for the threads to be created and start running
thread_func

   int firstReturnValue = pthread_kill(threadID[0], SIGINT);  //Send a SIGINT signal
to the first thread

   if(firstReturnValue == 0){
       printf("First Thread was successfully terminated by SIGINT signal\n");
   }

   printf("Time to reap resources from threads that are still in system\n");
```

```
    for(int i = 0; i < 3; i ++){
        pthread_join(threadID[i], NULL);
    }




}
```

As shown in the Code, three threads were created and the SIGINT(i.e. A terminating signal was sent to only the first thread). However, none of the print statements in the main thread were executed after the pthread_kill system call was invoked. This tells me that the main thread was terminated. Furthermore, none of the child threads executed their statements after the sleep() system call. This also tells me that the child thread were terminated

Conclusion: When pthread_kill is used to send a terminating signal to a specific thread, the entire thread group is taken down

<u>What happens when we send a terminating signal to the process and there are no handlers defined at all?</u>



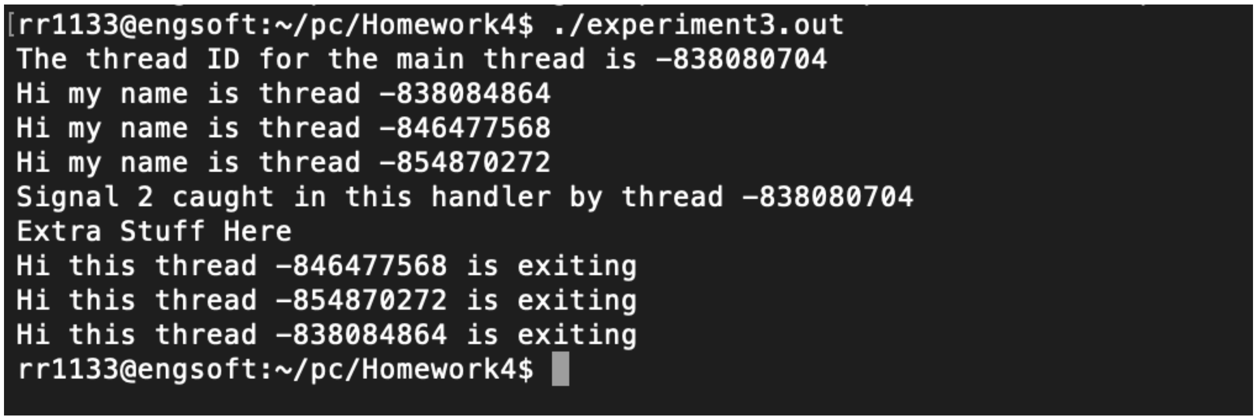
```
rr1133@engsoft:~/pc/Homework4$ ./experiment2.out
 Hi my name is thread -574294272
 Hi my name is thread -582686976
 Hi my name is thread -591079680

rr1133@engsoft:~/pc/Homework4$ 
```

```
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdatomic.h>

//experiment 2
void * firstThread(void * ptr){
   printf("Hi my name is thread %d\n", (int)(pthread_self()));
   sleep(30);  //long sleep to give time for main thread to send SIGINT signals
   printf("Hi this thread %d is exiting\n", (int)(pthread_self()));
   return NULL;
}
void * otherThreads(void * ptr){
   printf("Hi my name is thread %d\n", (int)(pthread_self()));
   sleep(10);
   printf("Hi this thread %d is exiting\n", (int)(pthread_self()));
   return NULL;
}

int main(){

   pthread_t threadID[3];
   pthread_attr_t threadAttributes[3];

   void * ptr = NULL;

   for(int i = 0; i < 3; i ++){
       pthread_attr_init(&threadAttributes[i]);
       if(i == 0){
           pthread_create(&threadID[i],&threadAttributes[i], firstThread, ptr);
       }
       else{
           pthread_create(&threadID[i],&threadAttributes[i], otherThreads, ptr);
       }
   }

   sleep(5); //Give some time for the threads to be created and start running
thread_func

   int firstReturnValue = kill(getpid(), SIGINT);  //Send a SIGINT signal to the
process

