SECURE CODING ASSIGNMENT 15

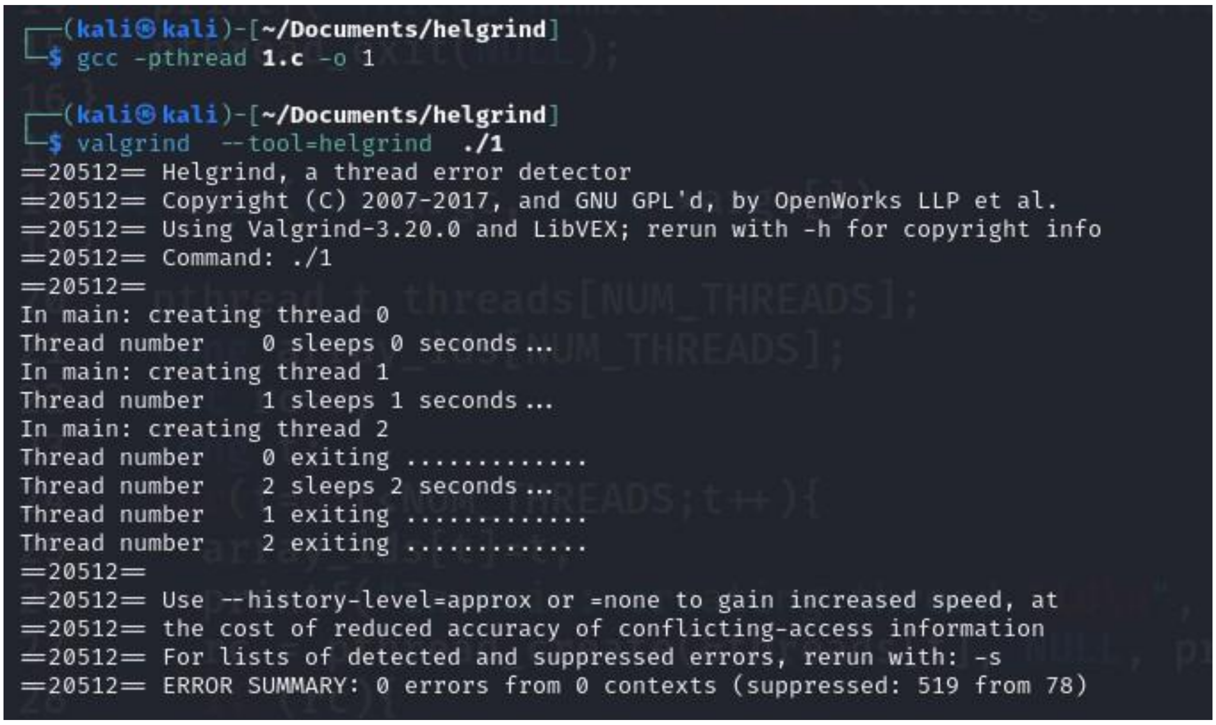
HELGRIND TOOL

Helgrind is a Valgrind tool for detecting synchronisation errors in C/C++ programs that use the POSIXthreading primitives. The main abstractions in POSIX are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables and barriers. Helgrind detects three types of errors: (1) misuses of the POSIX API, (2) potential deadlocks arising from lock ordering problems, and (3) data races.

The following code includes an example with a multithreaded code with several (NUM_THREADS) threads created and deleted. Each thread prints out a text and the main waits for the end of the remaining threads.

```
//compilation in linux with gcc -pthread option #include <pthread.h>
#include <stdio.h> #include <stdlib.h> #include <unistd.h>
#define NUM_THREADS 3
void *print_hello(void *threadid)
{
long tid;
tid = (long)threadid;
printf("Thread number \t %ld sleeps %ld seconds...\n",tid,tid); sleep(tid);
printf("Thread number \t %ld exiting  \n",tid);
pthread_exit(NULL);
}


int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS]; long array_ids[NUM_THREADS];

int rc=0; long t;
for(t=0;t<NUM_THREADS;t++){
array_ids[t]=t;
printf("In main: creating thread %ld\n", array_ids[t]);
rc = pthread_create(&threads[t], NULL, print_hello, (void *)t); if (rc){
printf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);
}
}
for (t=0;t<NUM_THREADS;t++){
pthread_join(threads[t],NULL);
}
return 0;
}
```

To compile the code, you need the -pthread option of the (gcc) compiler. The compiled code is thechecked by Helgrind expliciting the following option: --tool=helgrind.



```
┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ gcc -pthread 1.c -o 1

┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ valgrind  --tool=helgrind  ./1
==20512== Helgrind, a thread error detector
==20512== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==20512== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==20512== Command: ./1
==20512==
In main: creating thread 0
Thread number    0 sleeps 0 seconds ...
In main: creating thread 1
Thread number    1 sleeps 1 seconds ...
In main: creating thread 2
Thread number    0 exiting ............
Thread number    2 sleeps 2 seconds ...
Thread number    1 exiting ............
Thread number    2 exiting ............
==20512==
==20512== Use --history-level=approx or =none to gain increased speed, at
==20512== the cost of reduced accuracy of conflicting-access information
==20512== For lists of detected and suppressed errors, rerun with: -s
==20512== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 519 from 78)
```

