**Assignment: Understanding Processes**

a)

**P0**

**P1**

**P3 P2**

**P7 P5 P6 P4**

b)

**P0**

**P1**

**P2**

\*P1 does not execute the second *fork()* because the if condition is false for it

c)

**P0**

**P1**

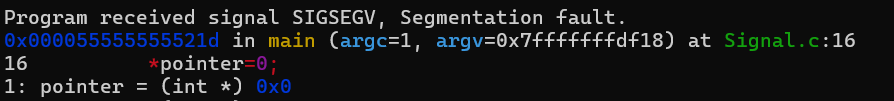
**P2**

**P3**

\*P1, P2, P3 exit the loop immidiately since the if condition is false, therefore they don’t fork further

**Assignment: Working with Signals**

The program has crashed because it initializes the pointer variable to *NULL* (***int \*pointer=(int \*) NULL;***), which is pointing to an invalid location, which cannot be accessed. When the program tries to assign 0 to the memory address that the pointer is pointing to, and since it attempts to access an invalid memory address, it leads to a segmentation fault.

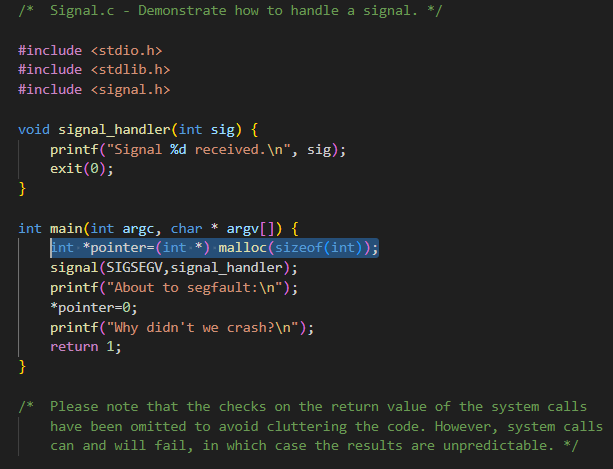


**Assignment: Modify Signal.c**

I had to use **malloc** to initialize the pointer. It allocates memory dinamically and returns a pointer to this part of memory. Since the pointer is declared as integer, I used **sizeof(int)** to make sure that it is enough space allocated: ***int \*pointer=(int \*) malloc(sizeof(int));***

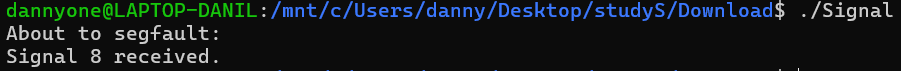
Now, when we dereference the pointer with **\*pointer = 0;** , we are writing to a valid memory location and therefore the program doesn’t crash.

The source code:

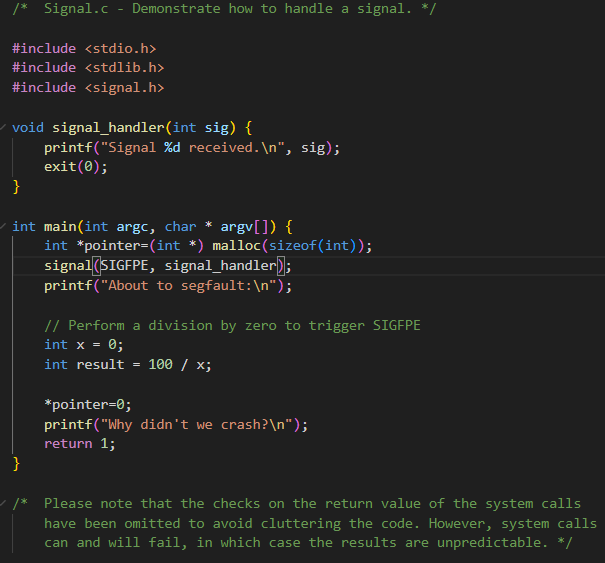


**Assignment: SIGSEGV**

I decided to replace SIGSEGV with **SIGFPE**, since this signal is raised when the program performs an illegal arithmetic operation, and that’s very easy to generate. So I have inserted a piece of code which divides a number by zero. After running the program I got the different signal:

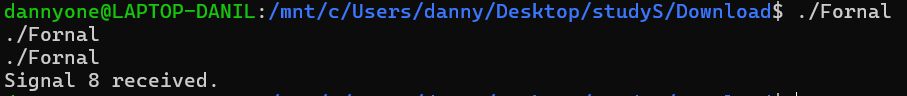


The source code:

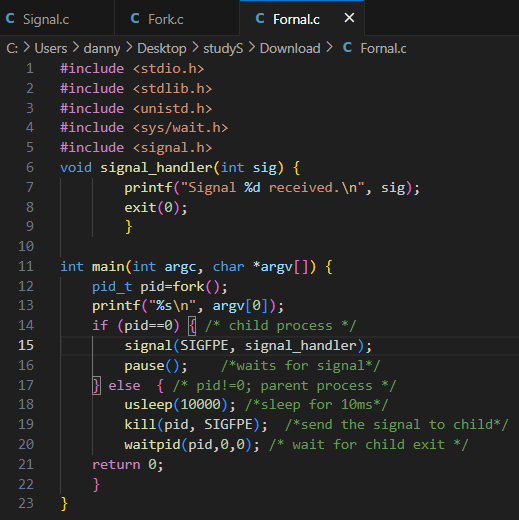


**Assignment: Trying it all Together**

To make the parent send the signal which is handled by child, I had to use **pause()** and **kill()**. However, **kill()** in parent should happen only after child executes **pause()**, otherwise child doesn’t work, for this I had to make the parent sleep for *10ms*, so I used **usleep(10000)** which makes the process sleep for 10000 microseconds (*10ms*) After that every signal that was send by the parent, was handled by the child:

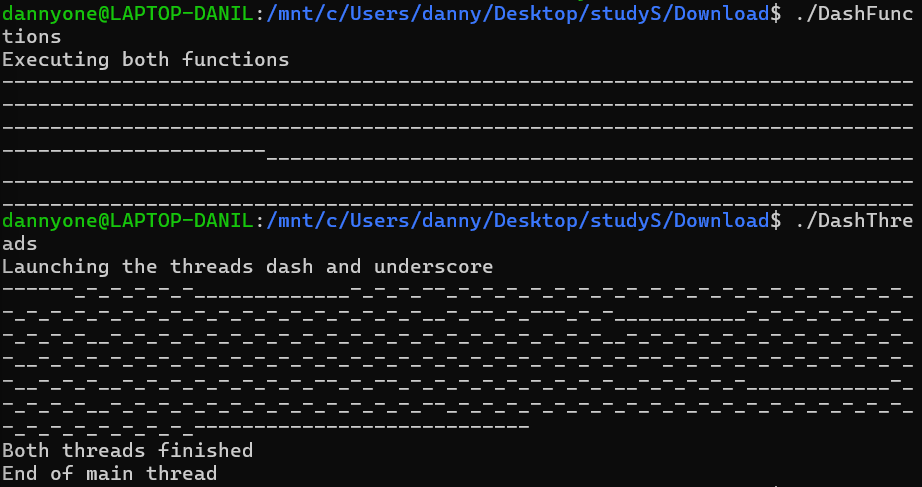


The source code:



**Assignment: Basics of Threads**

After running both programs, I got the following output:

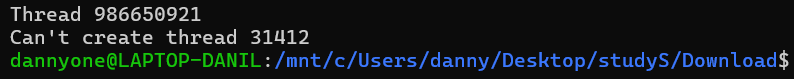


The key difference between the two programs is how they handle the execution of tasks: **sequentially** (*DashFunctions*) versus **concurrently** (*DashThreads*).

As we can see from the output, first program prints all the **“-”** and after that all the **“\_”**, while the second program alternates them. This happens because *DashFunctions* program uses **regular function calls**. When the first function is finished, then the next function starts. There is no overlap or concurrency. On the other hand, the *DashThreads* program uses **threads** which allow functions to run concurrently, meaning that both **“-”** and **“\_”** are printed at the same time, for this reason they are mixed in the output.

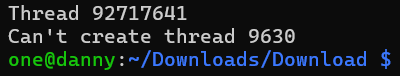
**Assignment: Working with Threads**

After changing the N several times, I got the following message.



