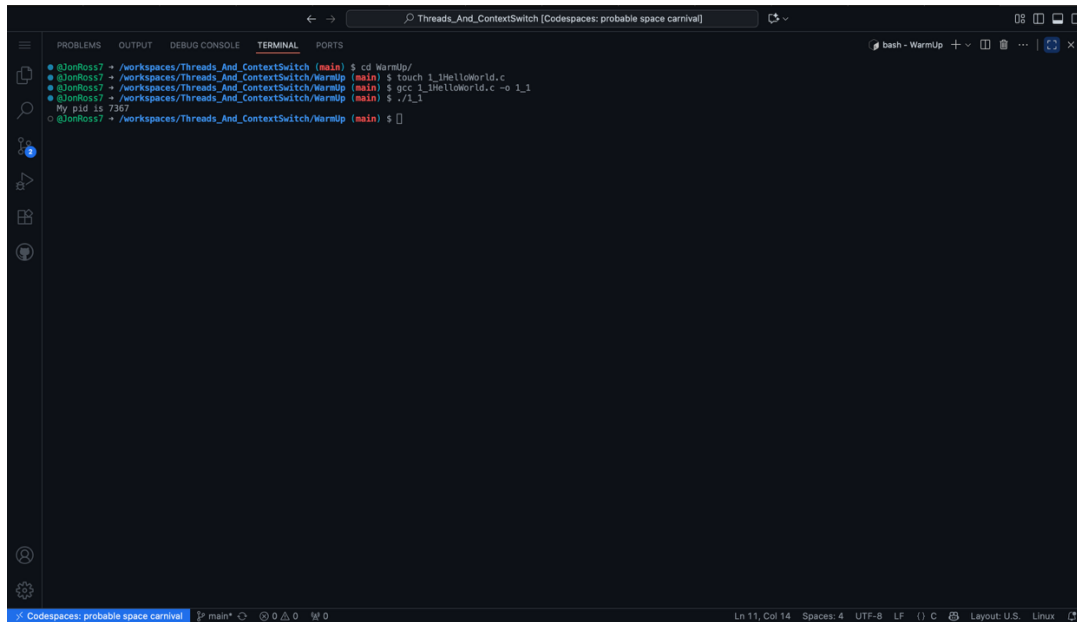


# Threads and Context Switching

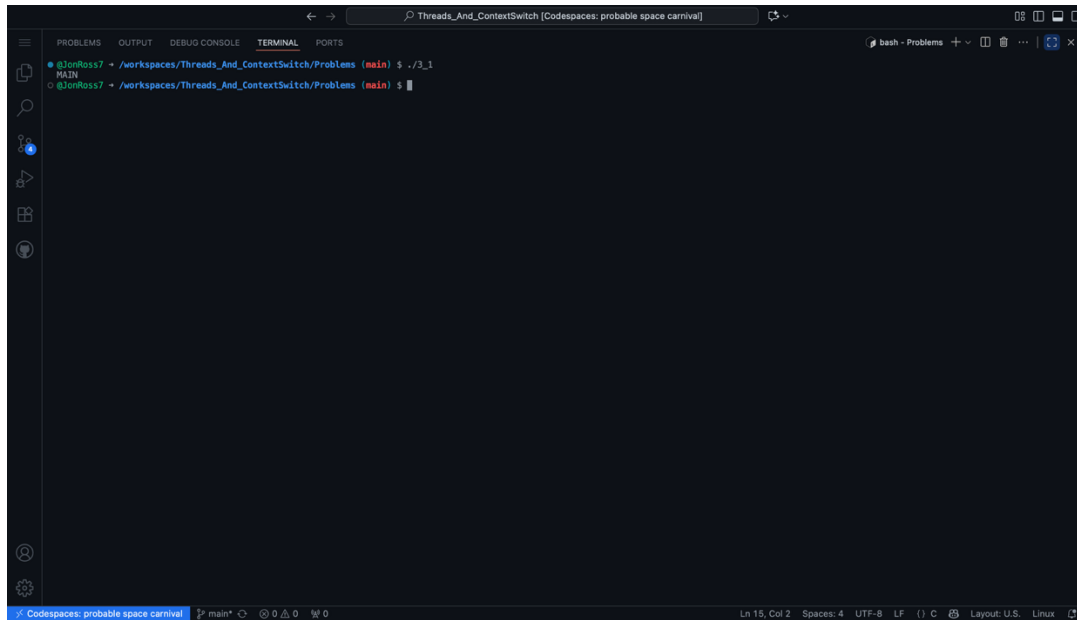
## 1.1 Hello World:



```
Threads_And_ContextSwitch (Codespaces: probable space carnival)
bash - WarmUp
@JonRoss7 → /workspaces/Threads_And_ContextSwitch (main) $ cd WarmUp/
@JonRoss7 → /workspaces/Threads_And_ContextSwitch/WarmUp (main) $ touch 1_helloWorld.c
@JonRoss7 → /workspaces/Threads_And_ContextSwitch/WarmUp (main) $ gcc 1_helloWorld.c -o 1_1
@JonRoss7 → /workspaces/Threads_And_ContextSwitch/WarmUp (main) $ ./1_1
My pid is 7367
@JonRoss7 → /workspaces/Threads_And_ContextSwitch/WarmUp (main) $
```

