**Pthread\_Create() Function in C Language**

Conceptually, a program is a single thread that executes several serial tasks, one after the other.  
The C language allows us to write multithreaded programs using the pthread library of the POSIX standard.  
A multi-threaded program is a thread or task that runs in parallel and concurrently with the main program and other threads opened by the same program in some part of it.

In this **Linux Hint** article, you will learn how to create a thread from a main program using the **pthread\_create()** function of the pthread library.

This explains the theoretical workings of this function, its syntax, input and output arguments, and the data type accepted in each case.  
We will then apply what we have learned in practical examples, including code snippets and images, in which we create and synchronize threads from the main function.

## Syntax of the pthread\_create() Function in C Language

int pthread\_create(pthread\_t \*restrict thread,  
                   const pthread\_attr\_t \*restrict attr,  
                   void \*(\*start\_routine)(void \*),  
                   void \*restrict arg);

## Description of the pthread\_create() Function in C Language

The **pthread\_create()** function creates a thread and executes it, in parallel and concurrently with the program that created it. This function executes the routine specified by its pointer in the input **start\_routine** by passing it the input argument **arg**.

Next, let us look at the input and output arguments of **pthread\_create()** in detail, as well as a description of the work each of these arguments performs within the function.

**thread**: This is an output argument that returns the identifier of the created thread. This identifier is used to reference it in certain thread management functions such as pthread\_join() or pthread\_cancel().

**attr**: This entry is the pointer to a structure of type pthread\_attr\_t whose members specify the attributes of the new thread. When this argument is sent to NULL, the attributes of the new thread are taken with their default parameters.

**start\_routine**: This is the pointer to the function that will be executed by the new thread.

**arg**: This is the pointer to the argument that the main function passes to the new thread.

If the thread is created successfully, **pthread\_create()** returns 0 as the result. If an error occurs, it returns -1 and stores in the global variable **errno** the specific numeric value representing that error.

The **pthread\_create()** function is defined in the pthread.h header. To use it, the following headers must be included in the “.c” file, as shown below:

#include <unistd.h>  
#include <pthread.h>

## Compilation Errors in Programs with Threads

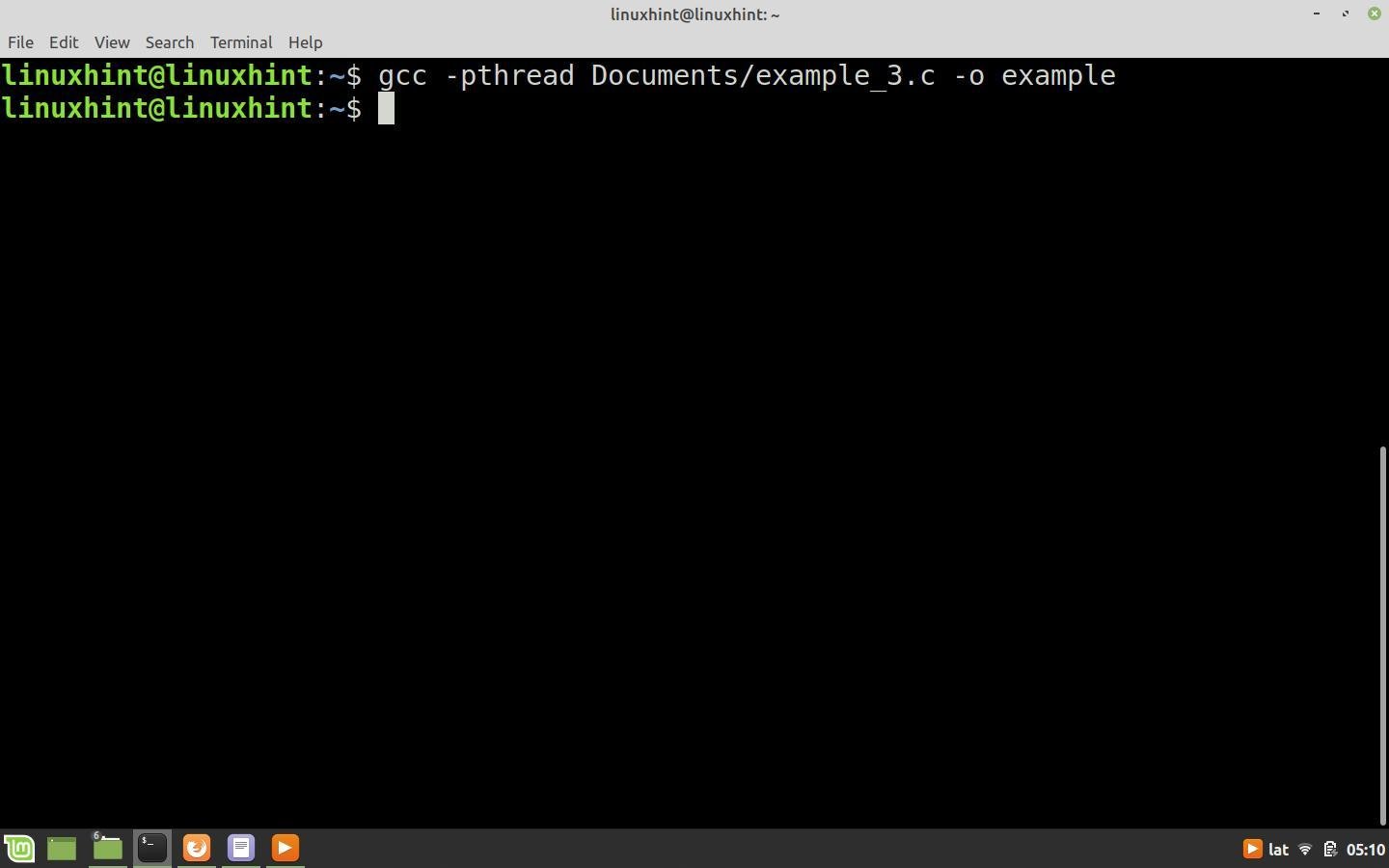
When compiling GCC programs that use threads, compilation may fail if not done correctly from the command line.  
The most common error message issued by the compiler states that one of the thread functions we refer to in the code is not defined.