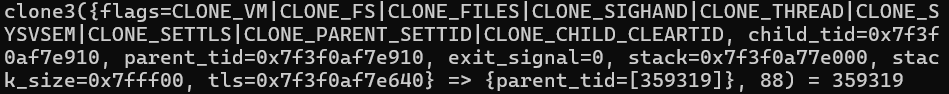
From that I can easily conlude, that limit for my laptop is **31412** threads.

On the PI I got the following output (**9630** threads):



Obviously, my Laptop can handle more threads then my PI.

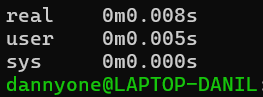
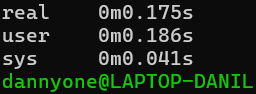
The system call responsible for creating threads is ***clone3***. This system call is used to create new processes by duplicating the calling process's memory space, file descriptors, and other attributes:



**Assignment: MyThread Timing**

After recompiling the **C** code, compiling **Java** code and measuring their execution times, I got the following result:

**C code Java code**

As we can see from the result the **C** code is much faster (the Java version took **0.167** seconds longer) The main reason for such a big gap between these to programs is *bytecode interpretation*. **Java** programs are compiled to bytecode, which is then interpreted by the *Java Virtual Machine* (*JVM*). **C**, on the other hand, is compiled directly to machine code, which is executed directly by the operating system, making it much faster. Moreover, *JVM*, needs to initialize and load classes, which takes time compared to native execution in **C**.

**Assignment: Power of Threads**

**Assignment: Parallel shell**

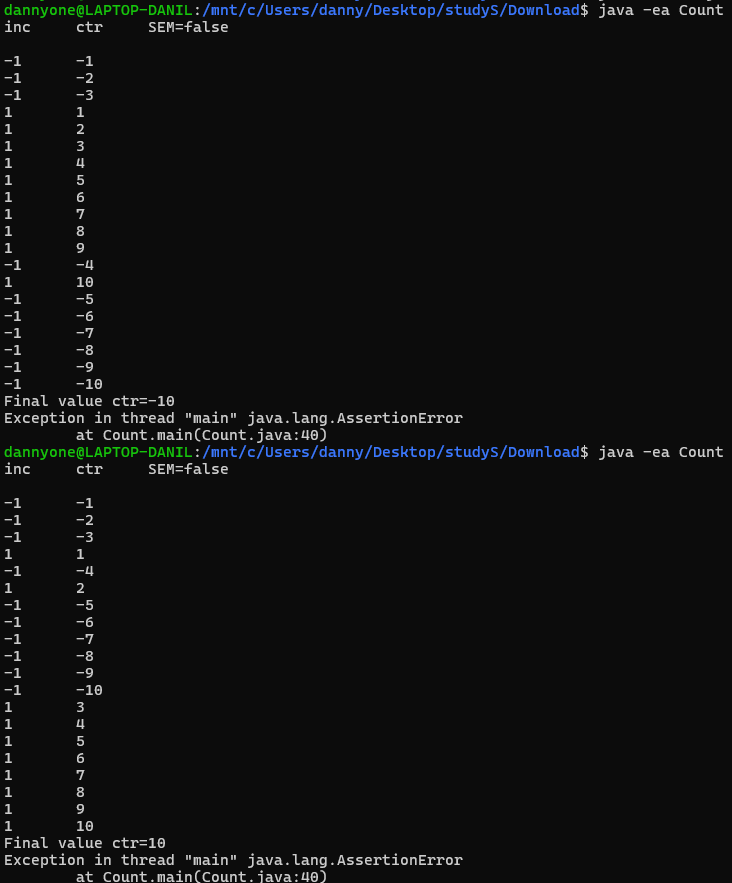
*cat index.html | sort | uniq -c | sort -rn | pr -2*

1. ***cat index.html:*** Reads the file sequentially and outputs the contents to STDOUT.
2. ***sort:*** Alphabetically sorts lines.
3. ***uniq -c:*** Counts occurrences of each unique line.
4. ***sort -rn:*** Sorts by frequency in descending numerical order. (the **-rn** flags mean "reverse" and "numerical")
5. ***pr -2:*** Formats the output into two columns.

The ***sort*** stages could benefit most from concurrency, since many sorting algorithms can be parallelized by dividing the input into parts, sorting each concurrently, and then merging the sorted parts. Other stages have a minimum benefit from cocurrency possible.