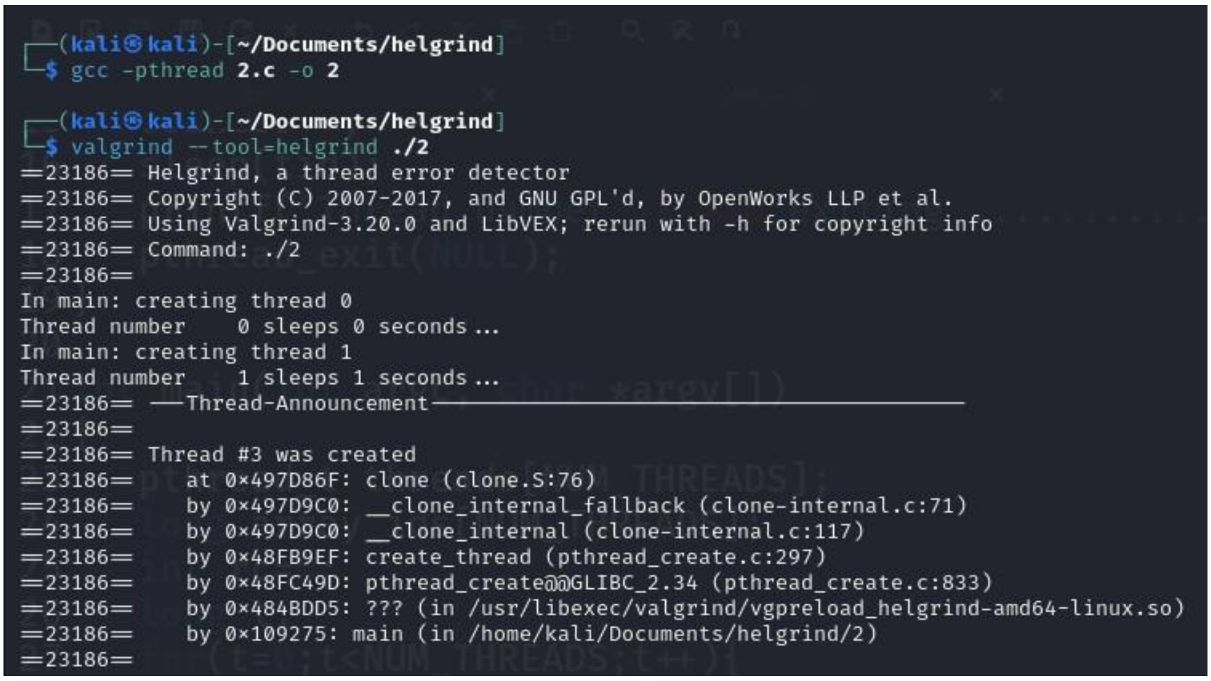
Let's change the code so that these threads share information using the counter variable. Potentially, this change causes race conditions, given the fact that several threads read, modify, and write on the same variable without any specific order (nor synchronising the code with a mutex):

//compilation in linux with gcc -pthread option

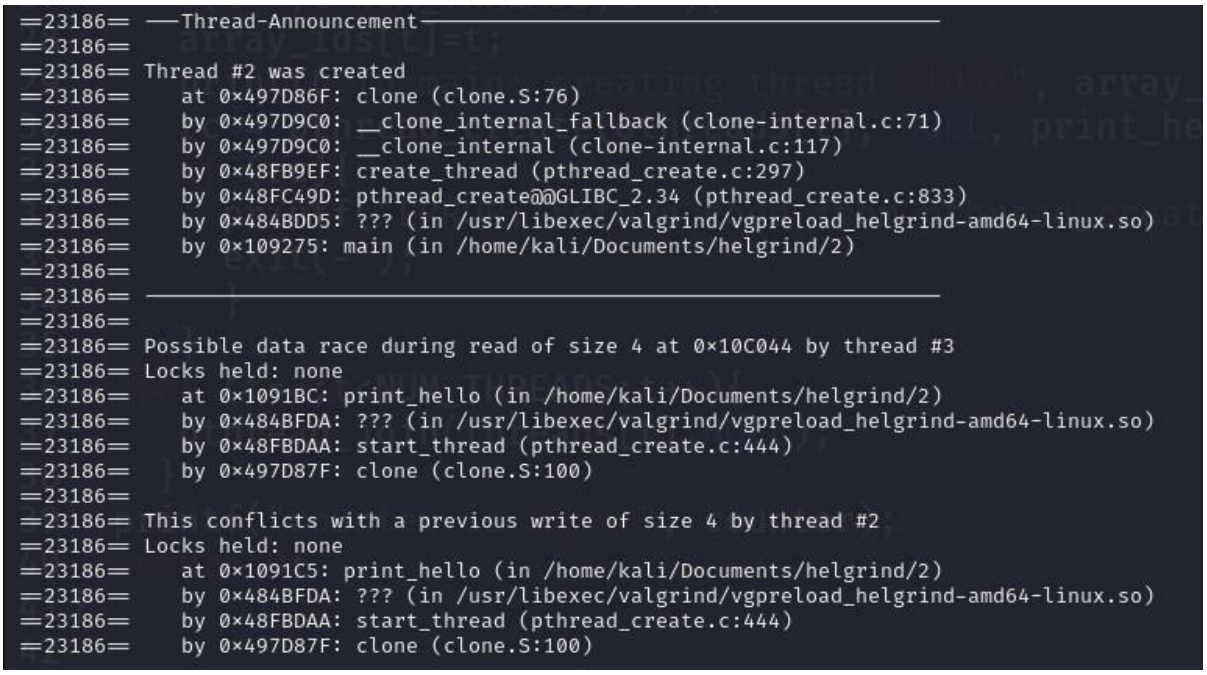
```
#include <pthread.h> #include <stdio.h> #include <stdlib.h> #include <unistd.h>
#define NUM_THREADS 2


int counter=0;
void *print_hello(void *threadid)
{
long tid;
tid = (long)threadid;
printf("Thread number \t %ld sleeps %ld seconds...\n",tid,tid); counter++;
sleep(tid);
printf("Thread number \t %ld exiting  \n",tid);
pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS]; long array_ids[NUM_THREADS];
int rc=0; long t;
for(t=0;t<NUM_THREADS;t++){
array_ids[t]=t;
printf("In main: creating thread %ld\n", array_ids[t]);
rc = pthread_create(&threads[t], NULL, print_hello, (void *)t); if (rc){
printf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);
}
}
for (t=0;t<NUM_THREADS;t++){
pthread_join(threads[t],NULL);
}
printf("counter is %i \n", counter); return 0;
}
```