These errors often result in wasting valuable time checking the headers we have inserted in the code, their integrity, and the directories associated with the compiler, since everything indicates that the problem is there.  
Although the functions that cause the error are defined in the “pthread.h” header and included in the code, the compiler does not recognize these functions unless the pthread library is called from the command line during compilation.

Below, you can see in green the correct way to call the pthread library from the command console during compilation of programs with threads:

~$ gcc -pthread path/filename.c -o out\_name

As shown in the figure below, the error disappears when the pthread library is called during compilation.



## How to Create and Execute a Thread with the pthread\_create() Function in the C Language

In this example, we will explain how **pthread\_create()** works. To do this, we will create a main function and from there open a thread that executes the **thread\_function()** function in parallel with the **main() function.**

The code for this example consists of two sections, the **main()** function and the **thread\_function()** function, which is the thread.  
Next, we will explain each of the two functions separately and then put them together to compile and run the code.

**thread\_function()**: This function consists of a **for** loop with 5 cycles, in each of which the message “From the thread” is displayed in the command console and after a delay of 3 seconds the cycle is repeated again. The message of this function is interleaved with that of the **main()** function so that you can see in real time what each of the processes is doing simultaneously.

Next, we ncl see the nclusi the **thread\_function()** function and the definition of its prototype:

void\* thread\_function(void\* n);  
  
void\* thread\_function(void\* n)  
{  
  for (int a=0;a!=5; a++)  
      {  
       [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)(“From the thread \n”);  
       sleep(3);  
      }  
  [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)(“nclus thread \n”);  
  pthread\_exit(n);  
  
}

**main() function**: In this example, the **main()** function is nclusióne for defining the variables and creating the thread.

The first step in creating a thread is to define a variable of type **pthread\_t** that ncl serve as the identifier of the thread. In this example we ncl call this variable **thread\_1.**

To nclus the thread, we call the **pthread\_create()** function and pass the thread identifier thread\_1 as the first argument.  
The attributes of the thread to be created are preset in this case, so the second input argument is NULL.  
As the third argument, we pass the pointer to the function to be executed in the new thread, in this case **thread\_function().**  
Since we do not need to pass any arguments to the new thread in this example, the arg pointer ncl also be NULL.

After calling **pthread\_create()**, the new thread begins execution and the function **main()** enters a **for** loop of 5 cycles, each of which prints the message “From main function” nested with the message “From the thread” of the function **thread\_function()**. After each message, a delay of 3 seconds is inserted into the for loops of both functions before a new cycle is started.

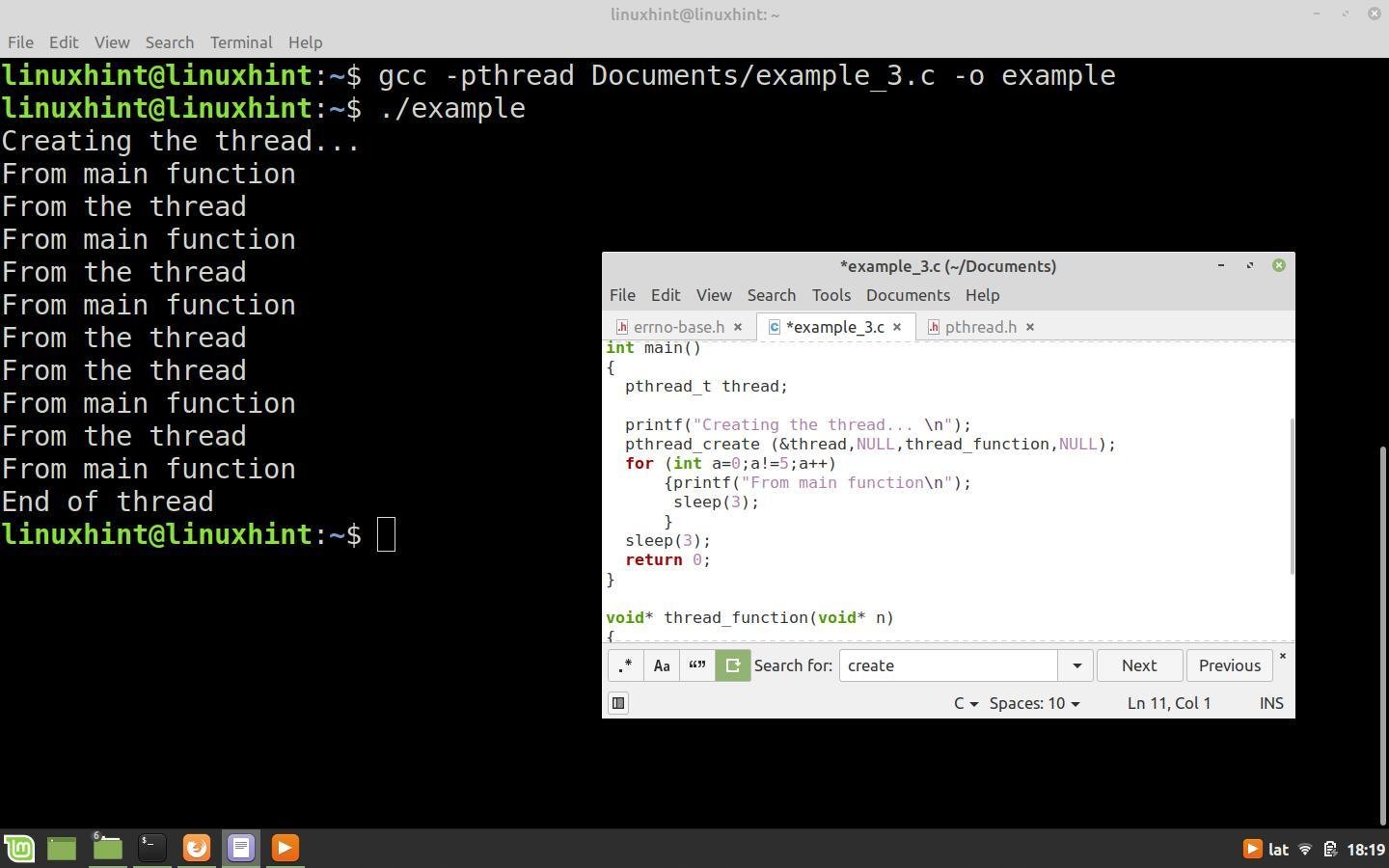
Below you can see the nclusi the function **main(),** the nclusión of the necessary headers, and the declaration of the prototype of the function **thread\_function():**

