HOMEWORK 2
Omar Atieh (170001765)

1)

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

void * firstprime(void * number); // first thread
void * rev(void * number); // second thread
int main(int num, char * array[]) {

 if (num < 2) {
   fprintf(stderr, "Input must be greater than 2 test code like this ./filename Number \n");
   exit(1);
 }
 pthread_t p;
 pthread_t p2;
 pthread_attr_t ptre;
 pthread_attr_init( & ptre);
 printf("\nthese are all the numbers under\n");
 pthread_create( & p, & ptre, firstprime, array[1]); //thread 1
 pthread_join(p, NULL);
 printf("\nThe new primes after reversing \n");
 pthread_create( & p2, & ptre, rev, array[1]); //thread 2
 pthread_join(p2, NULL);
 printf("\nDone\n");
}

//thread 2 reversing prime and checking for prime status

void * rev(void * number) {
 int inputtedNum = atoi(number);
 int i;
 int j;
 int tem = 0;
 for (i = 2; i < inputtedNum; i++) {

   int inputtedNum = atoi(number);
   int * primer = malloc(sizeof(int) * inputtedNum);
   for (i = 2; i < inputtedNum; i++)
     primer[i] = 1;
   for (i = 2; i < inputtedNum; i++)
     if (primer[i])
       for (j = i; i * j < inputtedNum; j++)
         primer[i * j] = 0;
   int arr[] = {};

   for (i = 2; i < inputtedNum; i++)
     if (primer[i] == 1) {
       arr[tem] = i;
       int temp = reverseDigits(arr[tem]);
       int temp2 = revPrime(temp);
     }
   return 0;
 }
```

```
 pthread_exit(0);
}

//thread 1
void * firstprime(void * number) {
 int i, j, inputtedNum = atoi(number);

 //checking prime here
 for (i = 2; i < inputtedNum; i++) {
   int i = 2;
   int j;
   int inputtedNum = atoi(number);
   int * primer = malloc(sizeof(int) * inputtedNum);
   while( i < inputtedNum){
     primer[i] = 1;
     i++;}
   for (i = 2; i < inputtedNum; i++)
     if (primer[i])
       for (j = i; i * j < inputtedNum; j++)
         primer[i * j] = 0;
   int arr[] = {};
   for (i = 2; i < inputtedNum; i++)
     if (primer[i])
       printf("%d ", i);
   return 0;
 }
 pthread_exit(0);
}

//checking if it reverse digits are prime
int revPrime(int num) {
 int hi=2;
 int count =0;
 while (hi < num) {
   if (num % hi == 0) {
     count = 1;
     break;
   }
   hi++;
 }
 if (count == 0) {
   printf("%d ", num);
 }
}

//reverse the digits
int reverseDigits(int num) {
 int rev_num = 0;
 while (num > 0) {
   rev_num = rev_num * 10 + num % 10;
   num = num / 10;
 }
 return rev_num;
}
```

To run the code you write gcc -pthread -o problem1 problem1.c

Then ./problem1 “number”

As such this these are my results

```
omar@omar-PC:~/Desktop$ ./Problem1 88

these are all the numbers under
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
The new primes after reversing
2 3 5 7 11 31 71 13 73 17 37 97
Done
```

In this program two threads are created. The first thread finds out all the prime numbers under whatever number the user inputted.The second thread will run the reverse of the primes and reverse the digits. So in this example, I put 88 and got all the primes under 88 and after reversing I get the numbers underneath.

2)

Code version where parent thread waits for child to finish

Compile with gcc fib.c -o fib -pthread

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

int* output;

/*calculates fibonacci sequence
 * function input and return
 * matches pthread_create*/
void* fibonacci(void* input){

    int num = *(int*)input;    //casting as int pointer and derefrencing

    int prev = 0;
    int curr = 1;

    if(num<1){
         printf("Error Enter number greater than 1\n");
         exit(0);
    }

    else if(num == 1){

         output = malloc(sizeof(int));
         output[0] = prev;
         return 0;
    }

    else if(num == 2){
         output = malloc(sizeof(int));
         output[0] = prev;
         output[1] = curr;
```

```
          return 0;
    }


    output = malloc(sizeof(int)*num);
    output[0]=prev;
    output[1]=curr;

    int i,temp;
    for(i = 2; i<num; i++){

          temp = curr;
          curr = curr + prev;
          prev = temp;
          output[i] = curr;
    }



}


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

    int input = atoi(argv[1]);    //argument to int


    pthread_t id;
    pthread_create(&id, NULL, fibonacci, &input);

    pthread_join(id, NULL);    //parent thread waits for child thread to finish

    int i;
    for(i=0; i<input; i++){
          printf("Output: %d \n",output[i]);    //answer stored in output
    }
}
```