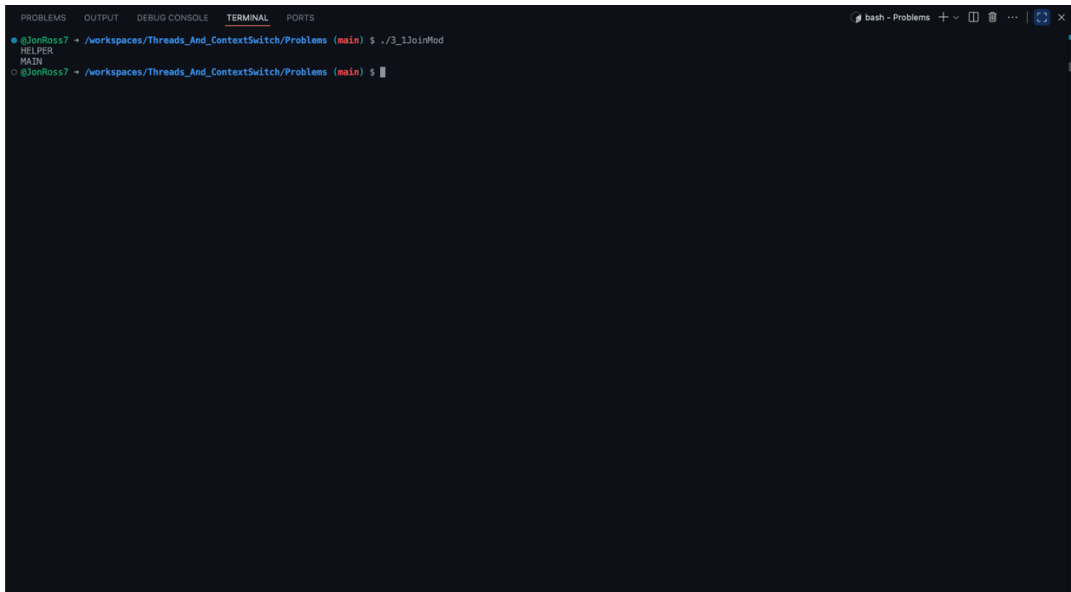
The following C code produces “My pid is [number]”

## 3.1 Join:



```
Threads_And_ContextSwitch (Codespaces: probable space carnival)
bash - Problems
@JonRoss7 → /workspaces/Threads_And_ContextSwitch/Problems (main) $ ./3_1
MAIN
@JonRoss7 → /workspaces/Threads_And_ContextSwitch/Problems (main) $
```

