Report Lab 1 by Vincent Honar & Jonatan Frykmer

1.

- **Which process is the parent, and which is the child process?**
  The child process is the process where pid is 0, i.e., the parent executes the "else" block. (B is the parent, A 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?**
  Each process has its own instance of the variable "i" in the state at which the fork was commenced. Neither "i" has any effect on the other because they do not occupy the same memory.
- **What was changed?**
  The code block which the parent process executes was modified to create a third child process. The parent process' code block was further broken down into an if else condition for the third child.
- **Which are the process identities (pid) of the child processes?**
  The child identities pid was 10991 & 10992 respectively when running the program once.

2.

- **What was changed?**
  The struct storing the value and its status was changed into containing two arrays. One of the arrays contained the various values and the other contained the status of the given position on the array, i.e., whether it was full or empty. A while loop (inside the original while loop) inside each process iterates through the "values" array looking for either an empty or full slot depending on said process. To achieve circularity, every element is accessed by "i%10". The aforementioned problems tend to occur right at the start of the program as the producer accesses the shared memory more frequently than the consumer which can lead to an abrupt stop mid-operation (a slot can be assigned as empty before properly read).

3.

- **Why did the problems in Task 2 occur, and how does your solution with semaphores solve them?**
  The problems in task two occurred due to the processes accessing/working with the shared memory simultaneously as one of the processes wasn't completely done with it. Semaphores allows us to hinder process A from accessing the shared memory whilst process B is in there and changing things.
- **Explanation of what was changed**
  By adding semaphores around the so-called critical region, we restrict the other process' access and thus reduce the risk of data being overridden or dropped.

4.

- **What was changed?**
  Mtext was changed into an int, user input was removed, a sleep of 7 seconds is put at the start of msgsend.c. A for-loop iterates 50 times, generating 50 random values and sending the values to the receiver. Once the for-loop has done 50 loops, it breaks and exits the sender. The receiver had a variable to end, this is now used to count the amount of received values. When 50 values have been received the program exits.

5.

- **What does the program print when you execute it?**
  The program prints: “This is the parent (main) thread.” And “This is the child thread.” In varying order.

6.

- **Why do we need to create a new struct threadArgs for each thread we create**
  The threads directly interact with the given parameters and change the values stored in said memoryspace of the treadArgs. If every thread worked with the same instance of threadArgs, they would just overwrite each other and create unpredictable results.

7.

- **What was changed?**
  A new attribute was added to the struct threadArgs called squaredId. In the child process squaredId is assigned the following value:
  $(childId + 1) \cdot (childId + 1)$ (the +1 is there to make it non-zero based)
  As each process joins back with the main thread squaredId is printed by the parent thread.

8.

- **Does the program execute correctly? Why/why not?**
  The program does not execute correctly all the time. The higher the number of threads, the more likely the balance will not be 0. Data races occur as each thread accesses balance whilst different threads are in different stages of reading, reassigning, saving the data in balance. This causes unpredictable results as the work of one thread can be completely nullified by another or distorted.

9.

- **What was changed?**
  A lock was placed inside the deposit and withdraw functions around where the values are changed. Once all threads have joined back up with the main thread, the lock is destroyed.

10.

- **See below.**
- **Creation summary**
  We initialised 5 threads representing each professor with their own id number. An array of the size 5 was also created to represent the forks and contain information whether a particular fork was taken or not. A professor’s right fork would have the professor’s id as index whilst the left fork would have professor id +1. We made use of if statement which looked at the status of a specific professor. Initially a professor would have the status “Thinking” meaning they could enter our first if statement and pick up a left fork if one was available. If it did so successfully the professor’s status would be changed stating that it picked up a left fork. The next if statement examines the status, if the professor has a left fork it will also attempt to pick up a right fork after waiting a while. To avoid deadlocks, if the professor is unable to pick up a right fork, they will throw away both their forks and go back to thinking. Once they have consumed their food, waiting an appropriate number of random seconds, both forks are put down and the said thread exits.

11.

- **Which four conditions lead to a deadlock and why?**

1. **Mutual exclusion**
   A resource can’t be used by more than one process at a time. With this condition resources are limited, and some processes may have to wait for another process to release a resource before they can attain one.

2. **Hold and wait**
   If a process does not release a shared resource and simply holds on to it and waits forever the resource can never be accessed by another program and thus causes a deadlock.
3. **Pre-emption**
   This means that a process can only give up a resource if it chooses to do so itself it can't be forced to do so by another process.
4. **Circular wait**
   This occurs when processes are forming a circle where one process waits for another to finish which in turn waits for another and so on.

- **What was changed?**
  We changed the Hold and wait condition to break the deadlock by making the professors drop their left fork if they were unable to find a right fork.

12.

- **How long time did it take to execute the program?**
  The fastest execution time was 7 seconds, whilst it could reach up to 20 seconds

13.

- **How long was the execution time of your parallel program?**
  The fastest execution time was 3 seconds whilst it could reach up to 11 seconds
- **Which speedup did you get (as compared to the execution time of the sequential version,where Speedup = $T_{sequential}/T_{parallel}$)?**
  2~4
- **What was changed?**
  The outermost for-loop in matmulseq was removed and moved into the main function as a thread initaliser. Another for-loop was added inside of the main function to allow the threads to join back once their work was finished. This was done to a copy of matmul_seq named matmul_para as the original was used in order to compare the two methods. A thread's id was sent to the loop as a parameter.

14.

- **How long was the execution time of your parallel program?**
  The program runs consistently faster, averaging 2 seconds runtime. The quotient is 2~5. Allowing multiple threads to concurrently work on a task of this size utilises the computer's resources more efficiently than having one thread do all the work.
- **What was changed?**
  The outermost for-loop in the matrix initialiser was moved to the main function as a thread initialiser. Another for-loop was also added inside main to let the threads join back.

15.

- **Which is the execution time and speedup for the application now?**
  The program runs consistently slightly below 2 seconds. This is considerably faster than task 13, most in part due the initialisation being parallelised as well. The marginal difference observed between the runtime of 14 and 15 could be because fewer threads have to act as delegators.
- **What was changed?**
  The old outer for-loops in the original file were reintroduced but with a variable limit and start depending on the thread-id + 1. A definition was added called "THREADS" in which the number of threads which we wish to initialise should be written.