**Threads Creation and Execution**

**LAB # 08**



**Spring 2023**

**CSE-204L Operating Systems Lab**

“On my honor, as student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work.”

Submitted to:

**Engr. Madiha Sher**

Date:

**24th May 2023**

**Department of Computer Systems Engineering**

**University of Engineering and Technology, Peshawar**

**Objectives:**

This lab examines aspects of **threads** and **multiprocessing** (and **multithreading**). The primary objective of this lab is to implement Thread Management Functions:

|  |
| --- |
| **Creating Threads** |
| **Terminating Thread Execution** |
| **Thread Identifiers** |
| **Joining Threads** |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

#### 

**What is thread?**

A **thread** is a semi-process, that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

**What are pthreads?**

|  |  |
| --- | --- |
|  | Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. |

|  |  |
| --- | --- |
|  | In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's. |

|  |  |
| --- | --- |
|  | **Pthreads** are defined as a set of **C language programming types and procedure calls**. Vendors usually provide a Pthreads implementation in the form of a **header/include file** and a library which you **link** with your program. |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Why pthreads?

|  |  |
| --- | --- |
|  | The primary motivation for using Pthreads is to realize potential program performance gains. |
|  | When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes. |
|  | All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication. |
|  | Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways: Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads. Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks. Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests. |
|  | Multi-threaded applications will work on a uniprocessor system, yet naturally take advantage of a multiprocessor system, without recompiling. |
|  | In a multiprocessor environment, the most important reason for using Pthreads is to take advantage of potential parallelism. This will be the focus of the remainder of this session. |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Thread Management Functions:**

The function **pthread\_create** is used to create a new thread, and the function **pthread\_exit** is used by a thread to terminate itself. The function **pthread\_join** is used by a thread to wait for termination of another thread.

|  |  |
| --- | --- |
| Function: | int **pthread\_create** (pthread\_t   \*threadhandle, */\*Thread handle returned by reference \*/* pthread\_attr\_t   \*attribute, */\* Special Attribute for starting thread, may be NULL \*/* void  \*(\*start\_routine)(void \*), */\* Main Function which thread executes \*/* void  \*arg */\* An extra argument passed as a pointer \*/ );* |
| Info: | Request the **PThread** library for **creation** of a new thread. The return value is **0** on **success**. The **pthread\_t** is an abstract datatype that is used as a handle to **reference** the thread. |

|  |  |
| --- | --- |
| Function: | Void **pthread\_exit** (void \*retval */\* return value passed as a pointer \*/*); |
| Info: | This Function is used by a thread to **terminate**. The return value is passed as a **pointer**. This pointer value can be anything so long as it does not exceed the size of (void \*). Be careful, this is system dependent. You may wish to return an address of a structure, if the returned data is very large. |

|  |  |
| --- | --- |
| Function: | Int **pthread\_join** (pthread\_t    threadhandle, */\* Pass threadhandle \*/* void   \*\*returnvalue */\* Return value is returned by ref. \*/*); |
| Info: | Return **0** on **success**, and **negative** on **failure**. The returned value is a **pointer** returned by **reference**. If you do not care about the return value, you can pass **NULL** for the second argument. |

**Thread Initialization:**

**Include the pthread.h library :**

**#include <pthread.h>**

**Declare a variable of type pthread\_t :**

**pthread\_t    the\_thread**

**When you compile, add -lpthread to the linker flags :**

**gcc   threads.c   -o   threads  -lpthread**

**+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++**

**Initially, threads are created from within a process. Once created, threads are peers, and may create other threads. Note that an "initial thread" exists by default and is the thread which runs main( ).**

**+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++**

**Thread Identifiers:**

**pthread\_self ( )**

Returns the unique thread ID of the calling thread. The returned data object is opaque can not be easily inspected.

**pthread\_equal ( thread1, thread2 )**

**Compares two thread IDs:**

If the two IDs are different 0 is returned, otherwise a non-zero value is returned.   
Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs.