   printf("Extra Stuff Here\n");
   for(int i = 0; i < 3; i ++){
       pthread_join(threadID[i], NULL);
   }
}
```

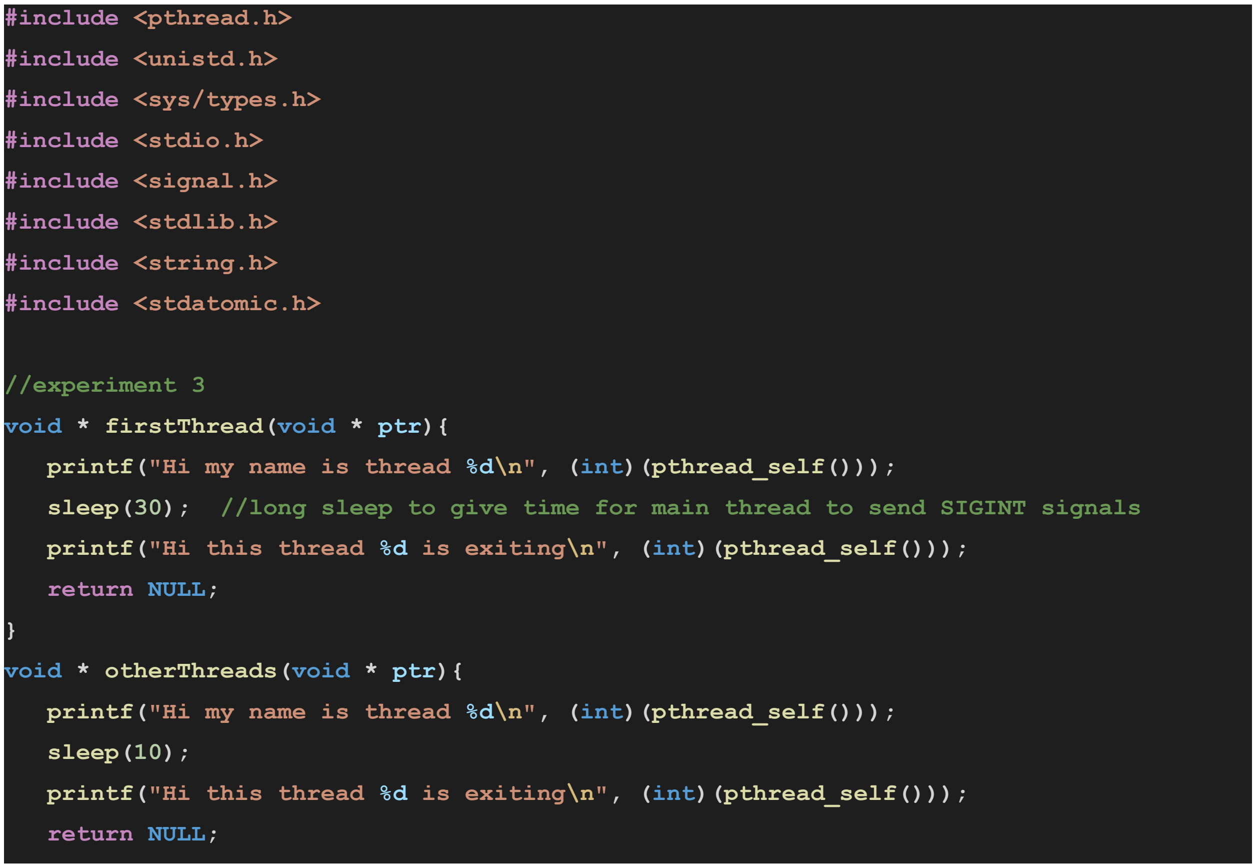
As shown in the Code, three threads were created and the SIGINT(i.e. A terminating signal was sent to only the first thread). However, none of the print statements in the main thread were executed after the kill system call was invoked. This tells me that the main thread was terminated. Furthermore, none of the child threads executed their statements after the sleep() system call. This also tells me that the child thread were terminated

Conclusion: When kill is used to send a terminating signal to the process, the entire thread group is taken down

<u>What happens when we send a terminating signal to the process when each thread has different signal handlers????</u>