Code version where the parent thread does not wait for child thread to terminate. Parent thread prints out sequence while it's being computed by child thread. I used pipes to accomplish this. The parent blocks till a new sequence number is written to the pipe by the child thread.
Compile with: gcc fib2.c -o fib2 -pthread

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>


int p[2];

/*calculates fibonacci sequence
 * function input and return
```

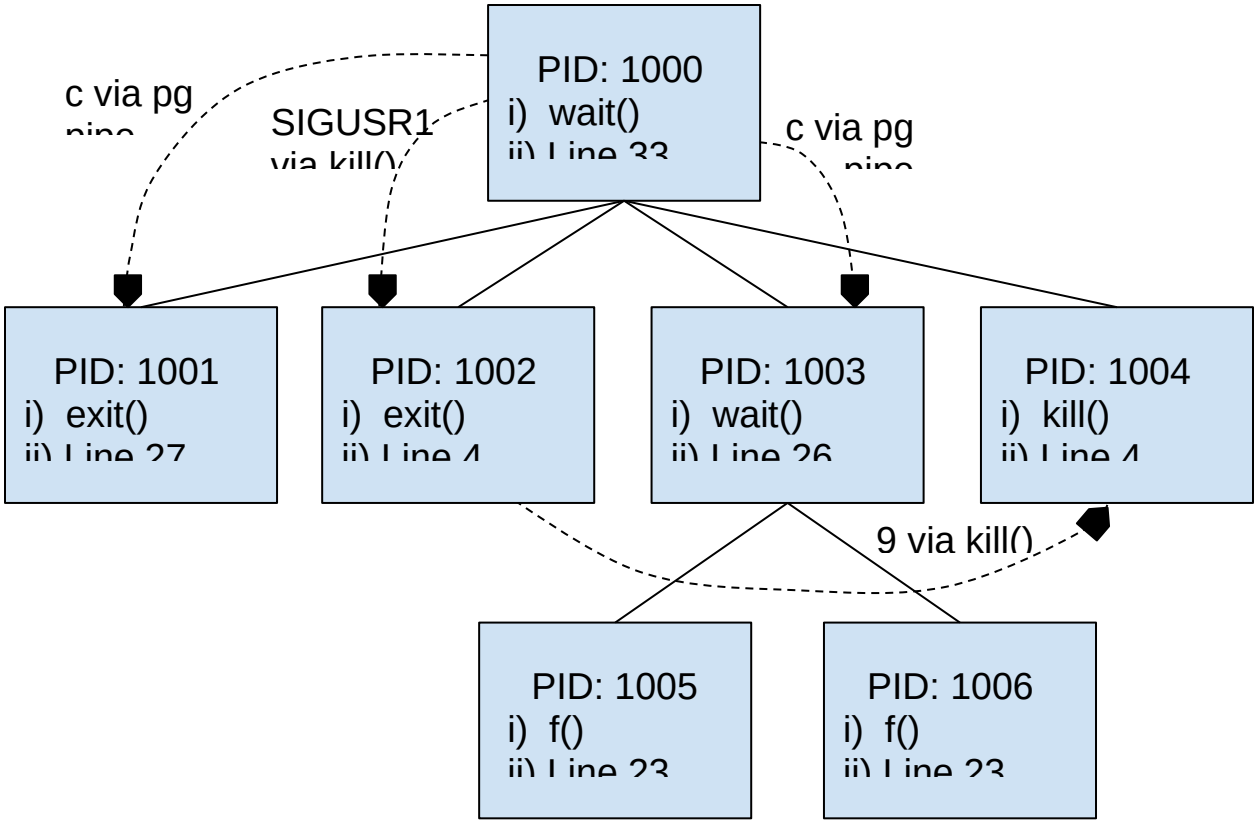
```c
 * matches pthread_create*/
void* fibonacci(void* input){

