OS Assignment 1

Anna Bergknut & Amanda Björk

Task 1

Implementation:

To produce only 3 forks the second fork is put in the if statement that the parent process will go into. To get the process id we chose to print the pid for the parent, this because the parent will receive the childs process id, the child will receive 0 if the fork was successful.

Questions:

Which process is the parent and which is the child process?

A is the parent and B is the child.

The variable i is used in both processes. Is the value of i in one process affected when the other process increments it? Why/Why not?

Here, global variable change in one process does not affect two other processes because the data/state of the two processes is different. And also parent and child run simultaneously so two outputs are possible.

Which are the process identities (pid) of the child processes?

For A it is 6837 and for B it is 6838

Task 2

Implementation:

Firstly we changed the buffer struct so it had an int array of size 10 and thre new variables were added to the struct. These were first, last and antal. Antal registered how many items where currently in the array. First registered the first item in the buffer and last registered where the next empty space in the buffer was. To make the buffer cirkuler three was a if statement for last so if the variable became 10 then it would become 0 instead and vice versa for first.

Task 3

Implementation:

To eliminate the race condition from task 2 we made two sephamores, one for an empty buffer and one for the full buffer. So the consumer will initiolize the empty semaphore at the start of its operation and the producer will unlock the empty semaphore when its operation is done. The same the same but reversed for full.

Questions:

Why did the problems in Task 2 occur, and how does your solution with semaphores solve them?

It will be race condition!! Semaphore means that only one fork can work with the moment it locks in at a time. It's like mutex_lock but for forks.

Task 4

Implementation:

To send an int the message struct needed to change. When the messenger does not wait for the user's input it will send its messages really fast without the receiver even ready. So we implemented a message receiver for the sender. When both processes are up and running the user can write “ready” and then the sender will start to send 50 numbers. After the sender has sent all 50 integers the sender will sleep before closing the connection. This is to ensure that the receiver has read all the messages.

Task 5

Questions:

What does the program print when you execute it?

It is a simple threads program. Where a "child" is created that has the start function child. There are two threads that work simultaneously. It prints “This is the parent (main) thread.” and “This is the child thread.”

Task 6

Implementation:

Depending on which argument you send in, the program will create as many threads as the user asks for. This is done by entering a number after calling the program. So if you write 3, the for loop that creates the threads will loop 3 times and then call the child function 3 times. Child prints “Greetings from child #0 of 3” only up to #2.

Questions:

Why do we need to create a new struct threadArgs for each thread we create?

To send in an argument from pthread_create it must be a void pointer; this means that the argument is not really executable, so the new threadArgs struct will only get the same address as the void pointer that was sent in. This is to be able to manage the arguments in the struct.

Task 7

Implementation:

We used the struct and array available as the bridge between the child and the parents. When the child did the square id it created it as a pointer to the strukt. We separated the final for loop to print the result from the parent with the for loop that join all the children and the parent so that we could get a clean print.

Task 8

Questions:

Does the program execute correctly? Why/why not?

Sometimes the program gets it right and returns 0 at the end. But this is not always the case. Many times they give either a negative or a positive bank balance, depending on the race condition. Since all threads can do exactly the same operations at the same time, they can get a little "confused" and think they've done their job even though it was another thread that did it. Then there is often too little work done on one end and does not give the right results and execution.

Task 9

Implementation:

We put mutex_lock and mutex_unlock before the transaction and after. This so that no more than 1 thread can read and edit the bankaccount.

Task 10

Implementation:

We followed a thread-like approach, breaking down tasks step by step. We created pseudo-code for functions, including main, and implemented them one at the time. For thread-related tasks, We set up professors and added mutex for chopsticks. Additionally, We incorporated an exit button function for code utility and ensured proper cleanup in the main.

Questions:

An implementation done in line with the description above is not deadlock-free,

Explain why? which conditions lead to the deadlock?

The deadlock occurs due to the combination of the "Hold and Wait" and "Circular Wait" conditions mainly but it has aspects from the other two too.

Each philosopher must acquire two chopsticks to eat. If a philosopher can only acquire one chopstick and is waiting for the other, they are holding onto a resource (chopstick) and waiting for another resource to become available. If all of them hold the left chopstick then none can get the second chopstick needed to eat and get out of the wait.

Task 11

Implementation:

We implemented that instead of thinking and holding onto a chopstick when unable to obtain the second one, the professor attempts to acquire it again and then gives up if unsuccessful.

Questions:

Which are the four conditions for deadlock to occur?

Mutual Exclusion: ensures that only one process can use a non-sharable resource at a time, preventing access by other processes until the resource is released.
Hold and Wait: involves a process holding at least one resource while waiting to acquire others held by different processes, potentially leading to deadlock as processes wait for each other to release resources.
No Preemption: means resources cannot be forcibly taken from a process; only the holding process can release them voluntarily, contributing to the deadlock scenario.
Circular Wait: occurs when there's a circular chain of processes, each waiting for a resource held by the next, creating a cycle of dependencies and the potential for deadlock.

Which of the four conditions did you remove in order to get a deadlock-free program, and how did you do that in your program?

"Hold and Wait" and "Circular Wait". If the professor is missing a chopstick he will think ones more and try again. If the chopstick is still not available he will put back the chopstick.

Task 12

Implementation:

We implemented the function time in the main program at the very beginning and before the return so that the whole executing time will be represented.

Questions:

How long time did it take to execute the program?

8 sec

Task 13

Implementation:

The parts of the code that was supposed to have multi threads was put into a function that wanted a void pointer as an argument. And only managed one row of the matrix. In the function mul_seq the threads and the arguments array are defined. The threads are initialised in that function and sent to the mul function where they manage one row each.

Questions:

How long time did it take to execute the program?

2 sec

Which speedup did you get?

Speedup = 8/2 = 4

Task 14

Implementation:

In a similar fashion as task 13 made for the mul_seq we did to the init_matrix function, by taking the multi threads part into a different function.

Questions:

Did the program run faster or slower now, and why?

It was faster with a speed of 1 sec. We parallel processed a second operation that can be very time consuming. This cut the executing time even more.

Task 15

Implementation:

The threads are initialised in the main function two times. First the threads are created to initialise the matrix. But then they are joined before being created again to make the multiplication. This is to avoid problems where only one part of the matrix is initialised when one thread is in the multiplication so it will be incorrectly calculated.
To divide the rows the arguments that the thread functions used had the first and last row represented.

Questions:

How long time did it take to execute the program?

2 sec

Which speedup did you get?

Speedup = 8/2 = 4

Do these execution times differ from those in Task 13? If so, why? If not, why?

The time doesn't differ from task 13. The threads work in different ways in the two solutions. In task 13 there are 1024 threads that work in parallel but only for one operation. When in task 15 there are fewer threads but they do more operations. This implementations may be indifferent when it comes to time for that reason.