**Assignment: Count.java**

After running the program several times, I got different results as you can see below:

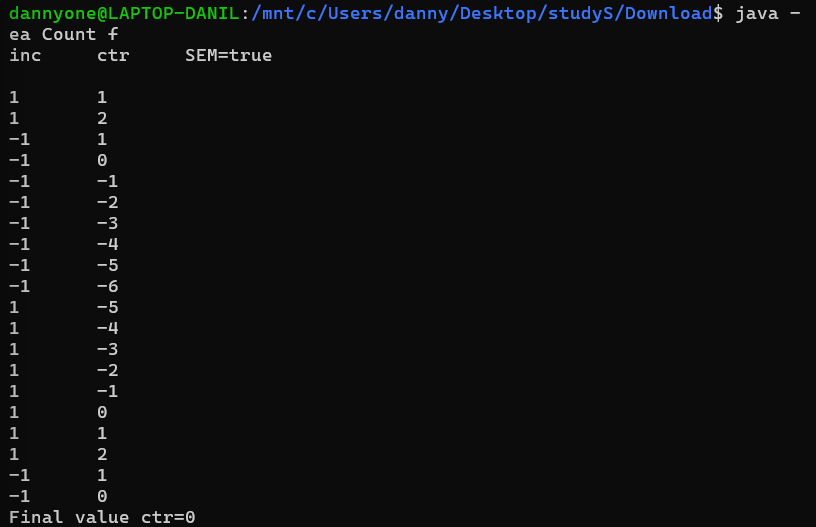


This is because there's no synchronization between the threads, since the **semaphore** **is** **false**: 

For this reason, both threads (one incrementing and the other decrementing the counter) access and modify the ***ctr*** variable at the same time, leading to an exception.

**Assignment: Count with Semaphore**

Yes, if we set the semaphore to **true**, every run ends up with the same final result (***ctr=0***):



Now threads don’t modify the variable at the same time, one thread adds 1 to ***ctr***, then the other subtracts, leading to ***ctr=0*** , meaning that they don't interfere each other. Moreover, concurrency is still possible, multiple threads are still running simultaneously, but it is now controlled and properly managed.

**Assignment: Count in C and Java**

Both, *Count.c* and *Count.java* produce the same result. Both of them end up with the **ctr** equal to 0. Even though programs use different tools, they have the same functionality and similar outputs:

Output of the *Count.c* program:

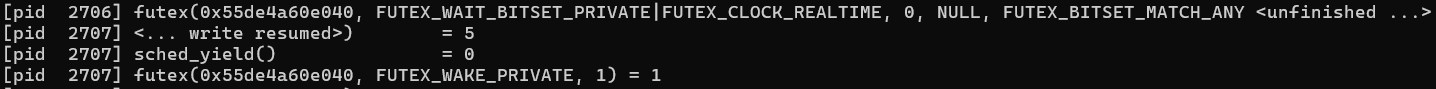


**Assignment: Count and strace**

The system calls involved in the implementation of a semaphore in your program are the **futex** (fast userspace mutex) system calls. They consist of two operations:

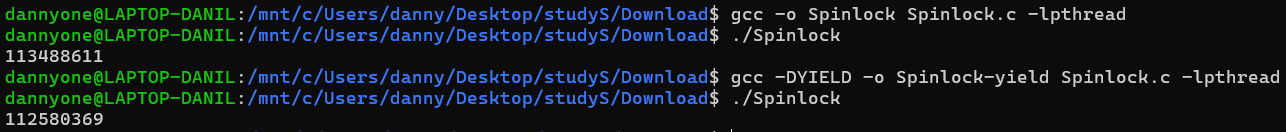
* The **FUTEX\_WAIT** operation makes a thread wait until the futex value changes, blocking the thread if it can't acquire the semaphore.
* The **FUTEX\_WAKE** operation wakes up threads waiting on the futex, allowing the next thread to proceed.

Here are some examples of futex in the output.