    int num = *(int*)input;    //casting as int pointer and derefrencing

    int prev = 0;
    int curr = 1;

    if(num<1){
         printf("Error Enter number greater than 1\n");
         exit(0);
    }

    else if(num == 1){
         write(p[1], &prev, sizeof(int));
         return 0;
    }

    else if(num == 2){
         write(p[1], &prev, sizeof(int));
         write(p[1], &curr, sizeof(int));
         return 0;
    }


    write(p[1], &prev, sizeof(int));
    write(p[1], &curr, sizeof(int));

    int i,temp;
    for(i = 2; i<num; i++){

         temp = curr;
         curr = curr + prev;
         prev = temp;
         write(p[1], &curr, sizeof(int));
    }

}


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

    int input = atoi(argv[1]);    //argument to int

    pipe(p);

    pthread_t id;
    pthread_create(&id, NULL, fibonacci, &input);


    int i,out;
    for(i=0; i<input; i++){
         read(p[0],&out, sizeof(int));
         printf("Output: %d \n",out);
    }
}
```

3)

Part 1

1. The argument “arg” contains the number of bytes to be read from the desired file. After the function is run, the variable *n* will have the value of -1 if an error occurs. When there is no error, *n* is the number of bytes read, and the file position is advanced by this number.
2. *wait(&status)* suspends the execution of the current process until the desired child finishes and terminates, or a signal is received. After this, the original process will continue to execute. This will return with the value -1 if the process that calls *wait* has no children, and with the process ID of the child that terminated when it runs successfully. Based on the output values of *wp*, WIFSIGNALED and WTERMSIG, the child process *wp* finished because it was interrupted by a signal, and according to the output value of WTERMSIG, this signal was SIGQUIT. This signal means the child process was quit directly by the user, using the quit key on the keyboard. This is how process *wp*=618 was ended. To accomplish this, process 412 used the *sigaction()* function, where the first parameter was the specified signal.
3. If O_NONBLOCK is set to 1, *read()* returns -1 and sets errno to EAGAIN. If O_NONBLOCK and O_NDELAY are set to 0, *read()* does not return anything(it blocks) until some data is written, or the pipe is closed by all other processes that have the pipe open for writing. If O_NDELAY is set to 1, *read()* returns 0.
4. If you call *read* after the last writer has exited, this means the write end of the pipe is now closed. As such, this *read* call will return 0. If you call *write* after the last reader has exited, this means the read end is now closed. As such, this *write* call will fail, and the process will be sent the SIGPIPE signal. By default, this handler terminates.

Part 2



The root process (PID: 1000) is created when the code commences. First, when this process gets to line 12, the signal SIGUSR1 becomes associated with the void function handler() written in lines 3-4. When the code reaches the first for loop at line 15, fork() is called on line 16 four times, thus creating four child processes (PIDs: 1001-1004). Afterwards, when the root process reaches line 31, the kill() function is called upon process pid[1] (PID: 1002), sending signal SIGUSR1 to said process. Next, the root process reaches another for loop on line 32. Like to previous loop, there are four iterations, one for each child processes. On line 34, the root process sends the value within c through a pipe, and the wait() function on the line prior tries to make sure the pipe is only written to and than read from once between