```
rr1133@engsoft:~/pc/Homework4$ ./experiment3.out
The thread ID for the main thread is -838080704
Hi my name is thread -838084864
Hi my name is thread -846477568
Hi my name is thread -854870272
Signal 2 caught in this handler by thread -838080704
Extra Stuff Here
Hi this thread -846477568 is exiting
Hi this thread -854870272 is exiting
Hi this thread -838084864 is exiting
rr1133@engsoft:~/pc/Homework4$
```

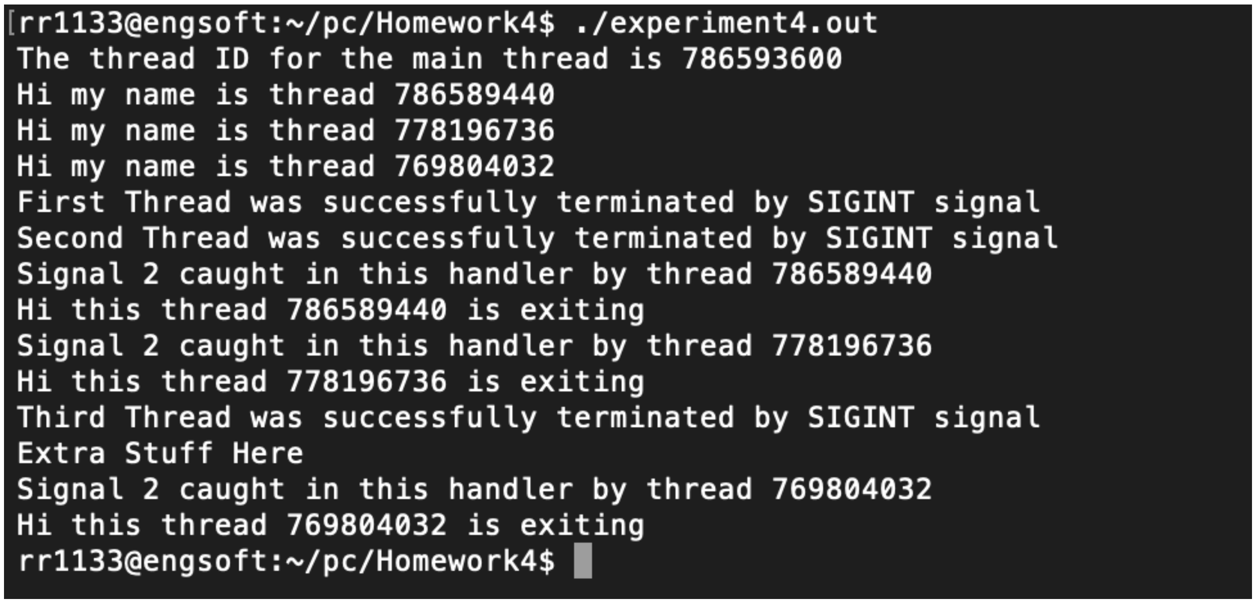


```
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdatomic.h>

//experiment 3
void * firstThread(void * ptr){
   printf("Hi my name is thread %d\n", (int)(pthread_self());
   sleep(30);  //long sleep to give time for main thread to send SIGINT signals
   printf("Hi this thread %d is exiting\n", (int)(pthread_self()));
   return NULL;
}
void * otherThreads(void * ptr){
   printf("Hi my name is thread %d\n", (int)(pthread_self());
   sleep(10);
   printf("Hi this thread %d is exiting\n", (int)(pthread_self()));
   return NULL;
}

void customHandler(int signum){
   printf("Signal %d caught in this handler by thread %d\n", signum,
(int)(pthread_self()));
}

int main(){

   pthread_t threadID[3];
   pthread_attr_t threadAttributes[3];

   void * ptr = NULL;
   printf("The thread ID for the main thread is %d\n",  (int)(pthread_self()));
   for(int i = 0; i < 3; i ++){
       pthread_attr_init(&threadAttributes[i]);
       if(i == 0){
           signal(SIGINT, SIG_IGN);
           pthread_create(&threadID[i],&threadAttributes[i], firstThread, ptr);
       }
       else if(i == 1){
           signal(SIGINT, SIG_DFL);
           pthread_create(&threadID[i],&threadAttributes[i], otherThreads, ptr);
       }
       else if(i == 2){
           signal(SIGINT, customHandler);
           pthread_create(&threadID[i],&threadAttributes[i], otherThreads, ptr);
       }
   }

   sleep(5); //Give some time for the threads to be created and start running
thread_func

   int firstReturnValue = kill(getpid(), SIGINT);  //Send a SIGINT signal to the
process