**Assignment: Spinlock**

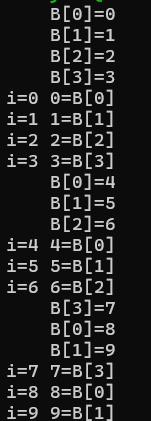
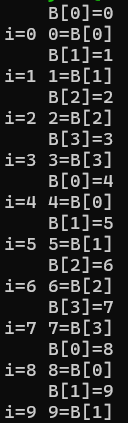
After compiling program in different ways, I got the following result:



The output of the program shows the number of itteration, and as we can see the second run had fewer itterations than the first one. It happens, since in the second run the program uses ***sched\_yield()***, which allows a thread to give up the CPU voluntarily, letting other threads run for some time before it resumes. It reduces the number of iterations, and therefore, makes the spinlock more CPU-efficient.

**Assignment: ProdCons**

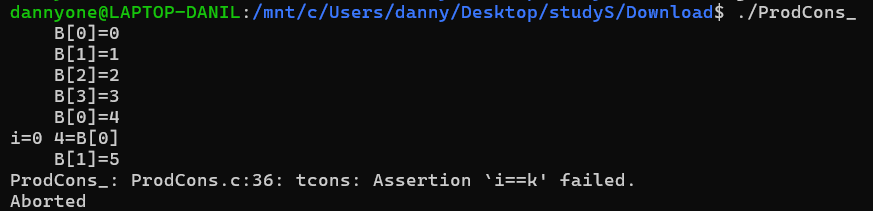
After running the program without changes and with the Spaces semaphore equal to 1, I got this result:

In the first case, the threads are less synchronized, allowing the producer to fill multiple buffer slots before the consumer catches up, whereas in the second case, each time the producer updates a single value in the buffer, the consumer immediately reads that value.

**Assignment: ProdCons initialisation**

After changing the Spaces semaphore from **N** to **N+1**, the program fails with the following output:



Here we can see that **B[0]** gets overwritten with the value **4** before the consumer has consumed the earlier values. This means the consumer reads an incorrect value and fails the ***assert(i == k)*** check. The value is overwritten, since the buffer only has **N** slots, however the *producer* thinks that there is **N+1** slots, so he overwrites the first value of the buffer, before the consumer consumes it.

**Assignment: ProdManyCons**

After running the program with the switch set to false, I got the following erorr:  

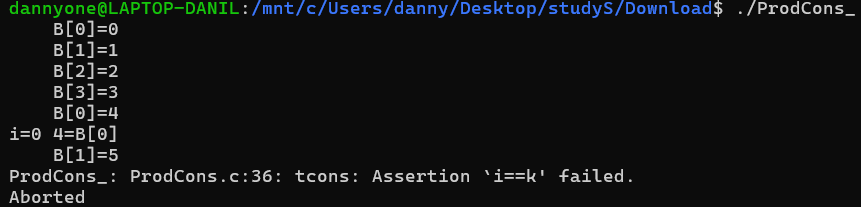

The program fails because in this case the Mutex is not used:



This leads to many consumers reading data and interfering with each other, for this reason they read copies of the data and the final *“sum assertion”* fails. If I set the switch to true, the program uses another semaphore and therefore only one consumer can read the data at the same time.

**Assignment: ProdCons sem\_wait Spaces**

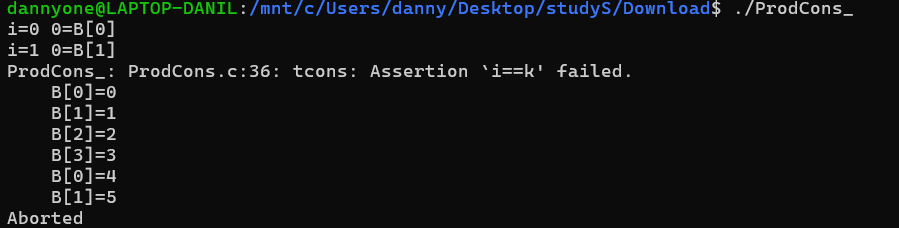
After removing the statement ***sem\_wait(&Spaces);*** I got the following error:



After removing ***sem\_wait(&Spaces);*** the producer doen’t need to wait if the buffer is full, and it starts to overwrite the data, which was not consumed yet, leading to data corruption and assertion failure.