**Example: Pthread Creation and Termination:**

**Lab6\_1.c**

**#include <stdio.h>**

**#include <pthread.h>**

**void \*kidfunc(void \*p)**

**{  
    printf ("Kid ID is ---> %d\n", getpid( ));**

**}  
  
main ( )**

**{  
    pthread\_t kid ;**

**pthread\_create (&kid, NULL, kidfunc, NULL);**

**printf ("Parent ID is ---> %d\n", getpid( )) ;**

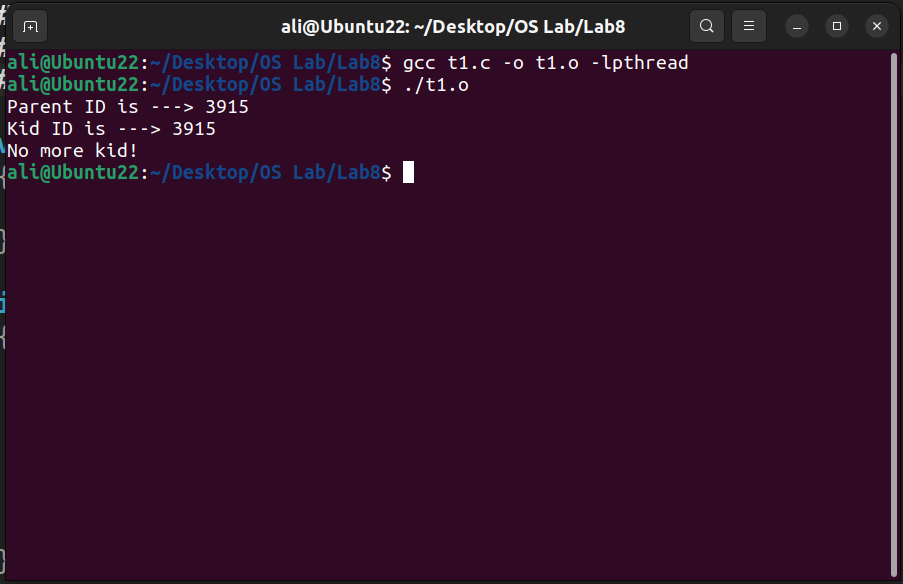
**pthread\_join (kid, NULL) ;**

**printf ("No more kid!\n") ;**

**}**

**Question: Are the process id numbers of parent and child thread the same or different? Give reason(s) for your answer.**

**Answer: Parent and child thread will have same process ID as they both are a part of the same process.**



**""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""**

**Lab6\_2.c**

**#include <stdio.h>**

**#include <pthread.h>**

**int glob\_data = 5 ;**

**void \*kidfunc(void \*p)**

**{  
    printf ("Kid here. Global data was %d.\n", glob\_data) ;**

**glob\_data = 15 ;**

**printf ("Kid Again. Global data was now %d.\n", glob\_data) ;**

**}  
  
main ( )**

**{  
    pthread\_t kid ;**

**pthread\_create (&kid, NULL, kidfunc, NULL) ;**

**printf ("Master thread here. Global data = %d\n", glob\_data) ;**

**glob\_data = 10 ;**

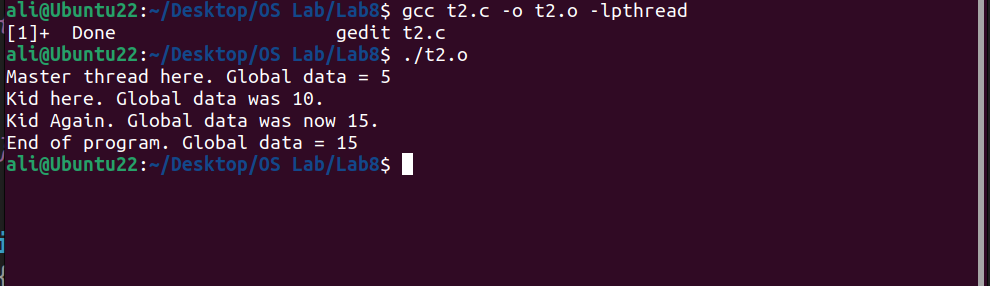
**pthread\_join (kid, NULL) ;**

**printf ("End of program. Global data = %d\n", glob\_data) ;**

**}**

**Question: Do the threads have separate copies of glob\_data?** Why? Or why not?

**Answer: No, threads do not have separate copy of glob\_data. Both master and child thread are accessing the glob\_data.**



**"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""**

**Multiple Threads:**

The simple example code below creates 5 threads with the **pthread\_create( )** routine. Each thread prints a "Hello World!" message, and then terminates with a call to **pthread\_exit( )**.