   printf("Extra Stuff Here\n");
   for(int i = 0; i < 3; i ++){
       pthread_join(threadID[i], NULL);
   }
}
```

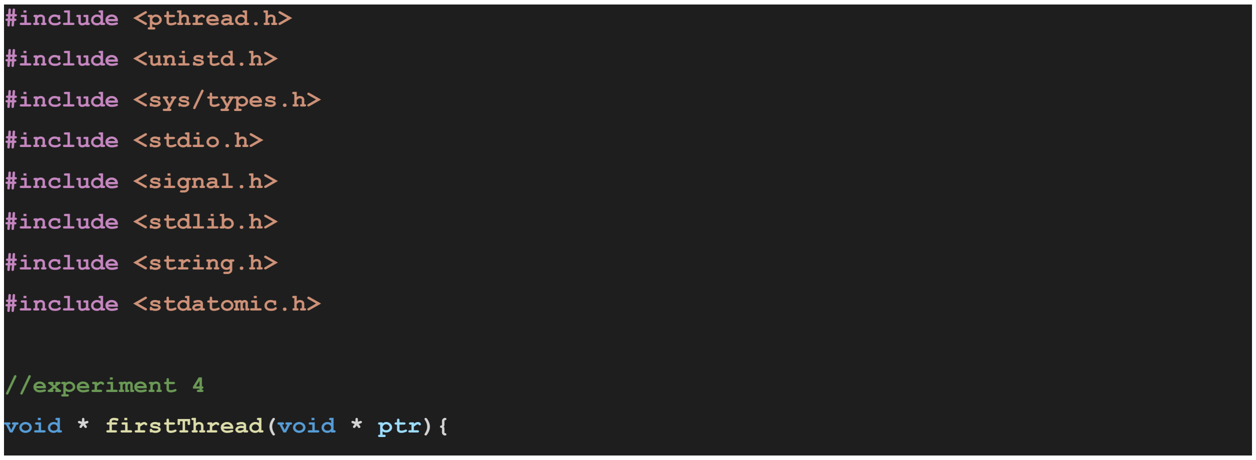
As shown in the Code, three threads were created with different signal handlers for the SIGINT signal. The SIGINT signal was sent to the entire process using the kill system call. As shown by the terminal output, the signal was caught by the main thread. Since, after the last for loop, the signal handler for SIGINT was established to be the custom handler, the main thread will use the custom handler to address the SIGINT signal.

Conclusion: When kill is used to send a terminating signal to the process, as discussed in lecture, one thread is chosen by the operating system to handle this signal. However, if this thread(in our case the main thread had the custom handler installed) has a custom handler, it will invoke the custom handler to handle the signal

<u>What happens when we send SIGINT signals to different threads using pthread_kill when we have defined signal handlers in our main thread(i.e. The process)?</u>



```
[rr1133@engsoft:~/pc/Homework4$ ./experiment4.out
The thread ID for the main thread is 786593600
Hi my name is thread 786589440
Hi my name is thread 778196736
Hi my name is thread 769804032
First Thread was successfully terminated by SIGINT signal
Second Thread was successfully terminated by SIGINT signal
Signal 2 caught in this handler by thread 786589440
Hi this thread 786589440 is exiting
Signal 2 caught in this handler by thread 778196736
Hi this thread 778196736 is exiting
Third Thread was successfully terminated by SIGINT signal
Extra Stuff Here
Signal 2 caught in this handler by thread 769804032
Hi this thread 769804032 is exiting
rr1133@engsoft:~/pc/Homework4$ 
```

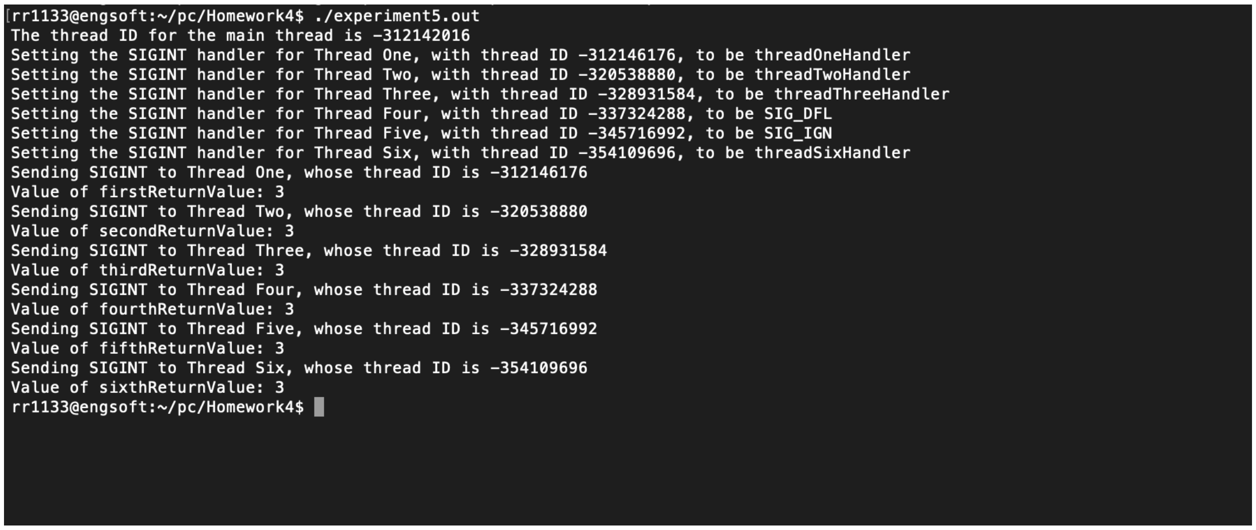