**Assignment: ProdCons sem\_wait Elements**

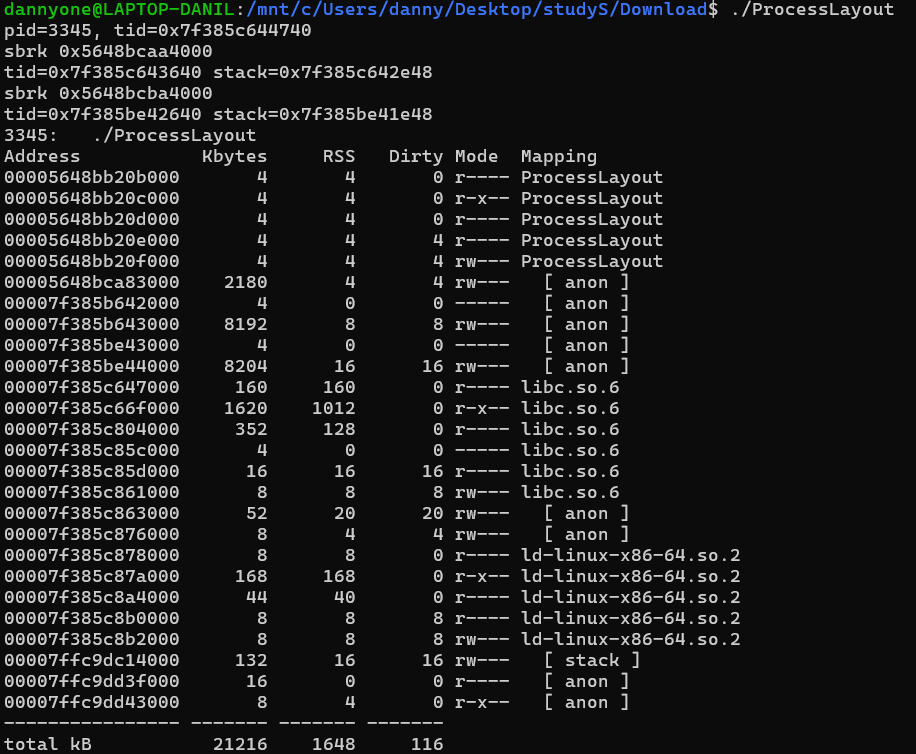
After removing the statement ***sem\_wait(&Elements);*** I got the following error:



Now, after removing ***sem\_wait(&Elements);*** , the consumer doesn’t need to wait untill buffer is full, so he starts reading from it, even though the producer didn’t put anything in the buffer yet. Since the initial value is 0, the first check succeded, however the next one already gives an error, because this value is read from the empty buffer. After the consumer fails, the producer fills the buffer and stops.

**Assignment: ProcessLayout**

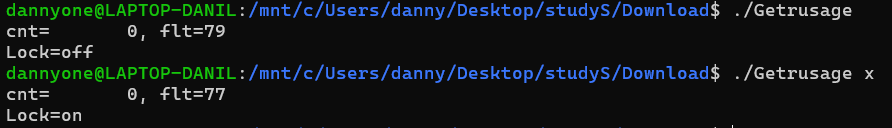
**N** value determines how many threads are created, so when I change its valuet to 2, the program creates 2 threads instead of 3. The output with two threads:



The heap (shown by ***sbrk***) and anonymous memory regions are shared by all threads (in both cases). On the other hand, each thread gets its own stack (shown by ***tid***), for this reason there are only two stacks (and therefore two ***tid’s***), when **N** is equal to two, three stacks (and therefore three ***tid’s***), when **N** is equal to three.

**Assignment: Getrusage**

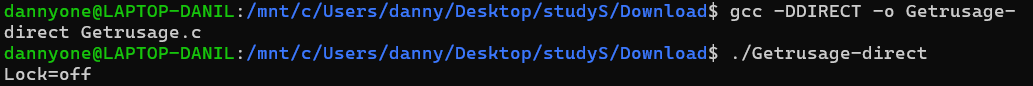
I got the following result:



The main difference is that in the second case (./*Getrusage x*), memory locking is enabled, which result in fewer page faults since memory is prevented from being swapped to disk. I am not sure why this difference is so small, probably because memory is locked in RAM, so the OS doesn’t need to fetch pages from disk.

**Assignment : Getrusage direct**

After recompiling the program and running it, I got this result:



Now the number of page faults is hidden, that’s the main difference.