the root and each of the four children. During the last iteration, the root process gets stuck in the wait() function on line 33 - this will be explained after the fourth child (PID: 1004) is discussed. Starting with the first child process (PID: 1001), it receives the value sent from the root process via the pg pipe on line 19. Following this, the process reaches two for loops, one on line 20, and the other on line 25. In each for loop, the integer j is set to 0, and they each share the same condition of j < i. As i is currently 0, the process just skips over these for loops and reaches line 27, where the exit() function is called. Next is the second child (PID: 1002). As explained earlier, the root process sent a SIGUSR1 signal to this process. The sleep() function on line 18 ensures the process won't proceed past this point prior to receiving the SIGUSR1 signal. As the root process assigned the handler() function to this signal, the second child goes to the handler() function. Before exit() is called on line 4, a kill() function is called on the process with PID: arg+2; arg = 1002 (the PID of child 2), thus the kill function is called upon PID: 1002+2 = 1004, i.e. the fourth

child process. The kill function sends signal 9, which immediately terminates said process. Going to the third child process (PID: 1003), it receives the value sent from the root process via the pg pipe on line 19. Next are the two for loops on lines 20 and 25. Each for loop is iterated two times, as i =2 and j is initialized to 0. In the first for loop, another fork() function is called upon, thus making the third child process a parent of two additional processes (PIDs: 1005, 1006). The second for loop has the new parent process (PID: 1003) wait for each child to finish at line 26, however both children call function f(), a function which never returns, thus leaving the two children to run indefinitely which in turn leaves their parent process waiting indefinitely. The fourth child process (PID: 1004) does start running when first created, however the scheduler about never gives the process enough time to create children on line 21 prior to process 1002 killing it. Because of the kill 9 signal sent to the fourth child process, the root process becomes stuck on line 33, as it indefinitely waits for a process which was abruptly terminated.

4)

PART 1

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


pthread_t tid1, tid2, tid3;

void sig_handle(int val){    //handler to handle signals in thread cannot handle sigstop
    printf("signal:%d  thread_id of handler:%lu \n",val,pthread_self());
}


void* sum_id_times_thousand(void* input){

    printf("child pthread tid: %lu \n", pthread_self());
    signal(*(int*)input, sig_handle);


    unsigned long long tid = pthread_self();
    unsigned long long val = tid * 1000;
```

```
    unsigned long long i,sum = 0;
    for(i = 0; i< val; i++){
         //printf("sum: %lld thread_id: %lld \n", sum, val);
         //sleep(1);
         sum++;
    }
}



int main(){

    printf("main tid: %lu \n", pthread_self());

    int sigint = SIGINT;
    int sigstop = SIGSTOP;    //sigstop cannot be handled
    int sigill = SIGILL;

    pthread_create(&tid1, NULL, sum_id_times_thousand, &sigint);    //creating threads
    pthread_create(&tid2, NULL, sum_id_times_thousand, &sigstop);
    pthread_create(&tid3, NULL, sum_id_times_thousand, &sigill);


    pthread_join(tid1, NULL); //joining all threads
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);


    signal(SIGINT, SIG_DFL); //SIGDFL default handler
    signal(SIGSTOP, SIG_DFL);
    signal(SIGILL, SIG_DFL);
}
```

