Ravi Raghavan: Problem 2C
Preston Stecklein: Problem 1, help with 4A
Parth Darji: Problem 4A
Milan Trivedi: Prob 1 part 2
Anthony Poppalardo: Problem 4A Part 2 and Part 3
Rohan Rahalkar: Problem 4A Part 3
Jay Rajan: Problem 4A part 1
Rahul Umrani: 4A Part 3

Problem 1:
Q1)

Multithreading bests single threading:

Example 1: Displaying an image, we can really speed up the process here when updating a display by using

Example 2: Back propagation for neural networks. This is a much more complicated task, but since we are performing an algorithm called gradient descent on each node of the neural network, we can definitely benefit from using multiple threads here.

Single threading beats multithreading:

Example 1: Binary search algorithm, because each step relies on the previous, it would be a better use of resources and complexity to use a single thread for this task.

Example 2: Another example of when it may be worth using a single thread rather than multithreading is a task is any kind of basic operation, let's say finding the average of 10 numbers. The cost of creating multiple threads here is not worth it in terms of performance.

Using multiple kernel threads provides better performance when we have N work items rather than a single work item. Another circumstance is when the OS supports more than one kernel thread per process, because we can execute different user threads on certain kernel threads. On a single processor system, we cannot achieve the parallelism that we can when using a multithreaded solution.

Q2)

1. The scheduler will not be able to map some of the user level processes to the kernel threads, which means that some of the user level processes will remain idle. This means

that the process will have to take more time because there will be a backlog in user level processes. Additionally, this means that there will be processing cores that are not utilized. If the processing cores are all not utilized then the process will take more time to complete.

2. If the kernel threads are equal to the number of processing cores, this means all of the cores will be utilized, ensuring that the processes finish as fast as they can. However, if a thread is stalled or blocked by a system call, then this can hold up the processing cores while the process waits, which can slow down performance.
3. Although all of the cores will be utilized, the kernel threads may have to context switch which comes with an overhead. The current thread's state will have to be saved and then subsequently loaded with every switch, which takes a substantial amount of time. This will slow down performance.

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/Problem2C.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/Problem2C_Backup.c

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.

Problem 2C Part 2(i.e. Where Parent Thread Has to Wait on Child Thread):
Github:
https://github.com/Ravi-Raghavan/CompSysHW4/blob/main/Problem2C_Alternate.c

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 writeDone would be 0 when the child thread is in the process of writing a data 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 writing to 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.

Problem 4A)

Part 1
Question 1:
Tid1 will execute the func function first. This is because lock is initially set to zero, and no errors/exceptions will be thrown during the execution of the func function.

tid2 thread will not finish execution.

When the second thread is created at address &tid2, it runs into an arithmetic exception because we are trying to divide by 0.
Program line where expectation occurs:
`x = (int)pthead_self()/((getpid()*(-1 + lock++));`
In the part where we have (-1 + lock++), this expression evaluates to 0 since lock was incremented to 1 by thread tid1. The expression -1+1 evaluates to 0, and we divide by 0, thus throwing an SIGFPE error. . We also increment the lock variable to 2.

Tid3 is then created and executed. However Tid1 is still the first to execute the function func()

Question 2:
Assuming tid 1 executes *func* first, then

When the tread tid1 executes func, it receives SIGSEGV after trying to dereference p. Because of this the tid1 tread will terminate. It is also important to know that tid2 will terminate before the line that sets x. This makes it so that it will not increment lock. Also both tid2 and tid3 will receive SIGSEGV when func is run, because of this they will both call sig_func when sig = 11. Because of this both will print “Caught signal no = 11”.

There are two possibilities when tid2 and tid3 executes the line x=(int)pthead_self()?(getpid()*(-1 + lock++)); For each case the line will execute because of how we have lock = 0 in the expression. After the line is run the tread will then go to sleep for 10 seconds then terminate. However, Because there is a division by zero the thread executes the line second will det a SIGFE. This makes 2 cases that we need to consider. The first one is when The thread tid 2 executes the line x=(int)pthead_self()?(getpid()*(-1 + lock++)); before the tread tid3. SInce tid 2 does not have a handler for SIGFPE it will terminate when the signal is received. The other scenario is when tid3 executes before tid2. Because tid 2 has sig_func as its handler for SIGFPE it will print out the statement “ Caught signal no= 8 ”.

Question 3:
Assuming tid 3 executes *func* first,

When tid 3 executes there will be a invalid memory access which will cause SIGSEGV to be sent to the tread. This causes it to print “ Caught signal no = 11\n”. Next x will be assigned with no errors, and lock will be incremented by 1. Tid3 will finish by sleeping for 10 seconds then it will terminate.

Again tid1 and tid 2 will execute func. Tid 1 will receive SIGSEGV and terminate since it does not have a handler. Because of this the x=(int)pthead_self()?(getpid()*(-1 + lock++)); line will never run then x quill increment by 1. Since Tid2 can revive the SIGSEGV signal, it will print “ Caught signal no = 11”. Then the line x=(int)pthead_self()?(getpid()*(-1 + lock++)); will cause

SIGFPE to send because there is a division by zero. Now tid2 does not have a handler and the signal will terminate the thread.

Part 2

1) Line C

   SIGSEVG is caused by a memory access violation. In our pseudo code we are given the line *pthread_kill(tid2, SIGSEVG);.* In this case, when the command in Line C is executed it causes the main to signal SIGSEVG to be thread two for the three iterations. The handler for this is sig_func which is bound to thread two.

2) Line E

   SIGINT is responsible for gating and intercepting the various signals encrypted across a A system. In our pseudo code we are given line E which states *pthread_kill(tid3, SIGINT)*. For this line when it is executed 3 times it is being sent to thread three, three times. For this scenario there are two areas in which it could be handled one is func2 or the other is SIG_IGN.

Part 3

1) If we were to use kill (pid,2); our signal two is SIGINT, it could be handled and pushed into the threads one and three respectively. It will most likely ignore func2, when or if it reaches the second thread or pid, it should terminate.

2) If we were to use kill (pid, 11); our signal 11 is SIGSEVG which as stated could be drawn into threads three and two and executing func.

3) Kill (pid, 8); is associated with the signal SIGFPE. In this case this signal is completely ignored and is not handled by anything so it does not change or affect it.

4) Kill (pid, 15); is associated with the signal SIGTERM. There is no handler which executes/takes care of this signal. Therefore, this will lead to a process termination.

4A Part 2

After testing the simple multi-threaded program, we send a SIGTERM signal to one of the process threads. In this case, not just the particular thread will be terminated. The whole process will be terminated when we send the terminating signal. The kill system call also kills the entire process whereas pthread_kill only kills the thread, becasue it is the specific thread that recives the signal

Set some of the threads using default handlers