**Lab6\_3.c**  
**#include <pthread.h>**

**#include <stdio.h>**

**#define NUM\_THREADS 5**

**void \*PrintHello(void \*t)**

**{  
    printf("\n%ld: Hello World!\n", threadid);**

**pthread\_exit(NULL);**

**}  
int main( )**

**{  
    pthread\_t   threads [NUM\_THREADS];**

**int rc, t;**

**for(t=0; t < NUM\_THREADS; t++)    {**

**printf ("Creating thread %d\n", t);**

**rc = pthread\_create (&threads[t], NULL, PrintHello, NULL );**

**if (rc) {**

**printf("ERROR; return code from pthread\_create() is %d\n", rc);**

**exit(-1);**

**}  
       }**

   **pthread\_exit(NULL);**

**}**

**Sample output**

ccse> lab6\_3

Creating thread 0

Creating thread 1

Creating thread 2

Creating thread 3

Creating thread 4

0: Hello World!

1: Hello World!

2: Hello World!

3: Hello World!

4: Hello World!

**Program Output:**

A screenshot of a computer

Description automatically generated

**"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""**

**Difference between process and threads :**

**Lab6\_4.c**

**#include <stdio.h>**

**#include <pthread.h>**

**#include <unistd.h>**

**int this\_is\_global;**

**void thread\_func( void \*ptr );**

**int main( ) {**

**int local\_main;**

**int pid, status;**

**pthread\_t thread1, thread2;**

**printf("First, we create two threads to see better what context they share...\n");**

**this\_is\_global=1000;  
printf("Set this\_is\_global=%d\n",this\_is\_global);**

**pthread\_create( &thread1, NULL, (void\*)&thread\_func, (void\*) NULL);**

**pthread\_create(&thread2, NULL, (void\*)&thread\_func, (void\*) NULL);**

**pthread\_join(thread1, NULL);**

**pthread\_join(thread2, NULL);**

**printf("After threads, this\_is\_global=%d\n",this\_is\_global);**

**printf("\n");  
printf("Now that the threads are done, let's call fork..\n");**

**local\_main=17; this\_is\_global=17;**

**printf("Before fork(), local\_main=%d, this\_is\_global=%d\n",local\_main,**

**this\_is\_global);  
pid=fork();**

**if (pid == 0) { /\* this is the child \*/**

**printf("In child, pid %d: &global: %X, &local: %X\n", getpid(), &this\_is\_global, &local\_main);**

**local\_main=13; this\_is\_global=23;**

**printf("Child set local main=%d, this\_is\_global=%d\n",local\_main,**

**this\_is\_global);  
 exit(0);  
}  
else { /\* this is parent \*/**

**printf("In parent, pid %d: &global: %X, &local: %X\n", getpid(), &this\_is\_global, &local\_main);**

**wait(NULL);  
 printf("In parent, local\_main=%d, this\_is\_global=%d\n",local\_main,**

**this\_is\_global);**

**}  
exit(0);  
}  
  
void thread\_func(void \*dummy) {**

**int local\_thread;**

**printf("Thread %d, pid %d, addresses: &global: %X, &local: %X\n",**

**pthread\_self(), getpid(), &this\_is\_global, &local\_thread);**

**this\_is\_global++;  
printf("In Thread %d, incremented this\_is\_global=%d\n", pthread\_self(),**

**this\_is\_global);  
pthread\_exit(0);  
}**

**Sample output**

ccse> lab6\_4

First, we create two threads to see better what context they share...

Set this\_is\_global=1000

Thread 4, pid 2524, addresses: &global: 20EC8, &local: EF20BD6C

In Thread 4, incremented this\_is\_global=1001

Thread 5, pid 2524, addresses: &global: 20EC8, &local: EF109D6C

In Thread 5, incremented this\_is\_global=1002

After threads, this\_is\_global=1002

Now that the threads are done, let's call fork..