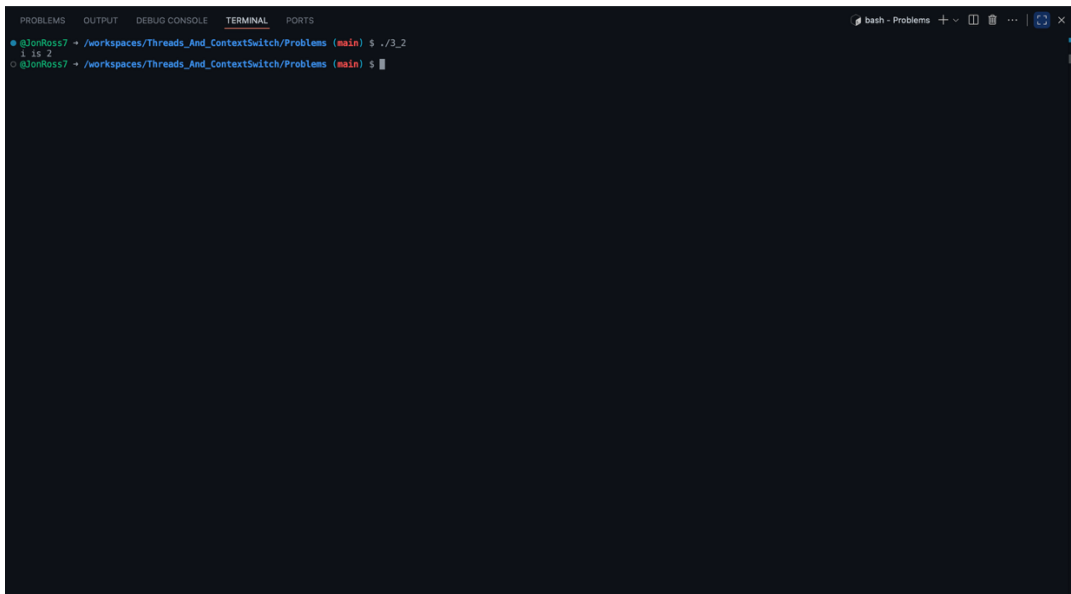
It will print MAIN from the original version of the code



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
@JonRoss7 ~ /workspaces/Threads_And_ContextSwitch/Problems (main) $ ./3_13joined
HELPER
MAIN
@JonRoss7 ~ /workspaces/Threads_And_ContextSwitch/Problems (main) $
```

To make sure HELPER is printed first, you have to wait for the helper thread to finish before printing in main. In pthreads, this is done by `pthread_join`.

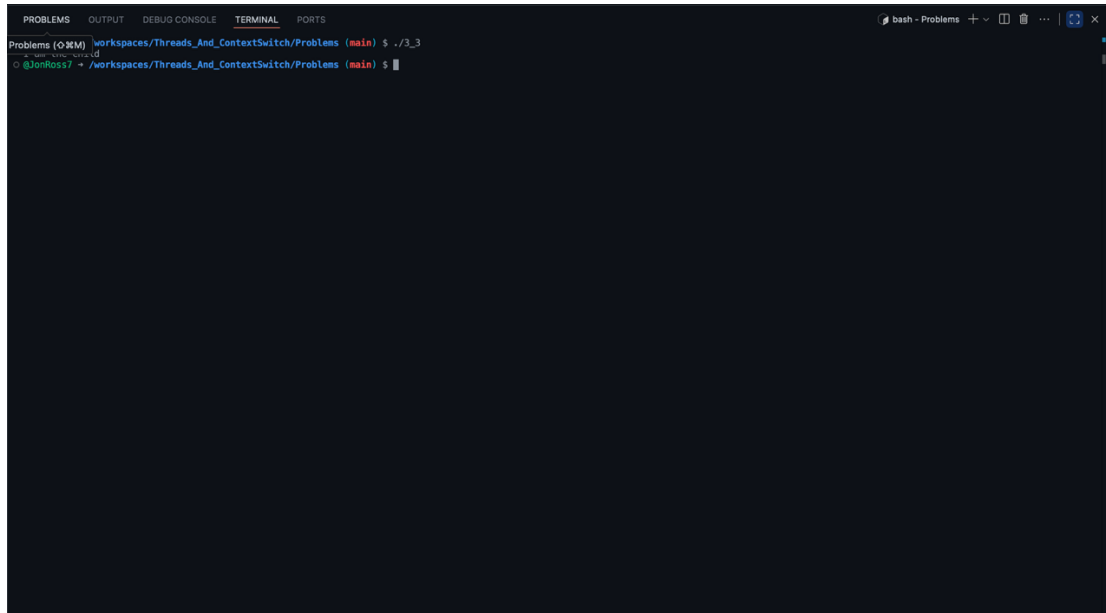
### 3.2 Stack Allocation:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
@JonRoss7 ~ /workspaces/Threads_And_ContextSwitch/Problems (main) $ ./3_2
1 is 2
@JonRoss7 ~ /workspaces/Threads_And_ContextSwitch/Problems (main) $
```

