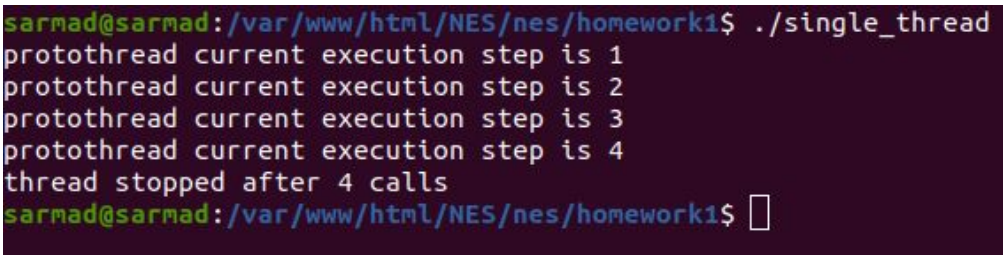
Group: Homework1_Group_110

Q2-4: In the **protothread** function a step_count variable is incremented every time the function is called from the main function and a condition is placed in the function to return the DONE as soon protothread has reached its fourth execution. To keep the last incremented value of step_count, the variable is made static.

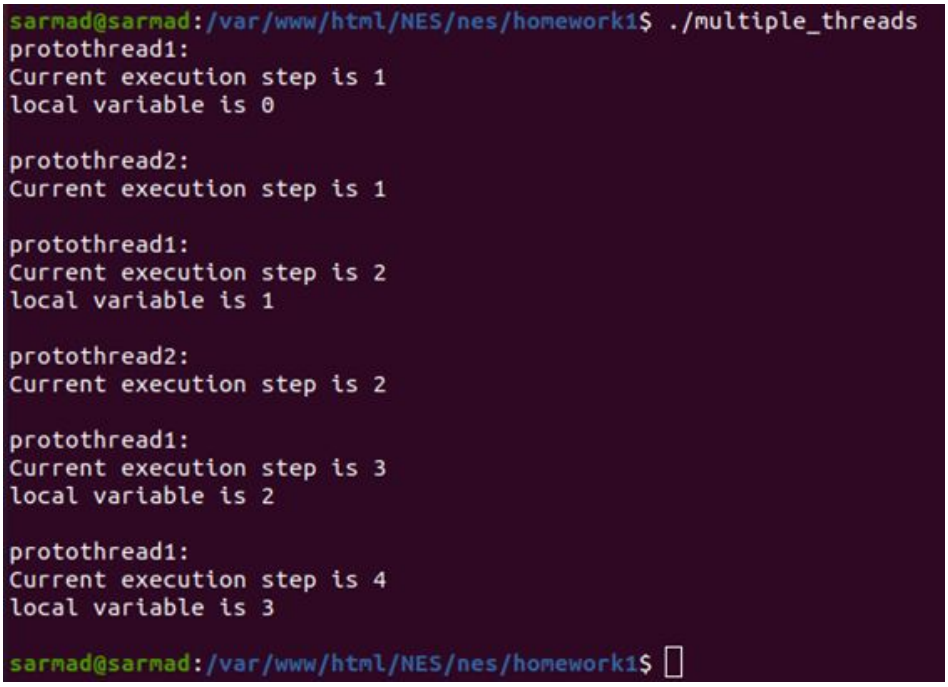


Q3-3 Multithread: protothread1 and **protothread2** have multiple execution steps (i.e 4 and 2 respectively). Both functions declare a separate static variable to keep a track of step count. Static variables are used to preserve the count between multiple execution steps.

Q3-5 Multithread with local variable:

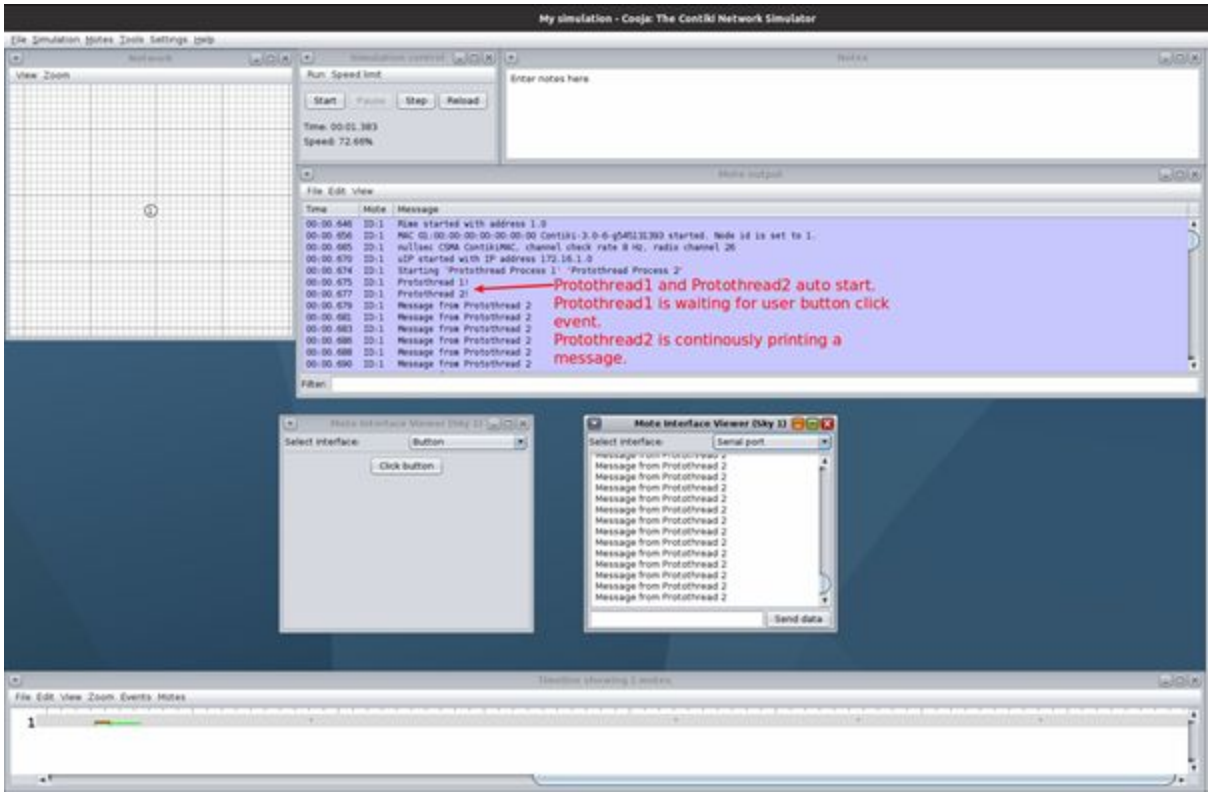
The local variable initiated by the protothread1 does not maintain its state/value between the multiple execution steps because a local variable exists on stack. The lifetime of such a local variable is only for that function call. As soon as the function execution is finished, the stack is freed. Every time the same function is called again, the stack is allocated again (possibly on a new address) and hence the function cannot obtain data that it created in the previous calls. So the local variable is assigned an indeterminate value (garbage for us) and its behavior is undefined which means we cannot predict its value and it can change from compiler to compiler and for each execution.