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <pthread.h>  
  
void\* thread\_function(void\* data);  
  
int main()  
{  
  pthread\_t thread\_1;  
  [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Creating the thread... \n");  
  pthread\_create (&thread\_1,NULL,thread\_function,NULL);  
  for (int a=0;a!=5;a++)  
    {  
      [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("From main function \n");  
    sleep(3);  
    }  
  sleep(3);  
  return 0;  
}

Next, we see the full code for this example. Copy and paste this fragment into a file with a “.c” extension. Compile the code and run it to see how the **main()** function and the new thread perform their tasks simultaneously.

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <pthread.h>  
  
void\* thread\_function(void\* data);  
  
int main()  
{  
  pthread\_t thread\_1;  
  [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Creating the thread... \n");  
  pthread\_create (&thread\_1,NULL,thread\_function,NULL);  
  for (int a=0;a!=5;a++)  
    {  
      [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("From main function \n");  
    sleep(3);  
    }  
  
  return 0;  
}  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
void\* thread\_function(void\* n)  
{    
  for (int a=0;a!=5; a++)  
      {  
       [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("From the thread \n");  
       sleep(3);  
      }  
  [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("End of thread \n");  
  pthread\_exit(n);  
  
}

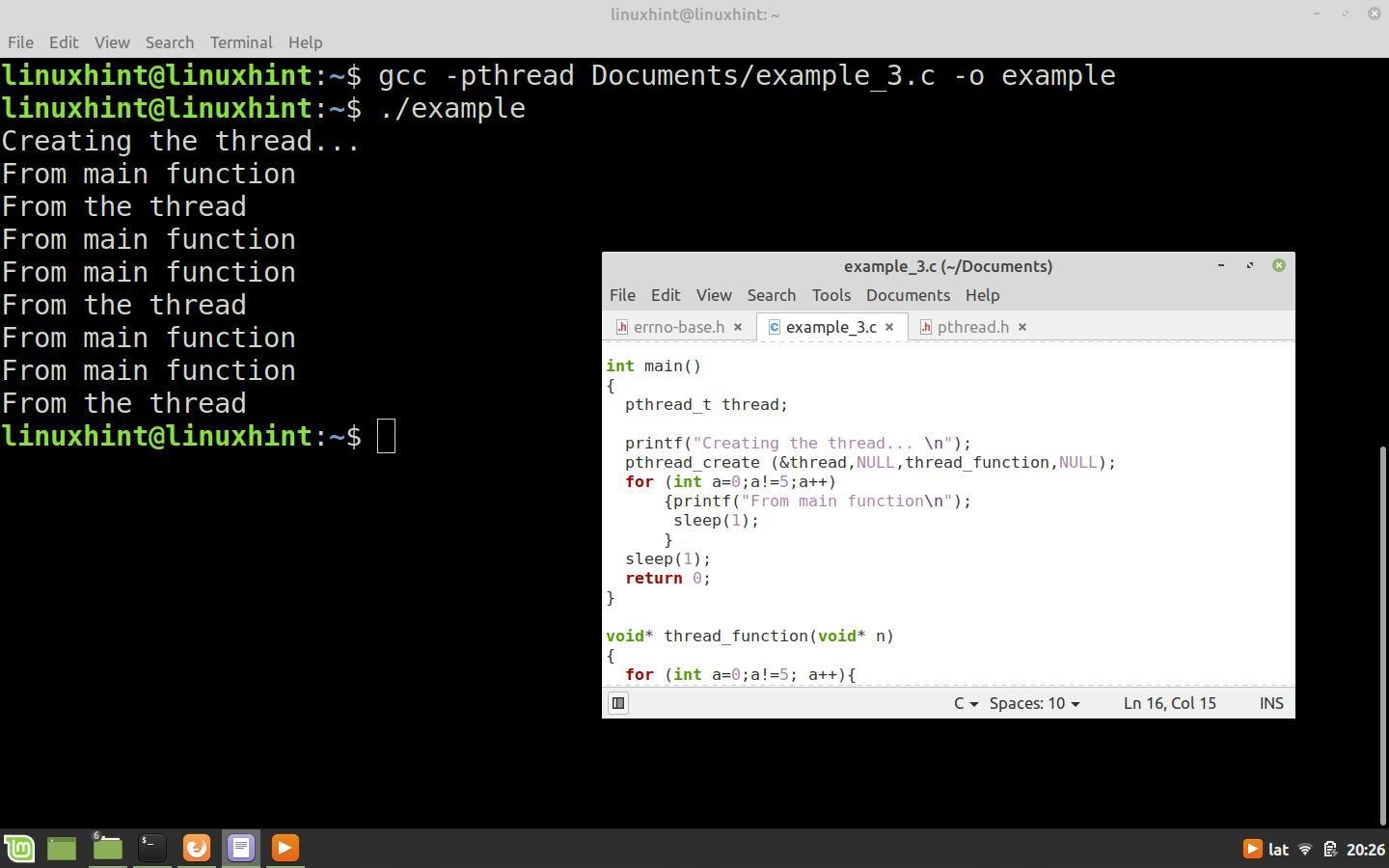
The image we see below shows the task executed by the thread we created with the **pthread\_create()** function in parallel with the task of the **main()** function, and the messages each of them sends to the command console:



## Synchronization of Threads in the C Language

In the code of the previous example, the execution time of the thread is shorter than that of the **main()** function and therefore both do their job correctly. This is due to the delay of 3 seconds that occurs when the **main()** function exits the loop.  
However, if the execution times of a thread are longer than those of the **main()** function and it is fully executed, the program ends and all threads that were created and are still performing a task are automatically closed.

Let us see what happens in the code of the previous example if we set a delay of 1 second per cycle in the for loop of the **main()** function and one of 3 seconds in the thread loop.