Before fork(), local\_main=17, this\_is\_global=17

In child, pid 2525: &global: 20EC8, &local: EFFFFD34

Child set local main=13, this\_is\_global=23

In parent, pid 2524: &global: 20EC8, &local: EFFFFD34

In parent, local\_main=17, this\_is\_global=17

**Program Output:**

A picture containing text, screenshot, font

Description automatically generated

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

**Assignments:**

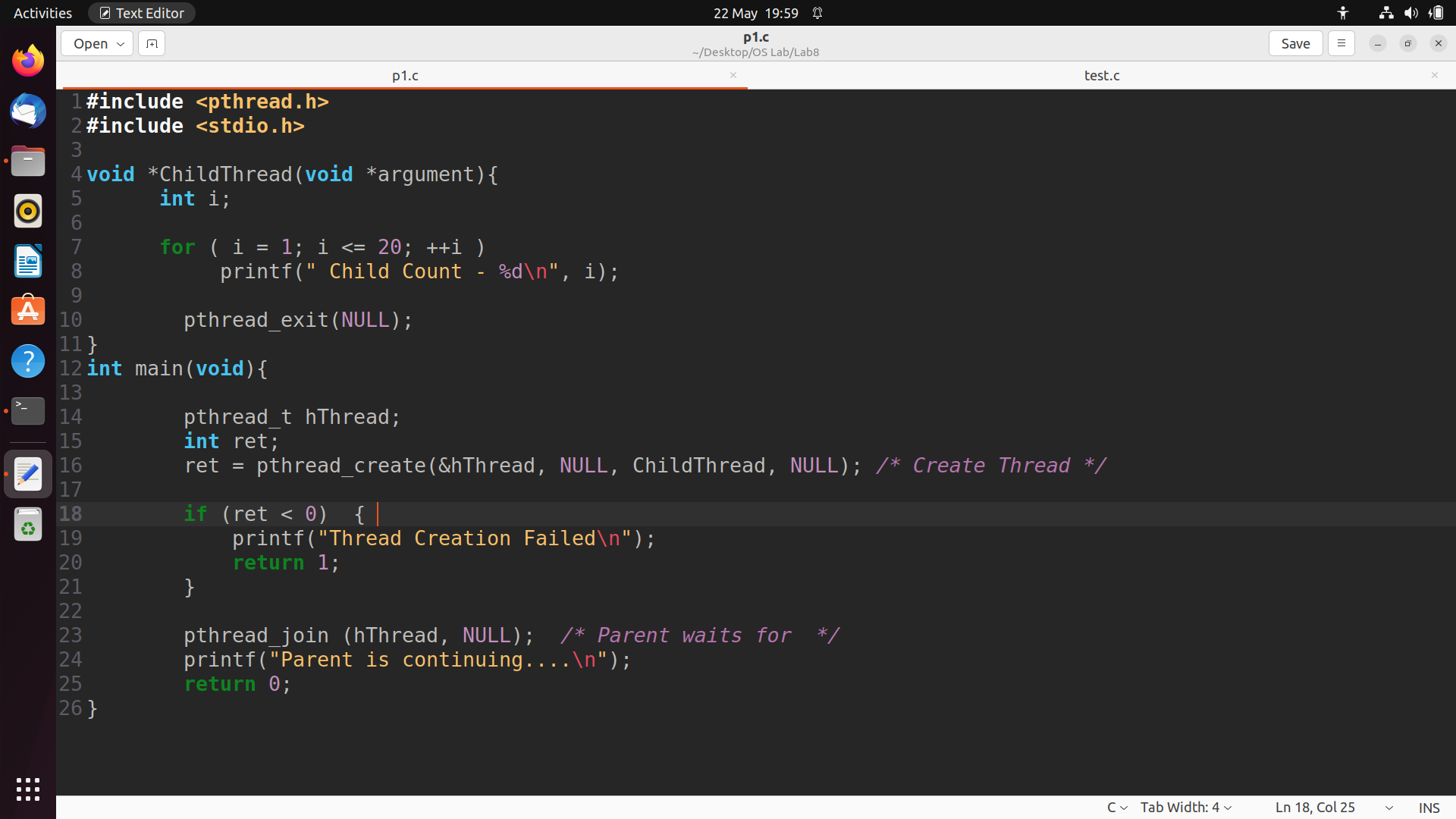
**Problem#1:**

The following **Box #1** program demonstrates a simple program where the **main thread creates another thread** to print out the numbers from 1 to 20. The **main thread waits till the child thread finishes**.