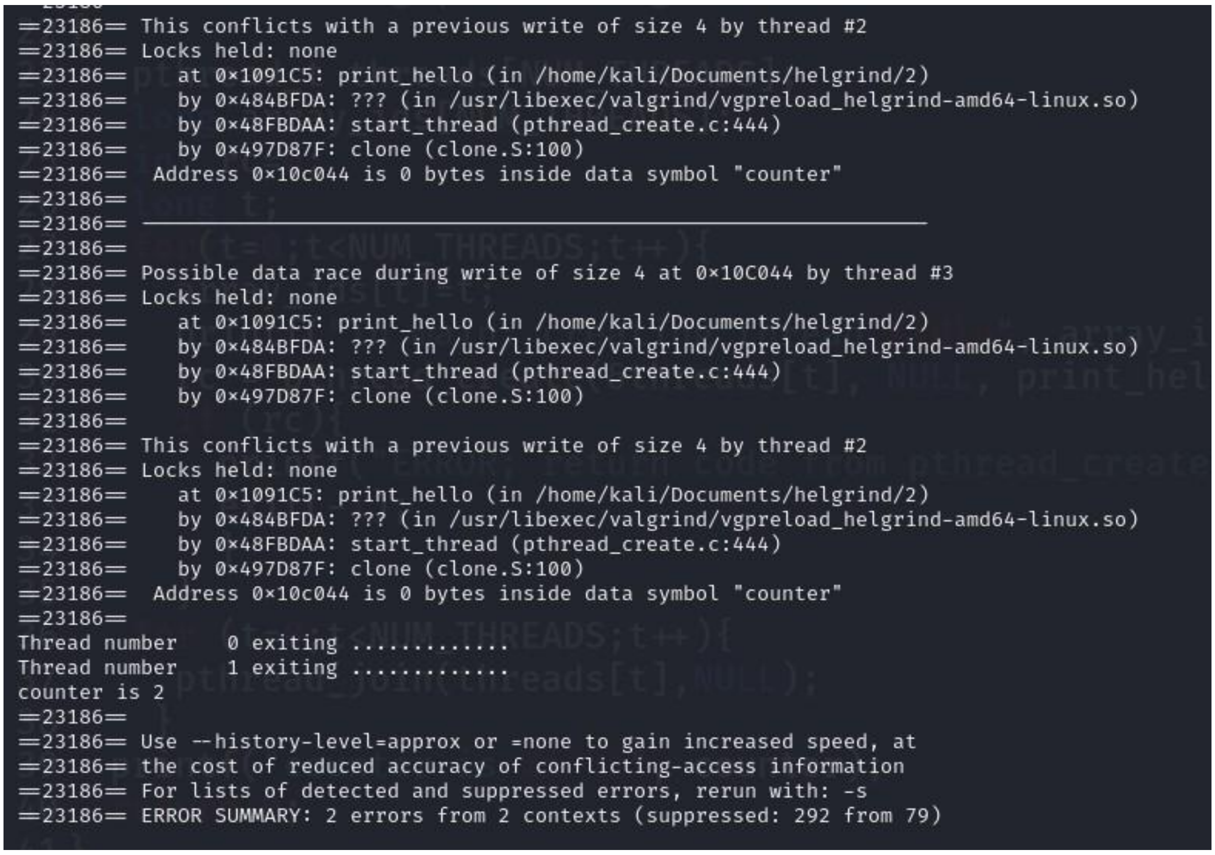
This problem is detected by Helgrind, as it runs on the previous code, which outputs a "data race"issue:



```
┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ gcc -pthread 2.c -o 2

┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ valgrind --tool=helgrind ./2
==23186== Helgrind, a thread error detector
==23186== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==23186== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==23186== Command: ./2
==23186==
In main: creating thread 0
Thread number    0 sleeps 0 seconds ...
In main: creating thread 1
Thread number    1 sleeps 1 seconds ...
==23186== ---Thread-Announcement------------------------------------------
==23186==
==23186== Thread #3 was created
==23186==    at 0x497D86F: clone (clone.S:76)
==23186==    by 0x497D9C0: __clone_internal_fallback (clone-internal.c:71)
==23186==    by 0x497D9C0: __clone_internal (clone-internal.c:117)
==23186==    by 0x48FB9EF: create_thread (pthread_create.c:297)
==23186==    by 0x48FC49D: pthread_create@@GLIBC_2.34 (pthread_create.c:833)
==23186==    by 0x484BDD5: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==23186==    by 0x109275: main (in /home/kali/Documents/helgrind/2)
==23186==
```



```
==23186== ---Thread-Announcement------------------------------------------
==23186==
==23186== Thread #2 was created
==23186==    at 0x497D86F: clone (clone.S:76)
==23186==    by 0x497D9C0: __clone_internal_fallback (clone-internal.c:71)
==23186==    by 0x497D9C0: __clone_internal (clone-internal.c:117)
==23186==    by 0x48FB9EF: create_thread (pthread_create.c:297)
==23186==    by 0x48FC49D: pthread_create@@GLIBC_2.34 (pthread_create.c:833)
==23186==    by 0x484BDD5: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==23186==    by 0x109275: main (in /home/kali/Documents/helgrind/2)
==23186==
==23186== ----------------------------------------------------------------
==23186==
==23186== Possible data race during read of size 4 at 0x10C044 by thread #3
==23186== Locks held: none
==23186==    at 0x1091BC: print_hello (in /home/kali/Documents/helgrind/2)
==23186==    by 0x484BFDA: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==23186==    by 0x48FBDAA: start_thread (pthread_create.c:444)
==23186==    by 0x497D87F: clone (clone.S:100)
==23186==
==23186== This conflicts with a previous write of size 4 by thread #2
==23186== Locks held: none
==23186==    at 0x1091C5: print_hello (in /home/kali/Documents/helgrind/2)
==23186==    by 0x484BFDA: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==23186==    by 0x48FBDAA: start_thread (pthread_create.c:444)
==23186==    by 0x497D87F: clone (clone.S:100)
```