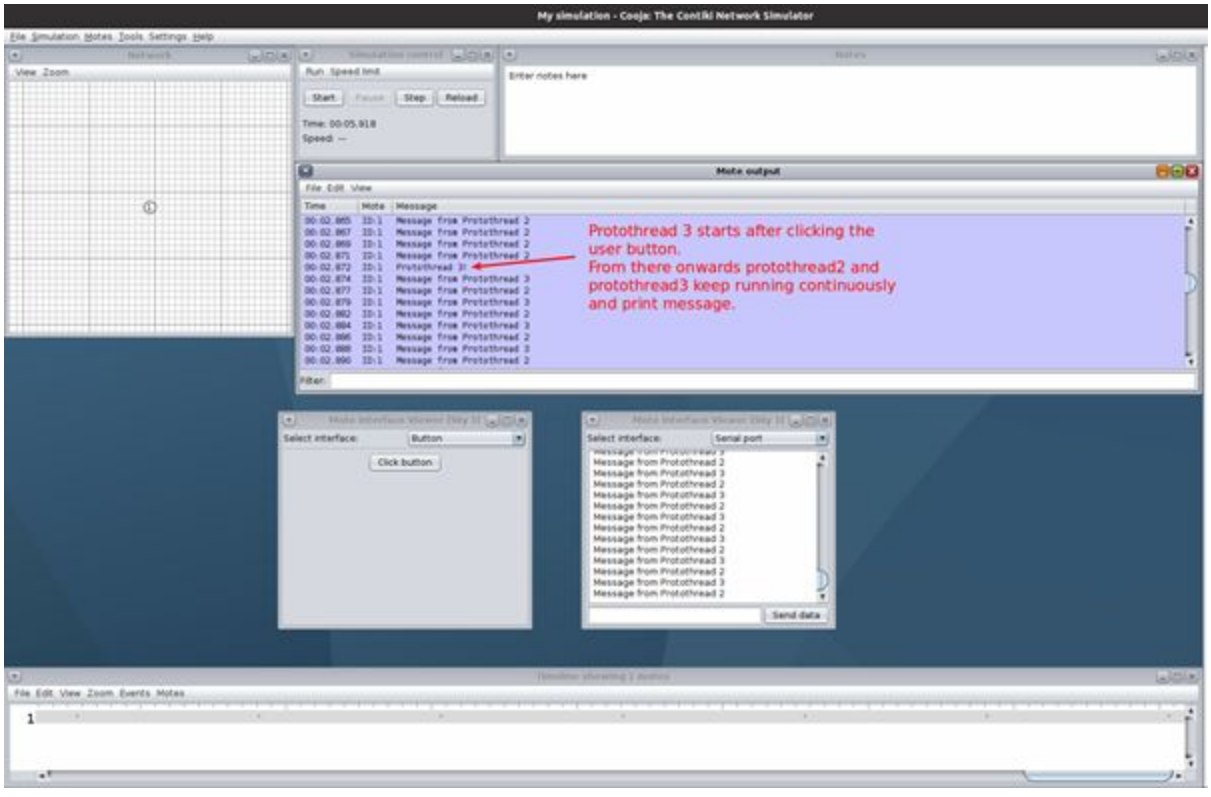
The local variables on stack should only be used when their state/data does not need to be preserved for the multiple calls of the same function/block/thread/process.
Solution:
Declare the variable as “static”. The static variables preserve their value because they remain in memory during the program execution. The reason is that static variables are not allocated memory on the stack but on the data segment. A program has a virtual address space and the data segment is part of it. The values can be read or written for the data segment.



Q4-1

2 new protothreads are created which are named as **protothread2** and **protothread3.** When the execution begins, The **protothread1** and **protothread2** autostarts (using AUTOSTART_PROCESS). The **protothread2** remains executing continuously and prints a message while **protothread1** is waiting for a button click event. When a button is pressed, the **protothread1** starts a new protothread i.e **protothread3** and the process itself ends. After that, both the new protothreads i.e **protothread2** and **protothread3** keeps on executing. The macro used here is **PROCESS_PAUSE** which yields a protothread temporarily so the other protothread can also execute.


My simulation - Cooja: The Contiki Network Simulator
Protothread1 and Protothread2 auto start.
Protothread1 is waiting for user button click event.
Protothread2 is continously printing a message.
Mote Interface Viewer (Sky 1)
Click button
Send data


My simulation - Cooja: The Contiki Network Simulator
Mote output
Protothread 3 starts after clicking the user button.
From there onwards protothread2 and protothread3 keep running continuously and print message.
Mote Interface Viewer (Sky 1)
Click button
Send data

Q4-3 Protothreads:

a) Protothreads is a way of running multiple tasks without any dedicated scheduler. Protothreads are event driven, light weight because of its stackless implementation and provide blocking context, without any overhead. Protothreads are lightweight in comparison to others because each thread doesn't have its own stack, all threads use a common stack it gives advantage of light weightiness as well as it has its own drawbacks like the values of local variables in a thread that are changed during the execution are not retained when thread yields instead it keeps the initialized value on stack. In order to avoid such cases static variables are used in protothreads to keep the updated value within the multiple executions.

b) As the protothread is event driven and doesn't have any dedicated scheduler while Linux threads are managed by the scheduler. In protothreads if we declare a same static variable (let say **i**) each thread will have its own copy of **i** and they will preserve their state within multiple executions while in linux threads static variables are shared among all the threads. So, it is not possible to reimplement Linux thread for sensor motes.

Q4-4:

PROCESS_BEGIN(): It defines the initiation of process. The processing in thread starts with this macro.
PROCESS_END(): It defines the termination of process. The process ends after this command so it should be at the end of all processing.
PROCESS_PAUSE(): This yields the process for some time and let the other process complete their execution.
PROCESS_WAIT_EVENT(): This macro stops the execution of the running process and waits for an event to be posted.
PROCESS_WAIT_EVENT_UNTIL(): It is similar to PROCESS_WAIT_EVENT() but with some additional condition.
PROCESS_WAIT_UNTIL(): This macro waits for a condition to be true until then it blocks the process.
PROCESS_YIELD(): it yields the executing process.
PROCESS_EXIT(): Exit the running current process.