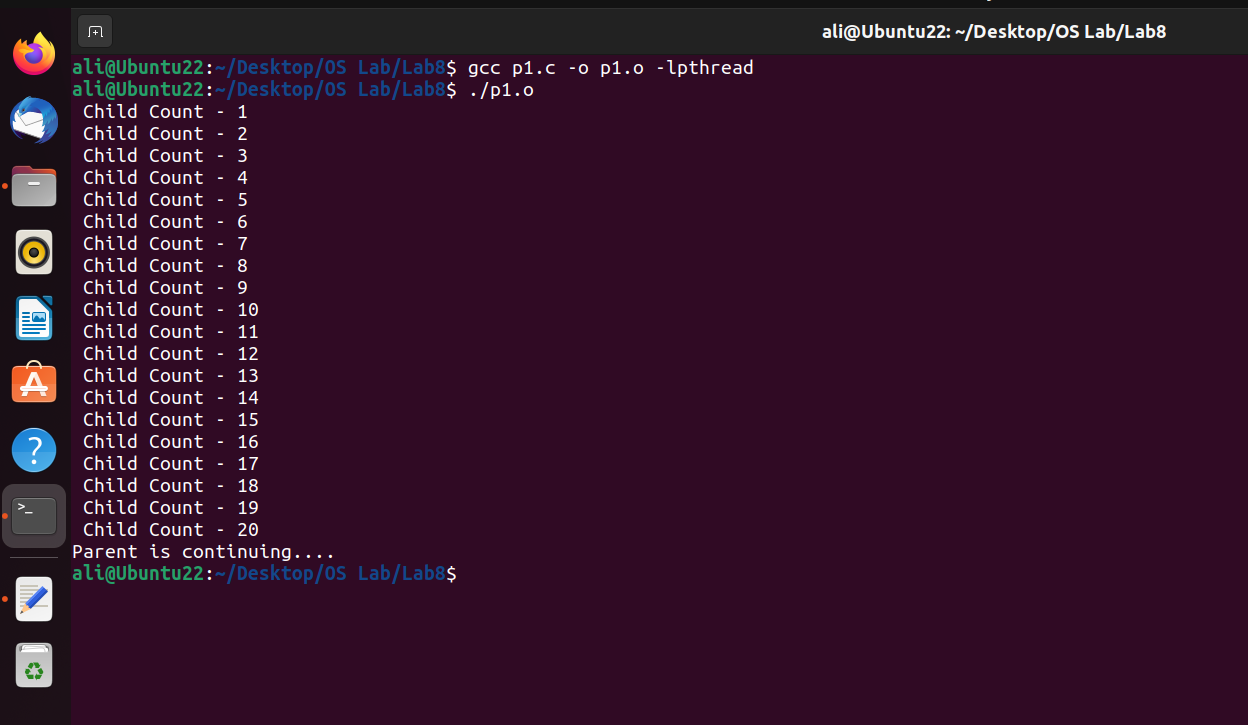
|  |
| --- |
| **/\* Box #1:  Simple Child Thread \*/**  **#include <pthread.h>**  **#include <stdio.h>**  **void \*ChildThread(void \*argument)**  **{**  **int i;**  **for ( i = 1; i <= 20; ++i ){**  **printf(" Child Count - %d\n", i);**  **}     pthread\_exit(NULL);**  **}  int main(void)**  **{**  **pthread\_t   hThread;        int   ret;**  **ret=pthread\_create(&hThread, NULL, (void \*)ChildThread, NULL); /\* Create**  **Thread \*/**  **if (ret < 0)**  **printf("Thread Creation Failed\n");   return 1;**  **pthread\_join (hThread, NULL);  /\* Parent waits for  \*/**  **printf("Parent is continuing....\n");**  **return 0;**  **}** |