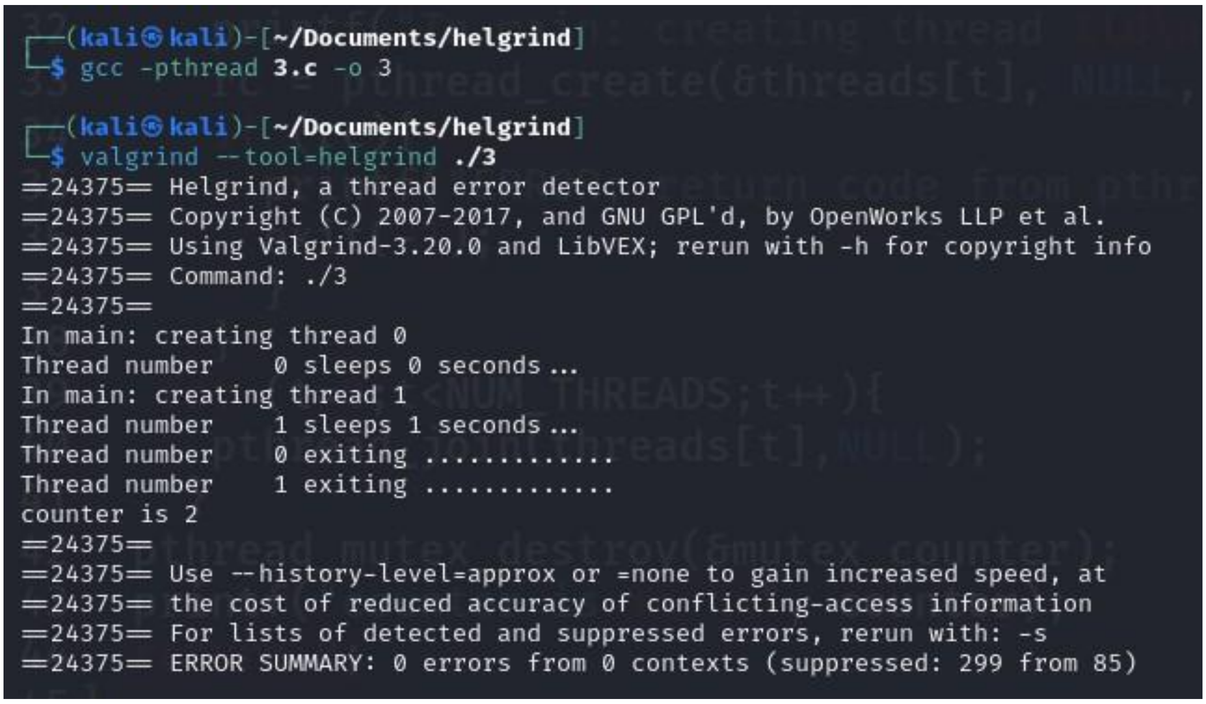
```
==23186== This conflicts with a previous write of size 4 by thread #2
==23186== Locks held: none
==23186==    at 0x1091C5: print_hello (in /home/kali/Documents/helgrind/2)
==23186==    by 0x484BFDA: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==23186==    by 0x48FBDAA: start_thread (pthread_create.c:444)
==23186==    by 0x497D87F: clone (clone.S:100)
==23186==  Address 0x10c044 is 0 bytes inside data symbol "counter"
==23186==
==23186== ----------------------------------------------------------------
==23186==
==23186== Possible data race during write of size 4 at 0x10C044 by thread #3
==23186== Locks held: none
==23186==    at 0x1091C5: print_hello (in /home/kali/Documents/helgrind/2)
==23186==    by 0x484BFDA: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==23186==    by 0x48FBDAA: start_thread (pthread_create.c:444)
==23186==    by 0x497D87F: clone (clone.S:100)
==23186==
==23186== This conflicts with a previous write of size 4 by thread #2
==23186== Locks held: none
==23186==    at 0x1091C5: print_hello (in /home/kali/Documents/helgrind/2)
==23186==    by 0x484BFDA: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==23186==    by 0x48FBDAA: start_thread (pthread_create.c:444)
==23186==    by 0x497D87F: clone (clone.S:100)
==23186==  Address 0x10c044 is 0 bytes inside data symbol "counter"
==23186==
Thread number    0 exiting ............
Thread number    1 exiting ............
counter is 2
==23186==
==23186== Use --history-level=approx or =none to gain increased speed, at
==23186== the cost of reduced accuracy of conflicting-access information
==23186== For lists of detected and suppressed errors, rerun with: -s
==23186== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 292 from 79)
```

If you are not interested in keeping the behaviour of the code, one solution are locks (mutex) to dealwith the issue. So that a correct version for the previous code is:

```
#include <pthread.h> #include <stdio.h> #include <stdlib.h>

#include <unistd.h> #define NUM_THREADS 2
```

```
pthread_mutex_t mutex_counter; int counter=0;

void *print_hello(void *threadid)
{
long tid;
tid = (long)threadid;
printf("Thread number \t %ld sleeps %ld seconds...\n",tid,tid);
pthread_mutex_lock(&mutex_counter);
counter++;
pthread_mutex_unlock(&mutex_counter); sleep(tid);
printf("Thread number \t %ld exiting  \n",tid);
pthread_exit(NULL);
}


int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS]; long array_ids[NUM_THREADS];
pthread_mutex_init(&mutex_counter,NULL); int rc=0;
long t;
for(t=0;t<NUM_THREADS;t++){
array_ids[t]=t;
printf("In main: creating thread %ld\n", array_ids[t]);
rc = pthread_create(&threads[t], NULL, print_hello, (void *)t); if (rc){
rintf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);
}

}
for (t=0;t<NUM_THREADS;t++){
pthread_join(threads[t],NULL);
}
pthread_mutex_destroy(&mutex_counter); printf("counter is %i \n", counter);
return 0;

}
```

Which removes the issue:



