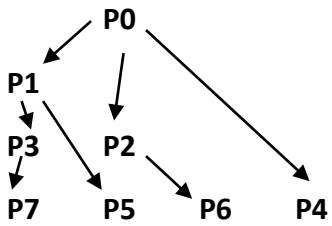
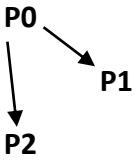
Assignment: Understanding Processes

a)

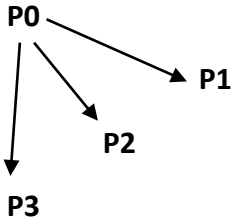


b)



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

c)



*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.

```
Program received signal SIGSEGV, Segmentation fault.
0x000055555555521d in main (argc=1, argv=0x7fffffffdf18) at Signal.c:16
16          *pointer=0;
1: pointer = (int *) 0x0
```

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:

```c
/*  Signal.c - Demonstrate how to handle a signal. */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void signal_handler(int sig) {
    printf("Signal %d received.\n", sig);
    exit(0);
}

int main(int argc, char * argv[]) {
    int *pointer=(int *) malloc(sizeof(int));
    signal(SIGSEGV,signal_handler);
    printf("About to segfault:\n");
    *pointer=0;
    printf("Why didn't we crash?\n");
    return 1;
}

/*  Please note that the checks on the return value of the system calls
    have been omitted to avoid cluttering the code. However, system calls
    can and will fail, in which case the results are unpredictable. */
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./Signal
About to segfault:
Signal 8 received.
```

The source code:

```c
/*  Signal.c - Demonstrate how to handle a signal. */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void signal_handler(int sig) {
    printf("Signal %d received.\n", sig);
    exit(0);
}

int main(int argc, char * argv[]) {
    int *pointer=(int *) malloc(sizeof(int));
    signal(SIGFPE, signal_handler);
    printf("About to segfault:\n");

    // Perform a division by zero to trigger SIGFPE
    int x = 0;
    int result = 100 / x;

    *pointer=0;
    printf("Why didn't we crash?\n");
    return 1;
}

/*  Please note that the checks on the return value of the system calls
    have been omitted to avoid cluttering the code. However, system calls
    can and will fail, in which case the results are unpredictable. */
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./Fornal
./Fornal
./Fornal
Signal 8 received.
```

The source code:

```
Signal.c    Fork.c    Fornal.c
C: > Users > danny > Desktop > studyS > Download > Fornal.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
void signal_handler(int sig) {
        printf("Signal %d received.\n", sig);
        exit(0);
        }

int main(int argc, char *argv[]) {
    pid_t pid=fork();
    printf("%s\n", argv[0]);
    if (pid==0) { /* child process */
        signal(SIGFPE, signal_handler);
        pause();    /*waits for signal*/
    } else  { /* pid!=0; parent process */
        usleep(10000); /*sleep for 10ms*/
        kill(pid, SIGFPE);  /*send the signal to child*/
        waitpid(pid,0,0); /* wait for child exit */
    return 0;
    }
}
```

Assignment: Basics of Threads

After running both programs, I got the following output:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./DashFunc
tions
Executing both functions
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
----------------------_______________________________________________________
_____________________________________________________________________________
_____________________________________________________________________________
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./DashThre
ads
Launching the threads dash and underscore
------_-_-_-_-_-_-------------_-_-_-_--_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-
-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_--_-_---_-_-___________-_-_-_-_-_-_-_-
-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_____________-_-
-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
-_-_-_-_-_-_-_-_-_------------------------------
Both threads finished
End of main thread
```

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.

```
Thread 986650921
Can't create thread 31412
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$
```

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

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

```
Thread 92717641
Can't create thread 9630
one@danny:~/Downloads/Download $
```

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:

```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_S
YSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, child_tid=0x7f3f
0af7e910, parent_tid=0x7f3f0af7e910, exit_signal=0, stack=0x7f3f0a77e000, stac
k_size=0x7fff00, tls=0x7f3f0af7e640} => {parent_tid=[359319]}, 88) = 359319
```

Assignment: MyThread Timing

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

C code

```
real    0m0.008s
user    0m0.005s
sys     0m0.000s
dannyone@LAPTOP-DANIL
```

Java code

```
real    0m0.175s
user    0m0.186s
sys     0m0.041s
dannyone@LAPTOP-DANIL
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ java -ea Count
inc     ctr     SEM=false

-1      -1
-1      -2
-1      -3
1       1
1       2
1       3
1       4
1       5
1       6
1       7
1       8
1       9
-1      -4
1       10
-1      -5
-1      -6
-1      -7
-1      -8
-1      -9
-1      -10
Final value ctr=-10
Exception in thread "main" java.lang.AssertionError
        at Count.main(Count.java:40)
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ java -ea Count
inc     ctr     SEM=false

-1      -1
-1      -2
-1      -3
1       1
-1      -4
1       2
-1      -5
-1      -6
-1      -7
-1      -8
-1      -9
-1      -10
1       3
1       4
1       5
1       6
1       7
1       8
1       9
1       10
Final value ctr=10
Exception in thread "main" java.lang.AssertionError
        at Count.main(Count.java:40)
```

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

```
SEM = args.length > 0 ;
```

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***):

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ java -
ea Count f
inc     ctr     SEM=true

1       1
1       2
-1      1
-1      0
-1      -1
-1      -2
-1      -3
-1      -4
-1      -5
-1      -6
1       -5
1       -4
1       -3
1       -2
1       -1
1       0
1       1
1       2
-1      1
-1      0
Final value 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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./Coun
t f
inc     ctr     Switch=true
1       1
1       2
1       3
1       4
1       5
1       6
1       7
1       8
1       9
1       10
-1      9
-1      8
-1      7
-1      6
-1      5
-1      4
-1      3
-1      2
-1      1
-1      0
```