**Compile and execute the Box #1 program and show the output and explain why the output is so?**

**Code:**

****

**Output:**

****

**Output Explanation:**

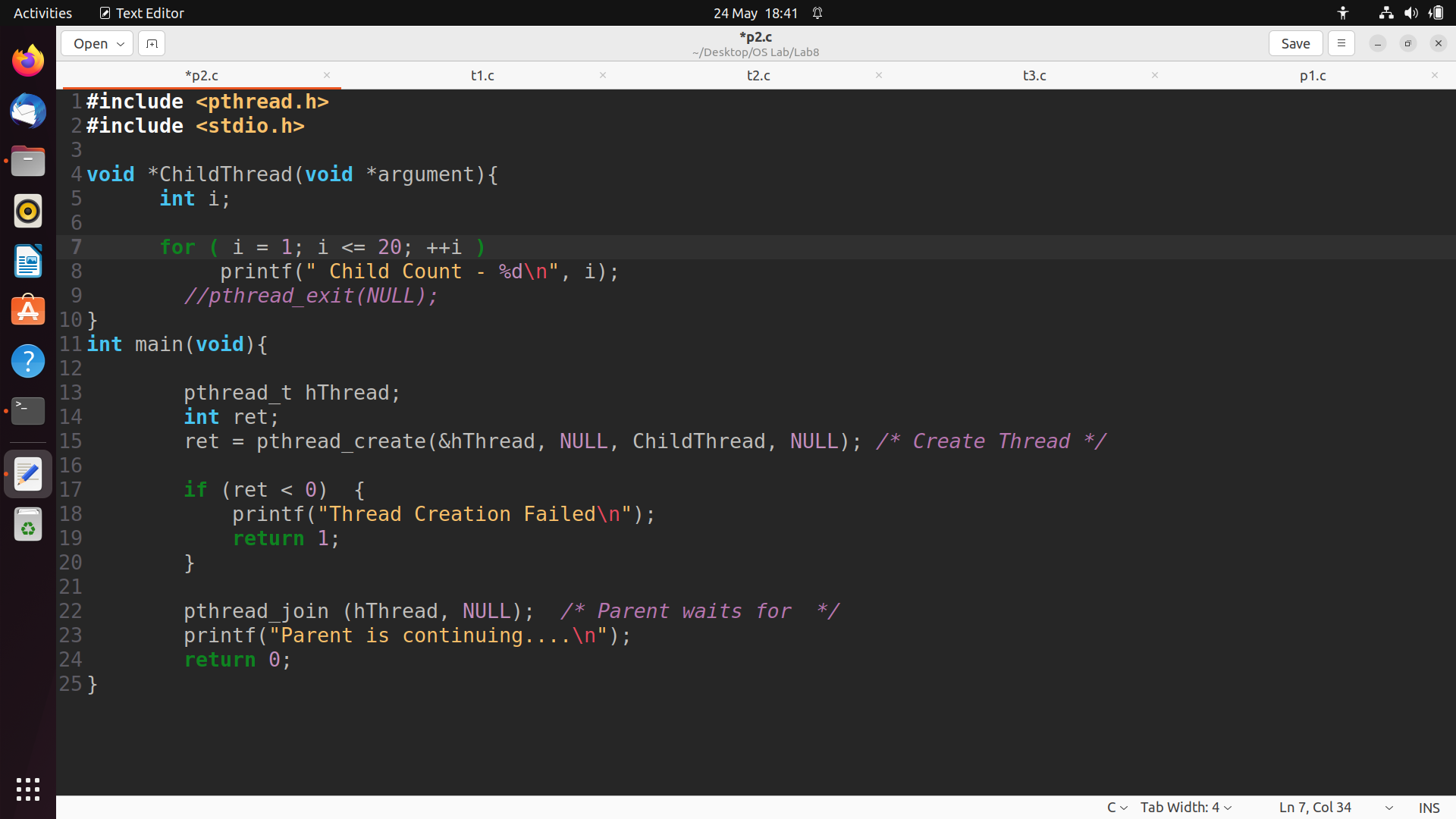
The child thread executes the ChildThread function, which runs a loop from 1 to 20 and prints each number as "Child Count - [number]". Meanwhile, the main thread creates the child thread using pthread\_create. pthread\_join blocks the main thread until the child thread finishes executing. Once the child thread completes, the main thread continues execution and prints "Parent is continuing..."

**Problem#2:**

**Write a program Box # 2 by removing pthread\_exit function from child thread function and check the output? Is it the same as output when pthread\_exit is used? If so Why? Explain?**

|  |
| --- |
| **/\* Box # 2: Implicit Thread Exit  \*/**  **#include <pthread.h>**  **#include <stdio.h>**  **void ChildThread (int argument)**  **{**  **int i;**  **...............................**  **/\* No pthread\_exit function \*/**  **}  int main(void)**  **{**  **pthread\_t   hThread;**  **pthread\_create (...........................................);**  **pthread\_join (hThread, NULL);**  **printf ("Master thread is continuing....\n");        return 0;**  **}** |

**Code:**



**Output:**

**A screenshot of a computer

Description automatically generated**

**Output Explanation:**

The output of this program will be the same as the Box #1 program, even though the pthread\_exit function is removed. This is because when the child thread reaches the end of its execution (end of the ChildThread function), it implicitly exits. In this case, the thread function simply returns without explicitly calling pthread\_exit.