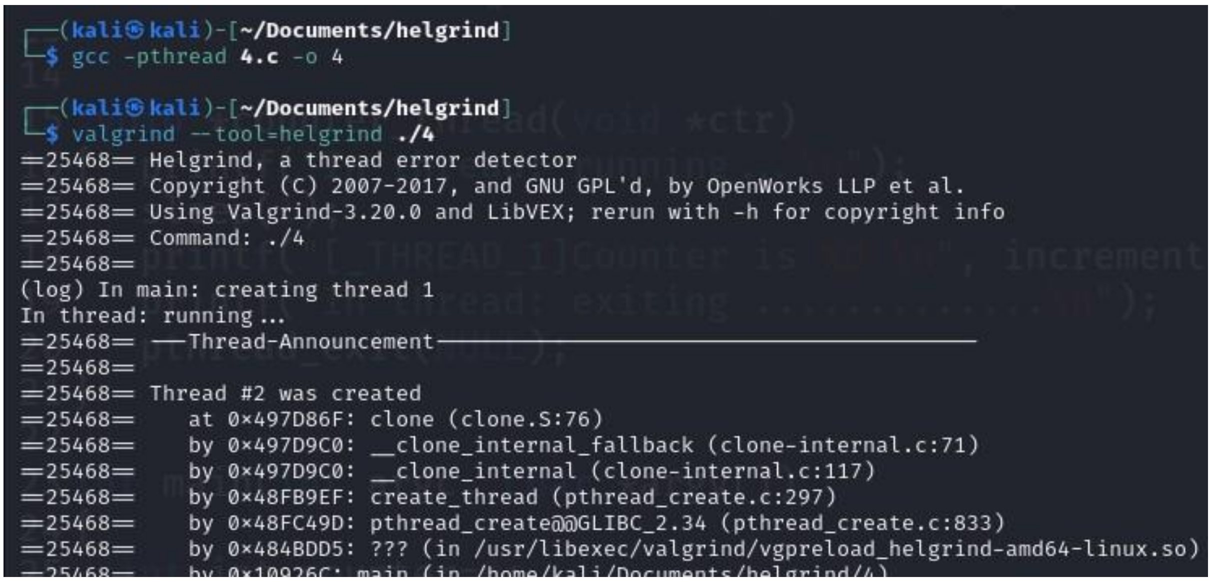
```
┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ gcc -pthread 3.c -o 3

┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ valgrind --tool=helgrind ./3
==24375== Helgrind, a thread error detector
==24375== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==24375== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==24375== Command: ./3
==24375==
In main: creating thread 0
Thread number    0 sleeps 0 seconds ...
In main: creating thread 1
Thread number    1 sleeps 1 seconds ...
Thread number    0 exiting ............
Thread number    1 exiting ............
counter is 2
==24375==
==24375== Use --history-level=approx or =none to gain increased speed, at
==24375== the cost of reduced accuracy of conflicting-access information
==24375== For lists of detected and suppressed errors, rerun with: -s
==24375== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 299 from 85)
```

Race Condition

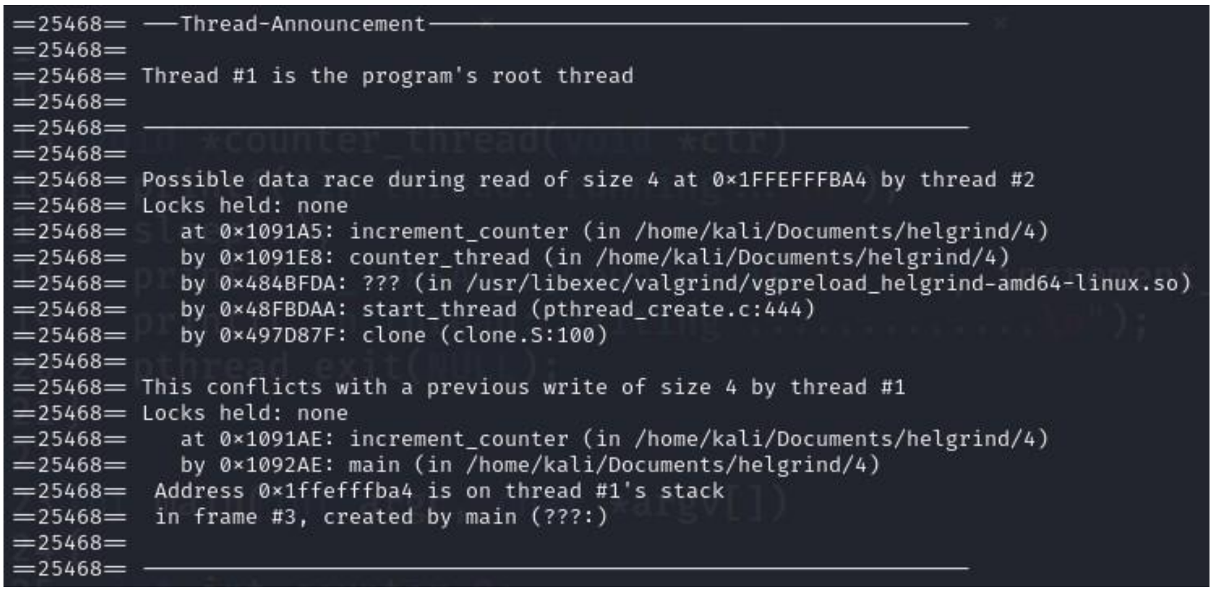
Our first example refers to a race condition. Our race codition is among the main and the uniqueother thread in the application.

```
//compilation in linux with gcc -pthread option#include <pthread.h>
#include <stdio.h>


#include <stdlib.h> #include <unistd.h>
int increment_counter(int *counter)
{
(*counter)++; return *counter;
}
void *counter_thread(void *ctr)
{ printf("In thread: running...\n"); sleep(1);
printf("[_THREAD_1]Counter is %d \n", increment_counter((int*)ctr) ); printf("In thread:
exiting    \n");
pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
int int_counter=0;
pthread_t threads[1]; int rc=0;
printf("(log) In main: creating thread %i\n", 1);
rc = pthread_create(&threads[0], NULL, counter_thread, (void *)&int_counter); if (rc){
printf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);
}
sleep(1);
int res_counter=increment_counter(&int_counter); pthread_join(threads[0],NULL);
printf("[_MAIN     ] Counter is %i \n", res_counter); return 0;
}
```