Instead of using complete pseudocode, I have written the code in C syntax so it is easier to follow.

Main Thread:

In main I declared three ints that store the value of signals we want to handle: SIGINT, SIGSTOP, SIGILL. However **SIGSTOP cannot be handled** but I have included it anyways. Whenever SIGSTOP is received it will pause execution till a SIGCONT is received.

Then I create three pthreads with the function we want to run, as described by the description. Also we add the signal we want that thread to handle into the arguments of the pthread.

After this the main thread waits for the other three threads to complete and join using pthread_join. Than we set the signals back to the default handler.

Pthread:

The main thread creates pthreads with the signal value the thread should handle in a custom way.

In the pthread we call signal giving it the input of the signal it should handle and the custom handler we made(sig_handle(int val)).

Then the thread does the operation as described in the instructions. This operations takes a long time to complete, but I guess that's the point so we can properly test to make sure the handler works.

How It Should Work:

While all the three pthreads are running it should handle the specified signals if directed at the running thread.

Notes on Compiling and Running

So this code may compile and run as it does on my personal installation of Xubuntu but there are some issues. For some reason the signal handling by the custom handler I created is done in the main thread and not the pthreads. Whenever I send a signal through the terminal to the running program it uses the custom handler but it's handled by the main thread and not the pthreads. Prof Striki and I tried troubleshooting this code but it doesn't seem to work exactly as expected. But since Professor mentioned to me that we don't have to compile and run the code for this problem I am going to leave it as that. I included above the expected behavior of what should happen. **New Update**: After further testing I've come to believe that the code is probably working as expected. The signal(int, signal_handler) function registers a new handler per process and not per individual thread as I originally believed. Below I included some changes of what I believe to be equivalent code …

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


pthread_t tid1, tid2, tid3;

void sig_handle(int val){    //handler to handle signals in thread cannot handle sigstop
    printf("signal:%d  thread_id of handler:%lu \n",val,pthread_self());
}


void* sum_id_times_thousand(void* input){

    printf("child pthread tid: %lu \n", pthread_self());
    signal(*(int*)input, sig_handle);
```

```
    unsigned long long tid = pthread_self();
    unsigned long long val = tid * 1000;


    unsigned long long i,sum = 0;
    for(i = 0; i< val; i++){
         //printf("sum: %lld thread_id: %lld \n", sum, val);
         //sleep(1);
         sum++;
    }
}



int main(){

    printf("main tid: %lu \n", pthread_self());

    int sigint = SIGINT;
    int sigstop = SIGSTOP;    //sigstop cannot be handled
    int sigill = SIGILL;

     signal(SIGINT, sig_handle);
     signal(SIGSTOP, sig_handle); //SIGSTOP cannot be handled
     signal(SIGILL, sig_handle);

    pthread_create(&tid1, NULL, sum_id_times_thousand, &sigint);    //creating threads
    pthread_create(&tid2, NULL, sum_id_times_thousand, &sigstop);
```

In the following question I used the old code but it should still work the same. In the old code I used sleep as a quick fix to any possible synchronization issues.

PART2

Q1

After doing extensive testing with the code above, I believe when a program(process) receives a signal sent from the terminal it is handled by any running thread usually the main thread. And it should use the custom handler since I believe it, overriding the default handler, applies to the entire process and not just a single thread. SIGINT, SIGILL will use the custom handler but SIGSTOP cannot be handled with a custom handler, it will pause execution.

Q2

I simulated a thread raising a signal to the whole process like this…

```
void sig_handle(int val){    //handler to handle signals in thread cannot handle sigstop
    printf("signal:%d  thread_id of handler:%lu \n",val,pthread_self());

    if(val == SIGINT)
         raise(SIGILL);
```

```
}
```

Output:

```
vinay@Vinay-VirtualBox:~/Desktop$ gcc prob4.c -o prob4 -pthread
vinay@Vinay-VirtualBox:~/Desktop$ ./prob4
main tid: 3086366080
child pthread tid: 3075808064
child pthread tid: 3067415360
child pthread tid: 3084200768
^Csignal:2  thread_id of handler:3086366080
signal:4  thread_id of handler:3086366080
```

Whenever SIGINT is raised, it will also raise SIGILL. What I noticed is that SIGINT and SIGILL is being handled by the same thread which is the main thread. Since SIGSTOP cannot be handled it will just stop execution if that signal is raised

I can also have a thread direct a signal to another thread…

```
sleep(3);
pthread_kill(tid1, SIGINT);
```

This is added to the main thread, after the pthreads are created, which directs SIGINT to the thread than handles SIGINT and as expected it handles it like it has been coded. Here is the output…

```
prob4.c: In function ‘main’:
prob4.c:48:2: warning: implicit declaration of function ‘sleep’ [-Wimplicit-func
tion-declaration]
  sleep(3);
  ^~~~~
vinay@Vinay-VirtualBox:~/Desktop$ ./prob4
main tid: 3086275968
child pthread tid: 3067325248
child pthread tid: 3075717952
child pthread tid: 3084110656
signal:2  thread_id of handler:3084110656
```

It prints out the value of SIGINT and the thread id that handles it(last line). The handling thread is one of the pthreads that were created. The reason I added sleep so we don’t encounter any synchronization issues and the warnings are just from not including.

If I send SIGINT to a thread that only handles SIGILL using…

```
sleep(3);
pthread_kill(tid3, SIGINT);
```

It still uses the custom handler even though I registered the signal handler in thread 1. This

confirms what I believed that signal handling with a custom handler occurs per process not just per process.

And if SIGSTOP is directed at tid2 it cannot be handled and will stop execution.