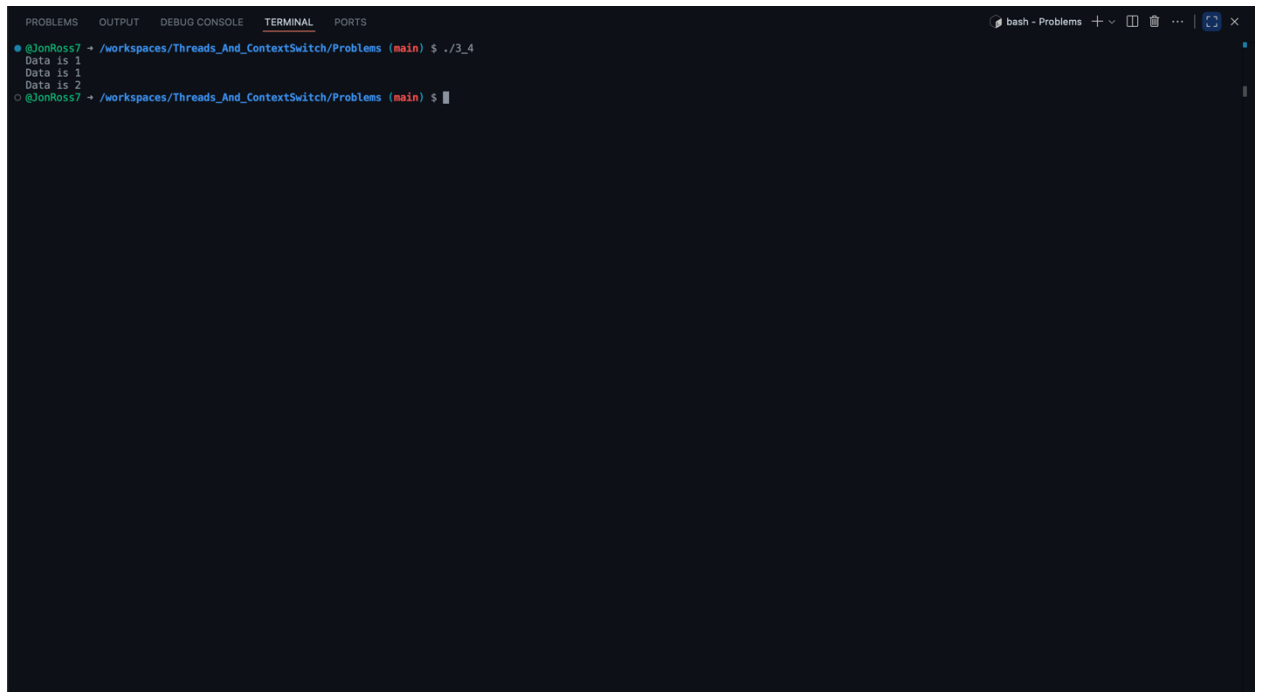
The following C code produces “i is 2.”

### 3.3 Heap Allocation:



The following C code produces “I am the Child”

### 3.4 Threads and Processes:



The following C code produces “Data is 1, Data is 1, Data is 2”. You could retrieve the return value of worker by using pthread\_join and void\*.

### 3.5 Context Switching:

1. There are two stacks involved in this process. These two stacks are called the current thread stack and the next thread stack. One of these stacks retain the old thread's state, while the other provides the execution environment for the new thread
2. The order matters because the pop instruction must restore the registers in the reverse order the registers were saved in. This maintains the correct register state and stack alignment

### 3.6 Reflections on Threads:

1. Web servers and image processing are two examples in which multithreading can provide better performance than a single-thread solution.
2. User -Level Threads are managed by applications, faster context switching, but blocked system calls. Meanwhile, Kernel-level threads are managed by the OS and offer true parallelism. However, there is slower context switching. Kernel-level is better for I/O intensive tasks, while CPU-intensive tasks are better for user-level.
3. The actions taken by the kernel to context-switch between kernel level threads are to first save the current thread's register name. Then, you should update the thread control block. Afterwards, you should switch memory mappings if necessary. In addition, you should be able to restore the new thread's state. Lastly, it should be able to update the scheduler's data structures.
4. The resources used for thread creation are stack space, register context, and the thread control block. For process creation, it is the resources that are used for thread creation along with more memory space, file descriptors, and the process control block.