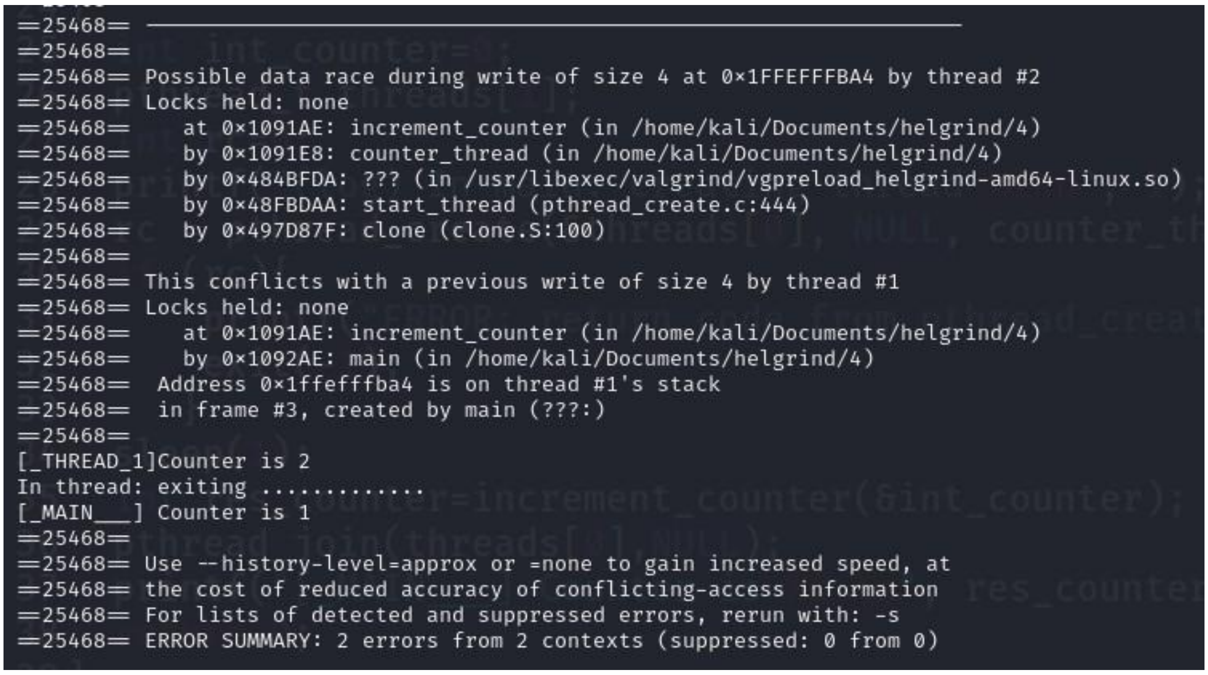
This issue is detected by Helgrind which returns the following output:



```
┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ gcc -pthread 4.c -o 4

┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ valgrind --tool=helgrind ./4
==25468== Helgrind, a thread error detector
==25468== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==25468== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==25468== Command: ./4
==25468==
(log) In main: creating thread 1
In thread: running ...
==25468== ---Thread-Announcement------------------------------------------
==25468==
==25468== Thread #2 was created
==25468==    at 0x497D86F: clone (clone.S:76)
==25468==    by 0x497D9C0: __clone_internal_fallback (clone-internal.c:71)
==25468==    by 0x497D9C0: __clone_internal (clone-internal.c:117)
==25468==    by 0x48FB9EF: create_thread (pthread_create.c:297)
==25468==    by 0x48FC49D: pthread_create@@GLIBC_2.34 (pthread_create.c:833)
==25468==    by 0x484BDD5: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==25468==    by 0x10926C: main (in /home/kali/Documents/helgrind/4)
```