As shown in the picture, the **main()** function completed the 5 cycles of its **for** loop, while the thread could only execute 3 cycles before the program closed.

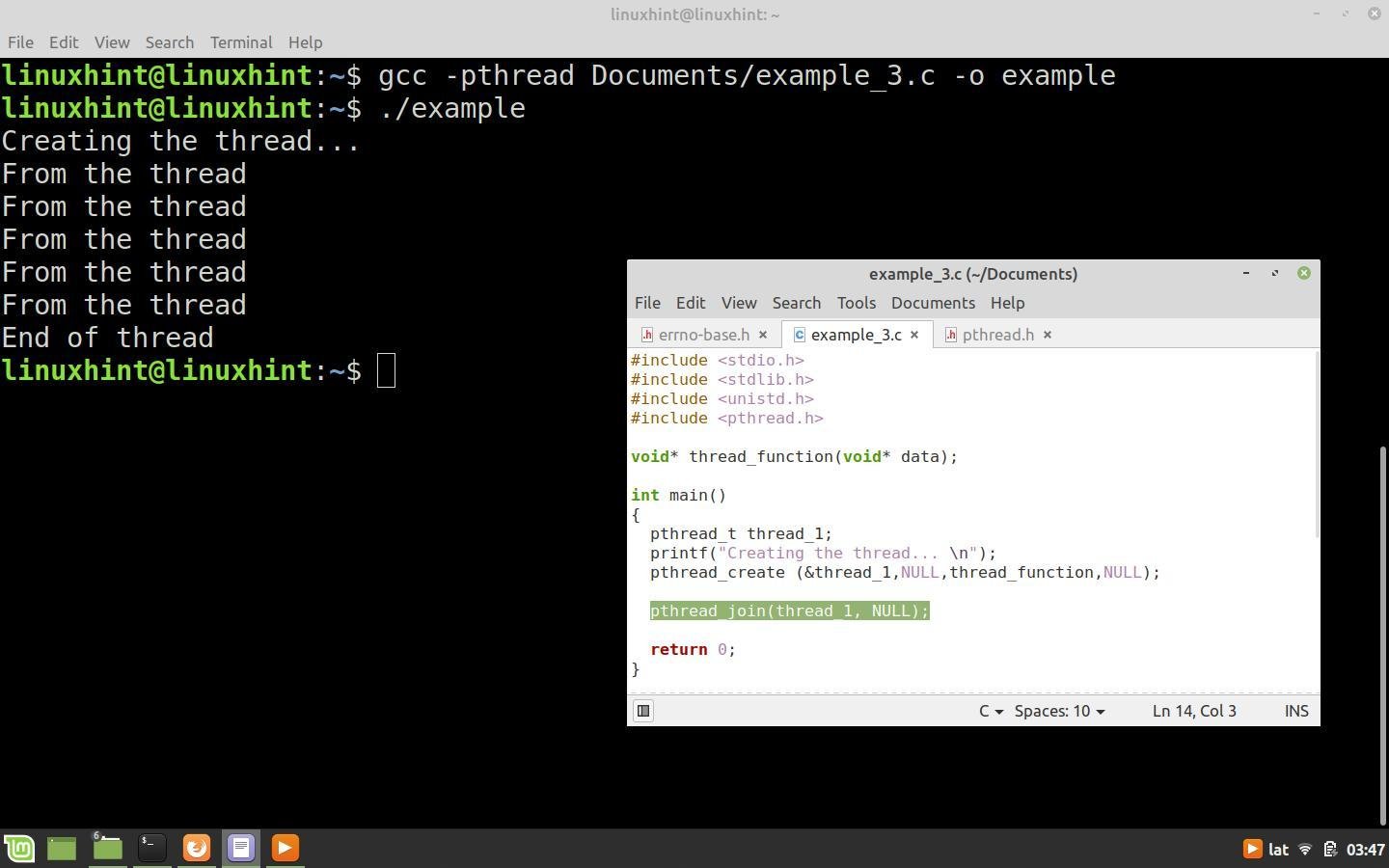
Closing a thread without fully executing it can lead to critical problems in certain cases, as it could be storing user-generated data, writing to a file system or storage device, or performing some other task at the time of termination.  
To avoid these problems, it is important to develop a mechanism to “wait” for the thread to complete execution before closing the program or executing a task. The **pthread\_join()** function stops the function that created the thread until that thread finishes its execution.

The **pthread\_join()** function takes two input arguments. The first is the thread identifier returned by the **pthread\_create()** function when the thread is created and the second argument is the pointer to a variable that returns the exit status of the thread.

Next, we use the code from the previous example, but replace the **for** loop of the **main()** function with the **pthread\_join()** function, as shown below:

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <pthread.h>  
  
void\* thread\_function(void\* data);  
  
int main()  
{  
  pthread\_t thread\_1;  
  [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Creating the thread... \n");  
  pthread\_create (&thread\_1,NULL,thread\_function,NULL);  
    
  pthread\_join (thread\_1, NULL);  
  
  return 0;  
}  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
void\* thread\_function(void\* n)  
{    
  for (int a=0;a!=5; a++)  
      {  
       [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("From the thread \n");  
       sleep(3);  
      }  
  [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("End of thread \n");  
  pthread\_exit(n);  
  
}

In this case, the **main()** function will just create the thread and wait for it to finish its task, then close the program.



## Possible Errors that the pthread\_create() Function Can Generate and How to Recognize Them

The **pthread\_create()** function can generate various errors, from invalid attribute settings to insufficient system resources for the new thread.  
When a function generates an error, a numeric error identification code is stored in the global variable **errno**. This variable and the numeric definitions for each error are defined in the “errno.h” header.  
Below is a description of each error that can occur when calling the **pthread\_create()** function and its numeric representation defined in “errno.h”.

**EAGAIN**: There are no resources available to create another thread. The numeric representation of this error is 35.

**EINVAL**: The attribute configuration in attr is not valid. The numeric representation of this error is 22.

**EPERM**: Operation not allowed. This error occurs when you do not have sufficient permissions to set the attribute parameters in attr. The numerical representation of this error is 1.