```
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdatomic.h>


//experiment 4
void * firstThread(void * ptr){
   printf("Hi my name is thread %d\n", (int)(pthread_self()));
   sleep(30);  //long sleep to give time for main thread to send SIGINT signals
   printf("Hi this thread %d is exiting\n", (int)(pthread_self()));
   return NULL;
}
void * otherThreads(void * ptr){
   printf("Hi my name is thread %d\n", (int)(pthread_self()));
   sleep(10);
   printf("Hi this thread %d is exiting\n", (int)(pthread_self()));
   return NULL;
}

void customHandler(int signum){
   printf("Signal %d caught in this handler by thread %d\n", signum,
(int)(pthread_self()));
}

int main(){

   pthread_t threadID[3];
   pthread_attr_t threadAttributes[3];

   void * ptr = NULL;
   printf("The thread ID for the main thread is %d\n",  (int)(pthread_self()));
   for(int i = 0; i < 3; i ++){
       pthread_attr_init(&threadAttributes[i]);
       if(i == 0){
           signal(SIGINT, SIG_IGN);
           pthread_create(&threadID[i],&threadAttributes[i], firstThread, ptr);
       }
       else if(i == 1){
           signal(SIGINT, SIG_DFL);
           pthread_create(&threadID[i],&threadAttributes[i], otherThreads, ptr);
       }
       else if(i == 2){
           signal(SIGINT, customHandler);
           pthread_create(&threadID[i],&threadAttributes[i], otherThreads, ptr);
       }
   }

   sleep(5); //Give some time for the threads to be created and start running
thread_func
```

```
   int firstReturnValue = pthread_kill(threadID[0], SIGINT);  //Send a SIGINT signal
to the first thread

   if(firstReturnValue == 0){
       printf("First Thread was successfully terminated by SIGINT signal\n");
   }

   int secondReturnValue = pthread_kill(threadID[1], SIGINT);  //Send a SIGINT signal
to the second thread

   if(secondReturnValue == 0){
       printf("Second Thread was successfully terminated by SIGINT signal\n");
   }

   int thirdReturnValue = pthread_kill(threadID[2], SIGINT);  //Send a SIGINT signal
to the third thread

   if(thirdReturnValue == 0){
       printf("Third Thread was successfully terminated by SIGINT signal\n");
   }

   printf("Extra Stuff Here\n");
   for(int i = 0; i < 3; i ++){
       pthread_join(threadID[i], NULL);
   }
}
```

In this program, it is clear that when we send the SIGINT signal to each thread, each thread is executing the custom handler that I have defined above in this program. A valid conclusion from this experiment is that it is quite evident that signal handlers remain constant across all threads of a process WHEN the signal function() is used in the main thread(i.e. For the whole process). Even when I have explicitly changed the signal handlers during each iteration of the for loop, it is evident that all three threads in our program are executing the custom handler. In class, we often assumed that threads only had visibility up until the point when we called pthread_create. However, this experiment clearly shows that this assumption is not valid for a real life operation system.

<u>What happens when we send SIGINT signals to different threads using pthread_kill when we have defined different signal handlers for each thread?</u>



```
rr1133@engsoft:~/pc/Homework4$ ./experiment5.out
The thread ID for the main thread is -312142016
Setting the SIGINT handler for Thread One, with thread ID -312146176, to be threadOneHandler
Setting the SIGINT handler for Thread Two, with thread ID -320538880, to be threadTwoHandler
Setting the SIGINT handler for Thread Three, with thread ID -328931584, to be threadThreeHandler
Setting the SIGINT handler for Thread Four, with thread ID -337324288, to be SIG_DFL
Setting the SIGINT handler for Thread Five, with thread ID -345716992, to be SIG_IGN
Setting the SIGINT handler for Thread Six, with thread ID -354109696, to be threadSixHandler
Sending SIGINT to Thread One, whose thread ID is -312146176
Value of firstReturnValue: 3
Sending SIGINT to Thread Two, whose thread ID is -320538880
Value of secondReturnValue: 3
Sending SIGINT to Thread Three, whose thread ID is -328931584
Value of thirdReturnValue: 3
Sending SIGINT to Thread Four, whose thread ID is -337324288
Value of fourthReturnValue: 3
Sending SIGINT to Thread Five, whose thread ID is -345716992
Value of fifthReturnValue: 3
Sending SIGINT to Thread Six, whose thread ID is -354109696
Value of sixthReturnValue: 3
rr1133@engsoft:~/pc/Homework4$
```

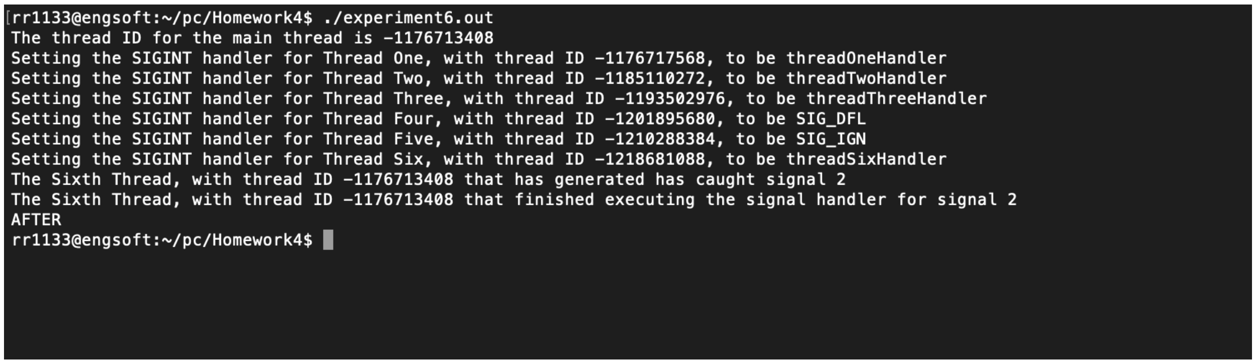
Code:

```
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdatomic.h>

// Experiment 5

void threadOneHandler(int signum){
   //sleep(1);
   printf("The First Thread, with thread ID %d that has generated has caught signal
%d\n", (int)(pthread_self()), signum);
   printf("The First Thread, with thread ID %d that finished executing the signal
handler for signal %d\n", (int)(pthread_self()), signum);
   //sleep(1);
}


void threadTwoHandler(int signum){
   //sleep(1);
   printf("The Second Thread, with thread ID %d that has generated has caught signal
%d\n", (int)(pthread_self()), signum);
```

```
   printf("The Second Thread, with thread ID %d that finished executing the signal
handler for signal %d\n", (int)(pthread_self()), signum);
   //sleep(1);
}


void threadThreeHandler(int signum){
   //sleep(1);
   printf("The Third Thread, with thread ID %d that has generated has caught signal
%d\n", (int)(pthread_self()), signum);
   printf("The Third Thread, with thread ID %d that finished executing the signal
handler for signal %d\n", (int)(pthread_self()), signum);
   //sleep(1);
}

void threadSixHandler(int signum){
   //sleep(1);
   printf("The Sixth Thread, with thread ID %d that has generated has caught signal
%d\n", (int)(pthread_self()), signum);
   printf("The Sixth Thread, with thread ID %d that finished executing the signal
handler for signal %d\n", (int)(pthread_self()), signum);
   //sleep(1);
}



void * threadOne(void * ptr){
   printf("Setting the SIGINT handler for Thread One, with thread ID %d, to be
threadOneHandler \n", (int)(pthread_self()));
   signal(SIGINT, threadOneHandler);
   return NULL;
}

void * threadTwo(void * ptr){
   printf("Setting the SIGINT handler for Thread Two, with thread ID %d, to be
threadTwoHandler \n", (int)(pthread_self()));
   signal(SIGINT, threadTwoHandler);
   return NULL;
}

void * threadThree(void * ptr){
   printf("Setting the SIGINT handler for Thread Three, with thread ID %d, to be
threadThreeHandler \n", (int)(pthread_self()));
   signal(SIGINT, threadThreeHandler);
   return NULL;
}

void * threadFour(void * ptr){
   printf("Setting the SIGINT handler for Thread Four, with thread ID %d, to be
SIG_DFL \n", (int)(pthread_self()));
   signal(SIGINT, SIG_DFL);
   return NULL;
}

void * threadFive(void * ptr){
   printf("Setting the SIGINT handler for Thread Five, with thread ID %d, to be
SIG_IGN \n", (int)(pthread_self()));
   signal(SIGINT, SIG_IGN);
   return NULL;
}


void * threadSix(void * ptr){
   printf("Setting the SIGINT handler for Thread Six, with thread ID %d, to be
threadSixHandler \n", (int)(pthread_self()));
   signal(SIGINT, threadSixHandler);
   return NULL;
}


int main(){

   pthread_t threadID[6];
   pthread_attr_t threadAttributes[6];