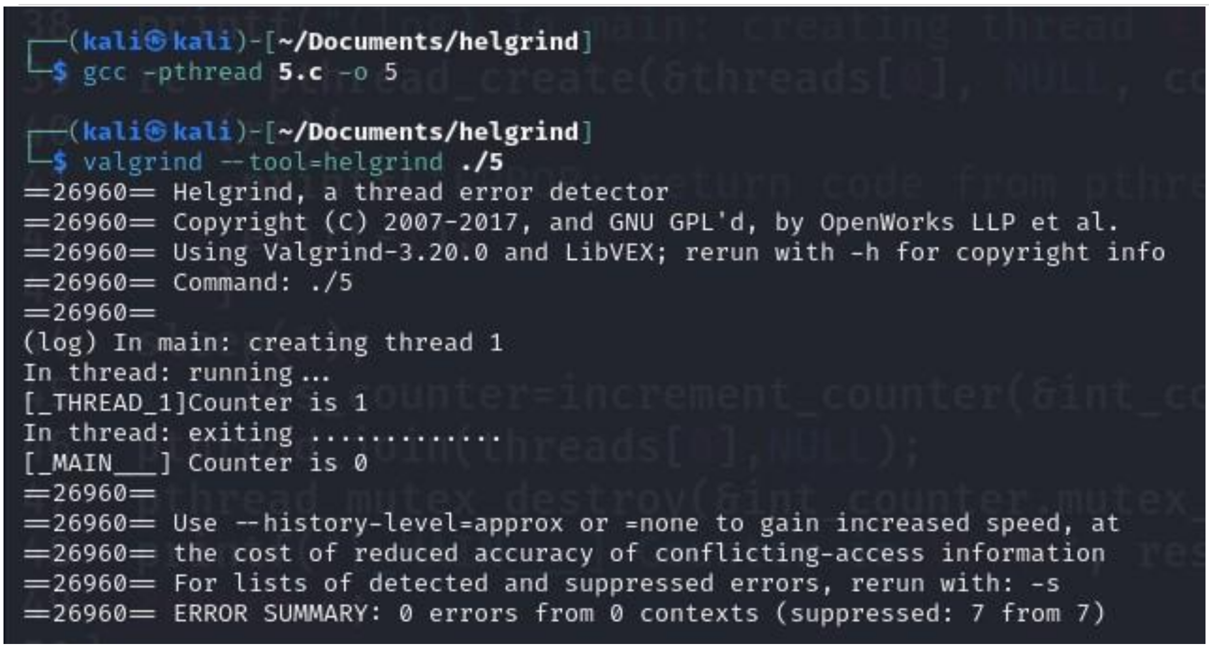
```
==25468== ---Thread-Announcement------------------------------------------
==25468==
==25468== Thread #1 is the program's root thread
==25468==
==25468== ----------------------------------------------------------------
==25468==
==25468== Possible data race during read of size 4 at 0x1FFEFFFBA4 by thread #2
==25468== Locks held: none
==25468==    at 0x1091A5: increment_counter (in /home/kali/Documents/helgrind/4)
==25468==    by 0x1091E8: counter_thread (in /home/kali/Documents/helgrind/4)
==25468==    by 0x484BFDA: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==25468==    by 0x48FBDAA: start_thread (pthread_create.c:444)
==25468==    by 0x497D87F: clone (clone.S:100)
==25468==
==25468== This conflicts with a previous write of size 4 by thread #1
==25468== Locks held: none
==25468==    at 0x1091AE: increment_counter (in /home/kali/Documents/helgrind/4)
==25468==    by 0x1092AE: main (in /home/kali/Documents/helgrind/4)
==25468==  Address 0x1ffefffba4 is on thread #1's stack
==25468==  in frame #3, created by main (???:)
==25468==
==25468== ----------------------------------------------------------------
```



```
==25468== ------------------------------------------------------------------
==25468==
==25468== Possible data race during write of size 4 at 0x1FFEFFFBA4 by thread #2
==25468== Locks held: none
==25468==    at 0x1091AE: increment_counter (in /home/kali/Documents/helgrind/4)
==25468==    by 0x1091E8: counter_thread (in /home/kali/Documents/helgrind/4)
==25468==    by 0x484BFDA: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==25468==    by 0x48FBDAA: start_thread (pthread_create.c:444)
==25468==    by 0x497D87F: clone (clone.S:100)
==25468==
==25468== This conflicts with a previous write of size 4 by thread #1
==25468== Locks held: none
==25468==    at 0x1091AE: increment_counter (in /home/kali/Documents/helgrind/4)
==25468==    by 0x1092AE: main (in /home/kali/Documents/helgrind/4)
==25468==  Address 0x1ffefffba4 is on thread #1's stack
==25468==  in frame #3, created by main (???:)
==25468==
[_THREAD_1]Counter is 2
In thread: exiting ............
[_MAIN___] Counter is 1
==25468==
==25468== Use --history-level=approx or =none to gain increased speed, at
==25468== the cost of reduced accuracy of conflicting-access information
==25468== For lists of detected and suppressed errors, rerun with: -s
==25468== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

To sort out the isssue, one solution is to use a mutex. The following piece of code introduces the changes that address the issue:

```
//compilation in linux with gcc -pthread option #include <pthread.h>
#include <stdio.h> #include <stdlib.h> #include <unistd.h> struct struct_counter{ int i;
pthread_mutex_t mutex_i;
};
int increment_counter(struct struct_counter* counter)
{ int to_return=0;
pthread_mutex_lock(&((counter)->mutex_i)); to_return=(*counter).i++;
pthread_mutex_unlock(&((counter)->mutex_i)); return to_return;
}





void *counter_thread(void *ctr)
{ printf("In thread: running...\n"); sleep(1);
printf("[_THREAD_1]Counter is %d \n", increment_counter((struct struct_counter*)ctr) );
printf("In thread: exiting      \n");
pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
struct struct_counter int_counter; int_counter.i=0;
pthread_mutex_init(&int_counter.mutex_i,NULL); pthread_t threads[1];
int rc=0;
printf("(log) In main: creating thread %i\n", 1);
rc = pthread_create(&threads[0], NULL, counter_thread, (struct struct_counter
*)&int_counter); if (rc){
printf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);
}
sleep(1);
int res_counter=increment_counter(&int_counter); pthread_join(threads[0],NULL);
pthread_mutex_destroy(&int_counter.mutex_i);
printf("[_MAIN     ] Counter is %i \n", res_counter); return 0;
}
```



```
┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ gcc -pthread 5.c -o 5

┌──(kali㉿kali)-[~/Documents/helgrind]
└─$ valgrind --tool=helgrind ./5
==26960== Helgrind, a thread error detector
==26960== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==26960== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==26960== Command: ./5
==26960==
(log) In main: creating thread 1
In thread: running...
[_THREAD_1]Counter is 1
In thread: exiting ............
[_MAIN___] Counter is 0
==26960==
==26960== Use --history-level=approx or =none to gain increased speed, at
==26960== the cost of reduced accuracy of conflicting-access information
==26960== For lists of detected and suppressed errors, rerun with: -s
==26960== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 7 from 7)
```

POSIX thread (pthread) libraries

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a newconcurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for theprocess. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution.(One thread may execute while another is waiting for I/O or some other system latency.) Parallel programming technologies such as MPI and PVM are used in a distributed computing environmentwhile threads are limited to a single computer system. All threads within a process share the same address space. A thread is spawned by defining a function and it's arguments which will be processedin the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

Thread Basics:

Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling,data management and process interaction.

A thread does not maintain a list of created threads, nor does it know the thread that

created it.All threads within a process share the same address space.

Threads in the same process share:

- *Process instructions*
- *Most data*
- *open files (descriptors)*
- *signals and signal handlers*
- *current working directory*
- *User and group*

idEach thread has a

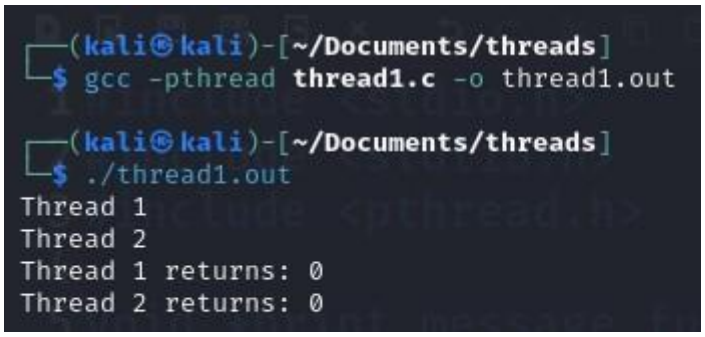
unique:

- *Thread ID*
- *set of registers, stack pointer*
- *stack for local variables, return addresses*
- *signal mask*
- *priority*
- *Return value: errno*

pthread functions return "0" if OK.

Thread Creation and Termination:

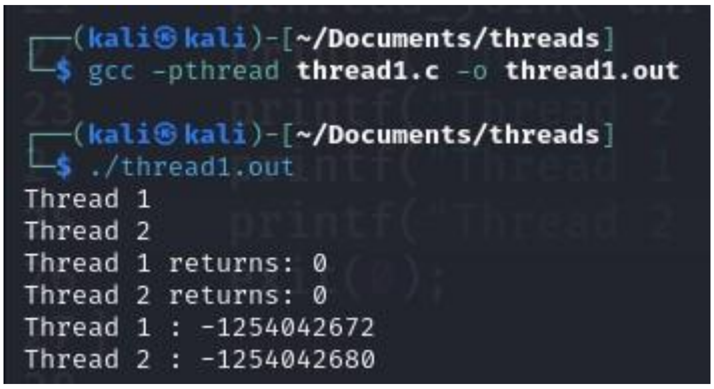
```
Example: pthread1.c#include <stdio.h>
#include <stdlib.h> #include <pthread.h>