## Conclusion

In this **Linux Hint** article, we showed you how to use the **pthread\_create()** function to create multitasking programs with threads that run in parallel with the main function.

We also told you how to compile programs that use threads correctly from the command line.  
We have also included a special section explaining the importance of taking into account the execution times of the created threads and we have taught you how to synchronize the threads with the main function to achieve their correct execution.

**Thread synchronization** is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes’ access to critical section is controlled by using synchronization techniques. When one thread starts executing the [critical section](https://www.geeksforgeeks.org/g-fact-70/) (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a [race condition](https://practice.geeksforgeeks.org/problems/what-is-race-condition) where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

**Thread Synchronization Problems**  
An example code to study synchronization problems :

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>    pthread\_t tid[2];  int counter;    void\* trythis(void\* arg)  {      unsigned long i = 0;      counter += 1;      printf("\n Job %d has started\n", counter);        for (i = 0; i < (0xFFFFFFFF); i++)          ;      printf("\n Job %d has finished\n", counter);        return NULL;  }    int main(void)  {      int i = 0;      int error;        while (i < 2) {          error = pthread\_create(&(tid[i]), NULL, &trythis, NULL);          if (error != 0)              printf("\nThread can't be created : [%s]", strerror(error));          i++;      }        pthread\_join(tid[0], NULL);      pthread\_join(tid[1], NULL);        return 0;  } |

**How to compile above program?**  
To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

gfg@ubuntu:~/$ gcc filename.c -lpthread

In this example, two threads(jobs) are created and in the start function of these threads, a counter is maintained to get the logs about job number which is started and when it is completed.

Output :

Job 1 has started

Job 2 has started

Job 2 has finished

Job 2 has finished

**Problem:** From the last two logs, one can see that the log ‘Job 2 has finished’ is repeated twice while no log for ‘Job 1 has finished’ is seen.

**Why it has occurred ?**  
On observing closely and visualizing the execution of the code, we can see that :

* The log ‘Job 2 has started’ is printed just after ‘Job 1 has Started’ so it can easily be concluded that while thread 1 was processing the scheduler scheduled the thread 2.
* If we take the above assumption true then the value of the ‘counter’ variable got incremented again before job 1 got finished.
* So, when Job 1 actually got finished, then the wrong value of counter produced the log ‘Job 2 has finished’ followed by the ‘Job 2 has finished’ for the actual job 2 or vice versa as it is dependent on scheduler.
* So we see that its not the repetitive log but the wrong value of the ‘counter’ variable that is the problem.
* The actual problem was the usage of the variable ‘counter’ by a second thread when the first thread was using or about to use it.
* In other words, we can say that lack of synchronization between the threads while using the shared resource ‘counter’ caused the problems or in one word we can say that this problem happened due to ‘Synchronization problem’ between two threads.

**How to solve it ?**

The most popular way of achieving thread synchronization is by using **Mutexes**.

### Mutex

* + A Mutex is a lock that we set before using a shared resource and release after using it.
  + When the lock is set, no other thread can access the locked region of code.
  + So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code.
  + So this ensures synchronized access of shared resources in the code.

**Working of a mutex**

* + Suppose one thread has locked a region of code using mutex and is executing that piece of code.
  + Now if scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked.
  + Only one of all the threads would make it to the execution but if this thread tries to execute the same region of code that is already locked then it will again go to sleep.
  + Context switch will take place again and again but no thread would be able to execute the locked region of code until the mutex lock over it is released.
  + Mutex lock will only be released by the thread who locked it.
  + So this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.

Hence, this system ensures synchronization among the threads while working on shared resources.

**A mutex is initialized and then a lock is achieved by calling the following two functions :** The first function initializes a mutex and through second function any critical region in the code can be locked.

* + 1. **int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr) :** Creates a mutex, referenced by mutex, with attributes specified by attr. If attr is NULL, the default mutex attribute (NONRECURSIVE) is used.

**Returned value**  
If successful, pthread\_mutex\_init() returns 0, and the state of the mutex becomes initialized and unlocked.  
If unsuccessful, pthread\_mutex\_init() returns -1.

* + 1. **int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex) :** Locks a mutex object, which identifies a mutex. If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. When the mutex has the attribute of recursive, the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then a count is incremented and no waiting thread is posted. The owning thread must call pthread\_mutex\_unlock() the same number of times to decrement the count to zero.

**Returned value**  
If successful, pthread\_mutex\_lock() returns 0.  
If unsuccessful, pthread\_mutex\_lock() returns -1.

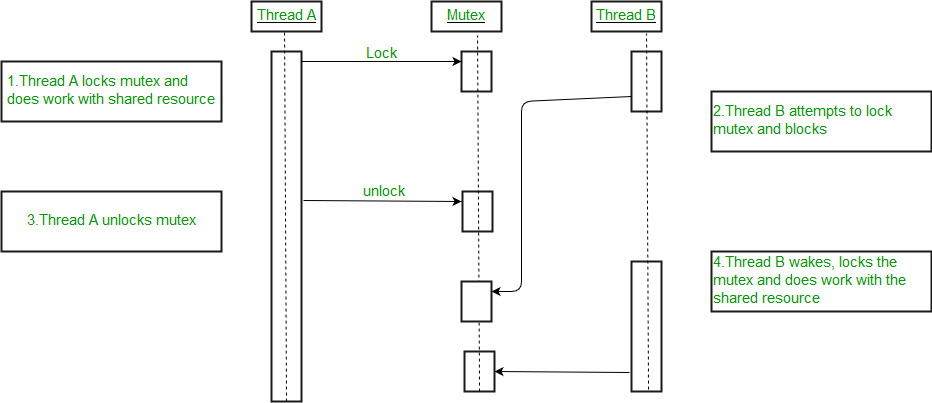
**The mutex can be unlocked and destroyed by calling following two functions :**The first function releases the lock and the second function destroys the lock so that it cannot be used anywhere in future.