   void * ptr = NULL;
   printf("The thread ID for the main thread is %d\n",  (int)(pthread_self()));
   for(int i = 0; i < 6; i ++){
       pthread_attr_init(&threadAttributes[i]);
       if(i == 0){
           pthread_create(&threadID[i],&threadAttributes[i], threadOne, ptr);
       }
       else if(i == 1){
           pthread_create(&threadID[i],&threadAttributes[i], threadTwo, ptr);
       }
       else if(i == 2){
           pthread_create(&threadID[i],&threadAttributes[i], threadThree, ptr);
       }
       else if(i == 3){
           pthread_create(&threadID[i],&threadAttributes[i], threadFour, ptr);
       }
       else if(i == 4){
           pthread_create(&threadID[i],&threadAttributes[i], threadFive, ptr);
       }
       else if(i == 5){
           pthread_create(&threadID[i], &threadAttributes[i], threadSix, ptr);
       }
   }

   sleep(2);  //give the program enough time for all threads to execute their
respective functions
   printf("Sending SIGINT to Thread One, whose thread ID is %d\n",
(int)(threadID[0]));
   int firstReturnValue = pthread_kill(threadID[0], SIGINT);

   if(firstReturnValue == 0){
       printf("First Thread was successfully terminated by SIGINT signal\n");
   }
   else{
       printf("Value of firstReturnValue: %d\n", firstReturnValue);
   }


   printf("Sending SIGINT to Thread Two, whose thread ID is %d\n",
(int)(threadID[1]));
   int secondReturnValue = pthread_kill(threadID[1], SIGINT);

   if(secondReturnValue == 0){
       printf("First Thread was successfully terminated by SIGINT signal\n");
   }
   else{
       printf("Value of secondReturnValue: %d\n", secondReturnValue);
   }
   printf("Sending SIGINT to Thread Three, whose thread ID is %d\n",
(int)(threadID[2]));
   int thirdReturnValue = pthread_kill(threadID[2], SIGINT);

   if(thirdReturnValue == 0){
       printf("First Thread was successfully terminated by SIGINT signal\n");
   }
   else{
       printf("Value of thirdReturnValue: %d\n", thirdReturnValue);
   }

   printf("Sending SIGINT to Thread Four, whose thread ID is %d\n",
(int)(threadID[3]));
   int fourthReturnValue = pthread_kill(threadID[3], SIGINT);

   if(fourthReturnValue == 0){
       printf("First Thread was successfully terminated by SIGINT signal\n");
   }
   else{
       printf("Value of fourthReturnValue: %d\n", fourthReturnValue);
   }



   printf("Sending SIGINT to Thread Five, whose thread ID is %d\n",
(int)(threadID[4]));
   int fifthReturnValue = pthread_kill(threadID[4], SIGINT);

   if(fifthReturnValue == 0){
       printf("First Thread was successfully terminated by SIGINT signal\n");
   }
   else{
       printf("Value of fifthReturnValue: %d\n", fifthReturnValue);
   }

   printf("Sending SIGINT to Thread Six, whose thread ID is %d\n",
(int)(threadID[5]));
   int sixthReturnValue = pthread_kill(threadID[5], SIGINT);

   if(sixthReturnValue == 0){
       printf("First Thread was successfully terminated by SIGINT signal\n");
   }
   else{
       printf("Value of sixthReturnValue: %d\n", sixthReturnValue);
   }



}
```

In this experiment, I tried to use the signal system call to establish unique signal handlers for each thread in my program. As shown by my output, my operating system does NOT recognize this since NONE of the handlers that I defined were even called. Hence, since I never used signal() in the main thread, and none of the if statements executed in my main thread showed that the pthread_kill wasn't successful for any of the threads in my program. Since none of the pthread_kill statements returned 0(i.e. Indicating success) and they all returned 3, this led me to believe that using pthread_kill to send a signal to a thread and then using signal() within a thread to try to install a handler for a particular signal leads to undefined behavior.

<u>What happens if I send a signal to a process where each thread within its own code unique to that thread, has attempted to define a signal handler for itself</u>



```
rr1133@engsoft:~/pc/Homework4$ ./experiment6.out
The thread ID for the main thread is -1176713408
Setting the SIGINT handler for Thread One, with thread ID -1176717568, to be threadOneHandler
Setting the SIGINT handler for Thread Two, with thread ID -1185110272, to be threadTwoHandler
Setting the SIGINT handler for Thread Three, with thread ID -1193502976, to be threadThreeHandler
Setting the SIGINT handler for Thread Four, with thread ID -1201895680, to be SIG_DFL
Setting the SIGINT handler for Thread Five, with thread ID -1210288384, to be SIG_IGN
Setting the SIGINT handler for Thread Six, with thread ID -1218681088, to be threadSixHandler
The Sixth Thread, with thread ID -1176713408 that has generated has caught signal 2
The Sixth Thread, with thread ID -1176713408 that finished executing the signal handler for signal 2
AFTER
rr1133@engsoft:~/pc/Homework4$
```

Code:

```
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdatomic.h>
// Experiment 5
void threadOneHandler(int signum){
  //sleep(1);
  printf("The First Thread, with thread ID %d that has generated has caught signal
%d\n", (int)(pthread_self()), signum);
  printf("The First Thread, with thread ID %d that finished executing the signal
handler for signal %d\n", (int)(pthread_self()), signum);
  //sleep(1);
}
void threadTwoHandler(int signum){
  //sleep(1);
  printf("The Second Thread, with thread ID %d that has generated has caught signal
%d\n", (int)(pthread_self()), signum);
  printf("The Second Thread, with thread ID %d that finished executing the signal
handler for signal %d\n", (int)(pthread_self()), signum);
  //sleep(1);
}
void threadThreeHandler(int signum){
  //sleep(1);
  printf("The Third Thread, with thread ID %d that has generated has caught signal
%d\n", (int)(pthread_self()), signum);
  printf("The Third Thread, with thread ID %d that finished executing the signal
handler for signal %d\n", (int)(pthread_self()), signum);
  //sleep(1);
}
void threadSixHandler(int signum){
  //sleep(1);
  printf("The Sixth Thread, with thread ID %d that has generated has caught signal
%d\n", (int)(pthread_self()), signum);
  printf("The Sixth Thread, with thread ID %d that finished executing the signal
handler for signal %d\n", (int)(pthread_self()), signum);
  //sleep(1);
}
void * threadOne(void * ptr){
  printf("Setting the SIGINT handler for Thread One, with thread ID %d, to be
threadOneHandler \n", (int)(pthread_self()));
  signal(SIGINT, threadOneHandler);
  return NULL;
}
void * threadTwo(void * ptr){
  printf("Setting the SIGINT handler for Thread Two, with thread ID %d, to be
threadTwoHandler \n", (int)(pthread_self()));
  signal(SIGINT, threadTwoHandler);
  return NULL;
}
```

```
void * threadThree(void * ptr){
  printf("Setting the SIGINT handler for Thread Three, with thread ID %d, to be
threadThreeHandler \n", (int)(pthread_self()));
  signal(SIGINT, threadThreeHandler);
  return NULL;
}
void * threadFour(void * ptr){
  printf("Setting the SIGINT handler for Thread Four, with thread ID %d, to be SIG_DFL
\n", (int)(pthread_self()));
  signal(SIGINT, SIG_DFL);
  return NULL;
}
void * threadFive(void * ptr){
  printf("Setting the SIGINT handler for Thread Five, with thread ID %d, to be SIG_IGN
\n", (int)(pthread_self()));
  signal(SIGINT, SIG_IGN);
  return NULL;
}
void * threadSix(void * ptr){
  printf("Setting the SIGINT handler for Thread Six, with thread ID %d, to be
threadSixHandler \n", (int)(pthread_self()));
  signal(SIGINT, threadSixHandler);
  return NULL;
}
int main(){
  pthread_t threadID[6];
  pthread_attr_t threadAttributes[6];
  void * ptr = NULL;
  printf("The thread ID for the main thread is %d\n",  (int)(pthread_self()));
  for(int i = 0; i < 6; i ++){
      pthread_attr_init(&threadAttributes[i]);
      if(i == 0){
          pthread_create(&threadID[i],&threadAttributes[i], threadOne, ptr);
      }
      else if(i == 1){
          pthread_create(&threadID[i],&threadAttributes[i], threadTwo, ptr);
      }
      else if(i == 2){
          pthread_create(&threadID[i],&threadAttributes[i], threadThree, ptr);
      }
      else if(i == 3){
          pthread_create(&threadID[i],&threadAttributes[i], threadFour, ptr);
      }
      else if(i == 4){
          pthread_create(&threadID[i],&threadAttributes[i], threadFive, ptr);
      }
      else if(i == 5){
          pthread_create(&threadID[i], &threadAttributes[i], threadSix, ptr);
      }
  }
  sleep(2);  //give the program enough time for all threads to execute their
respective functions
  kill(getpid(), SIGINT);

   printf("AFTER\n");
}
```

In this program, since the main thread is able to get to the print statement “AFTER”, it is clear that sending SIGINT to the process will invoke an arbitrary thread to handle that signal. My theory is that based on the order in which we call pthread_create, the signal() functions defined in those respective threads will be executed in that order. Hence, the last thread that called pthread_create, the signal() function that it called will be the one that will define the handler for the main thread. So, for example, in my output, the handler that was defined in the sixth thread was used. This implies to me that the sixth thread was the one that was created last. Hence, within the function for the 6th thread, signal(SIGINT, threadSixHandler) was called last and threadSixHandler was the handler associated with the SIGINT signal for this entire process.