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.

```
[pid  2706] futex(0x55de4a60e040, FUTEX_WAIT_BITSET_PRIVATE|FUTEX_CLOCK_REALTIME, 0, NULL, FUTEX_BITSET_MATCH_ANY <unfinished ...>
[pid  2707] <... write resumed>)        = 5
[pid  2707] sched_yield()               = 0
[pid  2707] futex(0x55de4a60e040, FUTEX_WAKE_PRIVATE, 1) = 1
```

Assignment: Spinlock

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

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ gcc -o Spinlock Spinlock.c -lpthread
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./Spinlock
113488611
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ gcc -DYIELD -o Spinlock-yield Spinlock.c -lpthread
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./Spinlock
112580369
```

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:

```
    B[0]=0
    B[1]=1
    B[2]=2
    B[3]=3
i=0 0=B[0]
i=1 1=B[1]
i=2 2=B[2]
i=3 3=B[3]
    B[0]=4
    B[1]=5
    B[2]=6
i=4 4=B[0]
i=5 5=B[1]
i=6 6=B[2]
    B[3]=7
    B[0]=8
    B[1]=9
i=7 7=B[3]
i=8 8=B[0]
i=9 9=B[1]
```

```
    B[0]=0
i=0 0=B[0]
    B[1]=1
i=1 1=B[1]
    B[2]=2
i=2 2=B[2]
    B[3]=3
i=3 3=B[3]
    B[0]=4
i=4 4=B[0]
    B[1]=5
i=5 5=B[1]
    B[2]=6
i=6 6=B[2]
    B[3]=7
i=7 7=B[3]
    B[0]=8
i=8 8=B[0]
    B[1]=9
i=9 9=B[1]
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./ProdCons_
    B[0]=0
    B[1]=1
    B[2]=2
    B[3]=3
    B[0]=4
i=0 4=B[0]
    B[1]=5
ProdCons_: ProdCons.c:36: tcons: Assertion `i==k' failed.
Aborted
```

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:

```
7=B[0]
ProdManyCons: ProdManyCons.c:32: tcons: Assertion `sum == (P*M-1)*P*M/2' failed
Aborted
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./ProdManyCons
```

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

```
if(Switch) {
    sem_wait(&Mutex);
}
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./ProdCons_
    B[0]=0
    B[1]=1
    B[2]=2
    B[3]=3
    B[0]=4
i=0 4=B[0]
    B[1]=5
ProdCons_: ProdCons.c:36: tcons: Assertion `i==k' failed.
Aborted
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./ProdCons_
i=0 0=B[0]
i=1 0=B[1]
ProdCons_: ProdCons.c:36: tcons: Assertion 'i==k' failed.
    B[0]=0
    B[1]=1
    B[2]=2
    B[3]=3
    B[0]=4
    B[1]=5
Aborted
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./ProcessLayout
pid=3345, tid=0x7f385c644740
sbrk 0x5648bcaa4000
tid=0x7f385c643640 stack=0x7f385c642e48
sbrk 0x5648bcba4000
tid=0x7f385be42640 stack=0x7f385be41e48
3345:   ./ProcessLayout
Address           Kbytes     RSS   Dirty Mode  Mapping
00005648bb20b000       4       4       0 r---- ProcessLayout
00005648bb20c000       4       4       0 r-x-- ProcessLayout
00005648bb20d000       4       4       0 r---- ProcessLayout
00005648bb20e000       4       4       4 r---- ProcessLayout
00005648bb20f000       4       4       4 rw--- ProcessLayout
00005648bca83000    2180       4       4 rw---   [ anon ]
00007f385b642000       4       0       0 -----   [ anon ]
00007f385b643000    8192       8       8 rw---   [ anon ]
00007f385be43000       4       0       0 -----   [ anon ]
00007f385be44000    8204      16      16 rw---   [ anon ]
00007f385c647000     160     160       0 r---- libc.so.6
00007f385c66f000    1620    1012       0 r-x-- libc.so.6
00007f385c804000     352     128       0 r---- libc.so.6
00007f385c85c000       4       0       0 ----- libc.so.6
00007f385c85d000      16      16      16 r---- libc.so.6
00007f385c861000       8       8       8 rw--- libc.so.6
00007f385c863000      52      20      20 rw---   [ anon ]
00007f385c876000       8       4       4 rw---   [ anon ]
00007f385c878000       8       8       0 r---- ld-linux-x86-64.so.2
00007f385c87a000     168     168       0 r-x-- ld-linux-x86-64.so.2
00007f385c8a4000      44      40       0 r---- ld-linux-x86-64.so.2
00007f385c8b0000       8       8       8 r---- ld-linux-x86-64.so.2
00007f385c8b2000       8       8       8 rw--- ld-linux-x86-64.so.2
00007ffc9dc14000     132      16      16 rw---   [ stack ]
00007ffc9dd3f000      16       0       0 r----   [ anon ]
00007ffc9dd43000       8       4       0 r-x--   [ anon ]
---------------- ------- ------- -------
total kB           21216    1648     116
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./Getrusage
cnt=       0, flt=79
Lock=off
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./Getrusage x
cnt=       0, flt=77
Lock=on
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ gcc -DDIRECT -o Getrusage-
direct Getrusage.c
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop/studyS/Download$ ./Getrusage-direct
Lock=off
```

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