* + 1. **int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex) :** Releases a mutex object. If one or more threads are waiting to lock the mutex, pthread\_mutex\_unlock() causes one of those threads to return from pthread\_mutex\_lock() with the mutex object acquired. If no threads are waiting for the mutex, the mutex unlocks with no current owner. When the mutex has the attribute of recursive the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then unlock will decrement the count and no waiting thread is posted to continue running with the lock. If the count is decremented to zero, then the mutex is released and if any thread is waiting for it is posted.

**Returned value**  
If successful, pthread\_mutex\_unlock() returns 0.  
If unsuccessful, pthread\_mutex\_unlock() returns -1

* + 1. **int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex) :** Deletes a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. mutex is set to an invalid value, but can be reinitialized using pthread\_mutex\_init().

**Returned value**  
If successful, pthread\_mutex\_destroy() returns 0.  
If unsuccessful, pthread\_mutex\_destroy() returns -1.

  
An example to show how mutexes are used for thread synchronization

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>    pthread\_t tid[2];  int counter;  pthread\_mutex\_t lock;    void\* trythis(void\* arg)  {      pthread\_mutex\_lock(&lock);        unsigned long i = 0;      counter += 1;      printf("\n Job %d has started\n", counter);        for (i = 0; i < (0xFFFFFFFF); i++)          ;        printf("\n Job %d has finished\n", counter);        pthread\_mutex\_unlock(&lock);        return NULL;  }    int main(void)  {      int i = 0;      int error;        if (pthread\_mutex\_init(&lock, NULL) != 0) {          printf("\n mutex init has failed\n");          return 1;      }        while (i < 2) {          error = pthread\_create(&(tid[i]),                                 NULL,                                 &trythis, NULL);          if (error != 0)              printf("\nThread can't be created :[%s]",                     strerror(error));          i++;      }        pthread\_join(tid[0], NULL);      pthread\_join(tid[1], NULL);      pthread\_mutex\_destroy(&lock);        return 0;  } |

In the above code:

* + A mutex is initialized in the beginning of the main function.
  + The same mutex is locked in the ‘trythis()’ function while using the shared resource ‘counter’.
  + At the end of the function ‘trythis()’ the same mutex is unlocked.
  + At the end of the main function when both the threads are done, the mutex is destroyed.

Output :

Job 1 started

Job 1 finished

Job 2 started

Job 2 finished

So this time the start and finish logs of both the jobs are present. So thread synchronization took place by the use of Mutex.

To obtain such a mutual exclusion, bounded waiting, and progress there have been several algorithms implemented, one of which is Dekker’s Algorithm. To understand the algorithm let’s understand the solution to the critical section problem first.   
A process is generally represented as : 

do {

//entry section

critical section

//exit section

remainder section

} while (TRUE);

The solution to the critical section problem must ensure the following three conditions: 

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

One of the solutions for ensuring above all factors is [Peterson’s solution](https://www.geeksforgeeks.org/petersons-algorithm-for-mutual-exclusion-set-1/).  
Another one is **Dekker’s Solution**. Dekker’s algorithm was the first probably-correct solution to the critical section problem. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.  
Although there are many versions of Dekker’s Solution, the final or 5th version is the one that satisfies all of the above conditions and is the most efficient of them all.   
**Note –** Dekker’s Solution, mentioned here, ensures mutual exclusion between two processes only, it could be extended to more than two processes with the proper use of arrays and variables.  
**Algorithm –** It requires both an array of Boolean values and an integer variable:

var flag: array [0..1] of boolean;

turn: 0..1;

repeat

flag[i] := true;

while flag[j] do

if turn = j then

begin

flag[i] := false;

while turn = j do no-op;

flag[i] := true;

end;

critical section

turn := j;

flag[i] := false;

remainder section

until false;

**First Version of Dekker’s Solution –** The idea is to use a common or shared thread number between processes and stop the other process from entering its critical section if the shared thread indicates the former one already running.

|  |
| --- |
| Main()  {        int thread\_number = 1;      startThreads();  }    Thread1()  {      do {            // entry section          // wait until threadnumber is 1          while (threadnumber == 2)              ;            // critical section            // exit section          // give access to the other thread          threadnumber = 2;            // remainder section        } while (completed == false)  }    Thread2()  {        do {            // entry section          // wait until threadnumber is 2          while (threadnumber == 1)              ;            // critical section            // exit section          // give access to the other thread          threadnumber = 1;            // remainder section        } while (completed == false)  } |

The problem arising in the above implementation is lockstep synchronization, i.e each thread depends on the other for its execution. If one of the processes completes, then the second process runs, gives access to the completed one, and waits for its turn, however, the former process is already completed and would never run to return the access back to the latter one. Hence, the second process waits infinitely then.  
**Second Version of Dekker’s Solution –** To remove lockstep synchronization, it uses two flags to indicate its current status and updates them accordingly at the entry and exit section.

|  |
| --- |
| Main()  {        // flags to indicate if each thread is in      // its critical section or not.      boolean thread1 = false;      boolean thread2 = false;        startThreads();  }    Thread1()  {        do {            // entry section          // wait until thread2 is in its critical section          while (thread2 == true)              ;            // indicate thread1 entering its critical section          thread1 = true;            // critical section            // exit section          // indicate thread1 exiting its critical section          thread1 = false;            // remainder section        } while (completed == false)  }    Thread2()  {        do {            // entry section          // wait until thread1 is in its critical section          while (thread1 == true)              ;            // indicate thread2 entering its critical section          thread2 = true;            // critical section            // exit section          // indicate thread2 exiting its critical section          thread2 = false;            // remainder section        } while (completed == false)  } |

The problem arising in the above version is mutual exclusion itself. If threads are preempted (stopped) during flag updation ( i.e during current\_thread = true ) then, both the threads enter their critical section once the preempted thread is restarted, also the same can be observed at the start itself, when both the flags are false.  
**Third Version of Dekker’s Solution –** To re-ensure mutual exclusion, it sets the flags before the entry section itself.

|  |
| --- |
| Main()  {        // flags to indicate if each thread is in      // queue to enter its critical section      boolean thread1wantstoenter = false;      boolean thread2wantstoenter = false;        startThreads();  }    Thread1()  {        do {            thread1wantstoenter = true;            // entry section          // wait until thread2 wants to enter          // its critical section          while (thread2wantstoenter == true)              ;            // critical section            // exit section          // indicate thread1 has completed          // its critical section          thread1wantstoenter = false;            // remainder section        } while (completed == false)  }    Thread2()  {        do {            thread2wantstoenter = true;            // entry section          // wait until thread1 wants to enter          // its critical section          while (thread1wantstoenter == true)              ;            // critical section            // exit section          // indicate thread2 has completed          // its critical section          thread2wantstoenter = false;            // remainder section        } while (completed == false)  } |

The problem with this version is a deadlock possibility. Both threads could set their flag as true simultaneously and both will wait infinitely later on.  
**Fourth Version of Dekker’s Solution –** Uses small time interval to recheck the condition, eliminates deadlock, and ensures mutual exclusion as well.

|  |
| --- |
| Main()  {        // flags to indicate if each thread is in      // queue to enter its critical section      boolean thread1wantstoenter = false;      boolean thread2wantstoenter = false;        startThreads();  }    Thread1()  {        do {            thread1wantstoenter = true;            while (thread2wantstoenter == true) {                // gives access to other thread              // wait for random amount of time              thread1wantstoenter = false;                thread1wantstoenter = true;          }            // entry section          // wait until thread2 wants to enter          // its critical section            // critical section            // exit section          // indicate thread1 has completed          // its critical section          thread1wantstoenter = false;            // remainder section        } while (completed == false)  }    Thread2()  {        do {            thread2wantstoenter = true;            while (thread1wantstoenter == true) {                // gives access to other thread              // wait for random amount of time              thread2wantstoenter = false;                thread2wantstoenter = true;          }            // entry section          // wait until thread1 wants to enter          // its critical section            // critical section            // exit section          // indicate thread2 has completed          // its critical section          thread2wantstoenter = false;            // remainder section        } while (completed == false)  } |

The problem with this version is the indefinite postponement. Also, a random amount of time is erratic depending upon the situation in which the algorithm is being implemented, hence not an acceptable solution in business critical systems.  
**Dekker’s Algorithm: Final and completed Solution –** -Idea is to use favoured thread notion to determine entry to the critical section. Favoured thread alternates between the thread providing mutual exclusion and avoiding deadlock, indefinite postponement, or lockstep synchronization.

|  |
| --- |
| Main()  {        // to denote which thread will enter next      int favouredthread = 1;        // flags to indicate if each thread is in      // queue to enter its critical section      boolean thread1wantstoenter = false;      boolean thread2wantstoenter = false;        startThreads();  }    Thread1()  {      do {            thread1wantstoenter = true;            // entry section          // wait until thread2 wants to enter          // its critical section          while (thread2wantstoenter == true) {                // if 2nd thread is more favored              if (favaouredthread == 2) {                    // gives access to other thread                  thread1wantstoenter = false;                    // wait until this thread is favored                  while (favouredthread == 2)                      ;                    thread1wantstoenter = true;              }          }            // critical section            // favor the 2nd thread          favouredthread = 2;            // exit section          // indicate thread1 has completed          // its critical section          thread1wantstoenter = false;            // remainder section        } while (completed == false)  }    Thread2()  {        do {            thread2wantstoenter = true;            // entry section          // wait until thread1 wants to enter          // its critical section          while (thread1wantstoenter == true) {                // if 1st thread is more favored              if (favaouredthread == 1) {                    // gives access to other thread                  thread2wantstoenter = false;                    // wait until this thread is favored                  while (favouredthread == 1)                      ;                    thread2wantstoenter = true;              }          }            // critical section            // favour the 1st thread          favouredthread = 1;            // exit section          // indicate thread2 has completed          // its critical section          thread2wantstoenter = false;            // remainder section        } while (completed == false)  } |

This version guarantees a complete solution to the critical solution problem.

The basic code of a semaphore is simple. But this code cannot be written directly, as the functions require to be atomic and writing code directly would lead to a context switch without function completion and would result in a mess.

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to :

1. Include semaphore.h
2. Compile the code by linking with -lpthread -lrt

To lock a semaphore or wait we can use the **sem\_wait** function:

int sem\_wait(sem\_t \*sem);

To release or signal a semaphore, we use the **sem\_post** function:

int sem\_post(sem\_t \*sem);

A semaphore is initialised by using **sem\_init**(for processes or threads) or **sem\_open** (for IPC).

sem\_init(sem\_t \*sem, int pshared, unsigned int value);

Where,

* + **sem** : Specifies the semaphore to be initialized.
  + **pshared** : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
  + **value** : Specifies the value to assign to the newly initialized semaphore.

To destroy a semaphore, we can use **sem\_destroy**.

sem\_destroy(sem\_t \*mutex);

To declare a semaphore, the data type is sem\_t.

**Code –**

|  |
| --- |
| // C program to demonstrate working of Semaphores  #include <stdio.h>  #include <pthread.h>  #include <semaphore.h>  #include <unistd.h>    sem\_t mutex;    void\* thread(void\* arg)  {      //wait      sem\_wait(&mutex);      printf("\nEntered..\n");        //critical section      sleep(4);        //signal      printf("\nJust Exiting...\n");      sem\_post(&mutex);  }      int main()  {      sem\_init(&mutex, 0, 1);      pthread\_t t1,t2;      pthread\_create(&t1,NULL,thread,NULL);      sleep(2);      pthread\_create(&t2,NULL,thread,NULL);      pthread\_join(t1,NULL);      pthread\_join(t2,NULL);      sem\_destroy(&mutex);      return 0;  } |

Compilation should be done with gcc a.c -lpthread -lrt

**Explanation –**  
2 threads are being created, one 2 seconds after the first one.  
But the first thread will sleep for 4 seconds after acquiring the lock.  
Thus the second thread will not enter immediately after it is called, it will enter 4 – 2 = 2 secs after it is called.  
So the output is:

Entered..

Just Exiting...

Entered..

Just Exiting...

but not:

Entered..

Entered..

Just Exiting...

Just Exiting...

Semaphores are used to synchronize operations between two or more processes. POSIX defines two different sets of semaphore functions:

1. 'System V IPC' — [semctl()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/semctl.html), [semop()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/semop.html), [semget()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/semget.html).
2. 'POSIX Semaphores' — [sem\_close()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_close.html), [sem\_destroy()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_destroy.html), [sem\_getvalue()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_getvalue.html), [sem\_init()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_init.html), [sem\_open()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_open.html), [sem\_post()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_post.html), [sem\_trywait()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_trywait.html), [sem\_unlink()](http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_unlink.html).

This section describes the System V IPC semaphores, so called because they originated with Unix System V.

First, you'll need to include the required headers. Old versions of POSIX required #include <sys/types.h>; modern POSIX and most systems do not require it.

#include <sys/sem.h>

Then, you'll need to define a key in both the parent as well as the child.

#define KEY 0x1111

This key needs to be the same in both programs or they will not refer to the same IPC structure. There are ways to generate an agreed key without hard-coding its value.

Next, depending on your compiler, you may or may not need to do this step: declare a union for the purpose of semaphore operations.

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

Next, define your try (semwait) and raise (semsignal) structures. The names P and V originate from [Dutch](https://en.wikipedia.org/wiki/Semaphore_(programming)#Operation_names)

struct sembuf p = { 0, -1, SEM\_UNDO}; # semwait

struct sembuf v = { 0, +1, SEM\_UNDO}; # semsignal

Now, start by getting the id for your IPC semaphore.

int id;

// 2nd argument is number of semaphores

// 3rd argument is the mode (IPC\_CREAT creates the semaphore set if needed)

if ((id = semget(KEY, 1, 0666 | IPC\_CREAT) < 0) {

/\* error handling code \*/

}

In the parent, initialise the semaphore to have a counter of 1.

union semun u;

u.val = 1;

if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL is a macro to specify that you're setting the value of the semaphore to that specified by the union u

/\* error handling code \*/

}

Now, you can decrement or increment the semaphore as you need. At the start of your critical section, you decrement the counter using the semop() function:

if (semop(id, &p, 1) < 0) {

/\* error handling code \*/

}

To increment the semaphore, you use &v instead of &p:

if (semop(id, &v, 1) < 0) {

/\* error handling code \*/

}

Note that every function returns 0 on success and -1 on failure. Not checking these return statuses can cause devastating problems.

### Example 1.1: Racing with Threads[#](https://riptutorial.com/c/example/31715/semaphores#undefined)

The below program will have a process fork a child and both parent and child attempt to print characters onto the terminal without any synchronization.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int main()

{

int pid;

pid = fork();

srand(pid);

if(pid < 0)

{

perror("fork"); exit(1);

}

else if(pid)

{

char \*s = "abcdefgh";

int l = strlen(s);

for(int i = 0; i < l; ++i)

{

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

}

}

else

{

char \*s = "ABCDEFGH";

int l = strlen(s);

for(int i = 0; i < l; ++i)

{

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

}

}

}

Output (1st run):

aAABaBCbCbDDcEEcddeFFGGHHeffgghh

(2nd run):

aabbccAABddBCeeCffgDDghEEhFFGGHH

Compiling and running this program should give you a different output each time .

### Example 1.2: Avoid Racing with Semaphores[#](https://riptutorial.com/c/example/31715/semaphores#undefined)

Modifying Example 1.1 to use semaphores, we have:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#define KEY 0x1111

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

struct sembuf p = { 0, -1, SEM\_UNDO};

struct sembuf v = { 0, +1, SEM\_UNDO};

int main()

{

int id = semget(KEY, 1, 0666 | IPC\_CREAT);

if(id < 0)

{

perror("semget"); exit(11);

}

union semun u;

u.val = 1;

if(semctl(id, 0, SETVAL, u) < 0)

{

perror("semctl"); exit(12);

}

int pid;

pid = fork();

srand(pid);

if(pid < 0)

{

perror("fork"); exit(1);

}

else if(pid)

{

char \*s = "abcdefgh";

int l = strlen(s);

for(int i = 0; i < l; ++i)

{

if(semop(id, &p, 1) < 0)

{

perror("semop p"); exit(13);

}

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

putchar(s[i]);

fflush(stdout);

if(semop(id, &v, 1) < 0)

{

perror("semop p"); exit(14);

}

sleep(rand() % 2);

}

}

else

{

char \*s = "ABCDEFGH";

int l = strlen(s);

for(int i = 0; i < l; ++i)

{

if(semop(id, &p, 1) < 0)

{

perror("semop p"); exit(15);

}

putchar(s[i]);

fflush(stdout);

sleep(rand() % 2);

putchar(s[i]);

fflush(stdout);

if(semop(id, &v, 1) < 0)

{

perror("semop p"); exit(16);

}

sleep(rand() % 2);

}

}

}

Output:

aabbAABBCCccddeeDDffEEFFGGHHgghh

Compiling and running this program will give you the same output each time.

#include <stdio.h>

#include <pthread.h>

#define MAX 16

#define THREAD\_MAX 4

int array[MAX] = { 1, 5, 7, 10, 12, 14, 15, 18, 20, 22, 25, 27, 30, 64, 110, 220 };

int key = 18;

int flag = 0; //flag to indicate that item is found in the array or not

int current\_thread = 0;

void\* ThreadSearch(void\* args) { //This is linear search function. It will be running using all threads

   int num = current\_thread++;

   for (int i = num \* (MAX / 4); i < ((num + 1) \* (MAX / 4)); i++){

      if (array[i] == key)

         flag = 1; //set flag if key is found

   }

}

int main() {

   pthread\_t thread[THREAD\_MAX];

   for (int i = 0; i < THREAD\_MAX; i++) { //create multiple threads

      pthread\_create(&thread[i], NULL, ThreadSearch, (void\*)NULL);

   }

   for (int i = 0; i < THREAD\_MAX; i++) {

      pthread\_join(thread[i], NULL); //wait untill all of the threads are completed

   }

   if (flag == 1)

      printf("Key element is found  
");

   else

      printf("Key element is not present  
");

}

## Output

$ gcc 1249.Thread\_search.cpp -lpthread

$ ./a.out

Key element is found