void *print_message_function( void *ptr )
{
char *message;
message = (char *) ptr; printf("%s \n", message);
}


int main()
{
pthread_t thread1, thread2;
char *message1 = "Thread 1"; char *message2 = "Thread 2"; int iret1, iret2;
iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1); iret2 =
pthread_create( &thread2, NULL, print_message_function, (void*) message2);
pthread_join( thread1, NULL);
pthread_join( thread2, NULL);
printf("Thread 1 returns: %d\n",iret1); printf("Thread 2 returns: %d\n",iret2); exit(0);
}
```



```
┌──(kali㉿kali)-[~/Documents/threads]
└─$ gcc -pthread thread1.c -o thread1.out

┌──(kali㉿kali)-[~/Documents/threads]
└─$ ./thread1.out
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

Details:

- *In this example the same function is used in each thread. The arguments are different. Thefunctions need not be the same.*
- *Threads terminate by explicitly calling pthread_exit, by letting the function return, or by acall to the function exit which will terminate the process including any threads.*
- *Function call: **pthread_create***



```
┌──(kali㉿kali)-[~/Documents/threads]
└─$ gcc -pthread thread1.c -o thread1.out

┌──(kali㉿kali)-[~/Documents/threads]
└─$ ./thread1.out
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
Thread 1 : -1254042672
Thread 2 : -1254042680
```

Arguments:

- *thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)*
- *attr - Set to NULL if default thread attributes are used. (else define members of the structpthread_attr_t defined in bits/pthreadtypes.h) Attributes include:*
    1. *detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option:PTHREAD_CREATE_DETACHED)*
    2. *scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)*
    3. *scheduling parameter*
    4. *inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parentthread: PTHREAD_INHERIT_SCHED)*
    5. *scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads:PTHREAD_SCOPE_PROCESS Pick one or the other not both.)*
    6. *guard size*
    7. *stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)*
    8. *stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),*
- *void * (*start_routine) - pointer to the function to be threaded. Function has a singleargument: pointer to void.*
- **arg - pointer to argument of function. To pass multiple arguments, send a pointer to astructure.*

Function call: pthread_exit

*void pthread_exit(void *retval);*

Arguments:

- *retval - Return value of thread.*

This routine kills the thread. The pthread_exit function never returns. If the thread is not detached,the thread id and return value may be examined from another thread by using *pthread_join.*

*Note: the return pointer *retval, must not be of local scope otherwise it would cease to exist oncethe thread terminates.*

- *[C++ pitfalls]: The above sample program will compile with the GNU C and C++ compiler g++.The following function pointer representation below will work for C but not C++. Note the subtle differences and avoid the pitfall below:*

```
void print_message_function( void *ptr );
...
...
iret1 = pthread_create( &thread1, NULL, (void*)&print_message_function, (void*) message1);
...
...
```

Thread Synchronization:

The threads library provides three synchronization mechanisms:

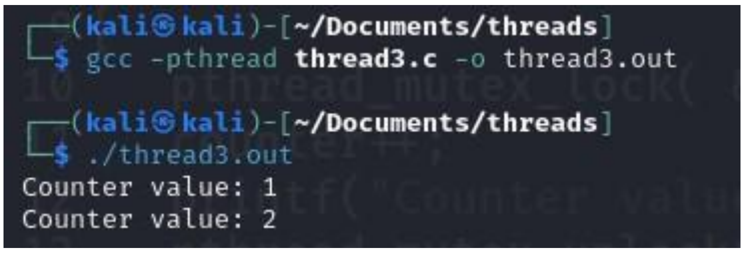
- *mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforcesexclusive access by a thread to a variable or set of variables.*
- *joins - Make a thread wait till others are complete (terminated).*
- *condition variables - data type pthread_cond_t*

Mutexes:

Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the
results of computations depends on the order in which these operations are performed. Mutexes areused for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. One can apply a mutex to protect a segment ofmemory ("critical region") from other threads. Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

```
#include <pthread.h> #include <stdio.h> #include <stdlib.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER; int counter = 0;

void *functionC()
{
pthread_mutex_lock( &mutex1 ); counter++;
printf("Counter value: %d\n",counter); pthread_mutex_unlock( &mutex1 );
}


int main()
{
```

```
int rc1, rc2;
pthread_t thread1, thread2;
if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
{
printf("Thread creation failed: %d\n", rc1);
}


if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
{
printf("Thread creation failed: %d\n", rc2);
}
pthread_join( thread1, NULL); pthread_join( thread2, NULL); exit(0);
}
```



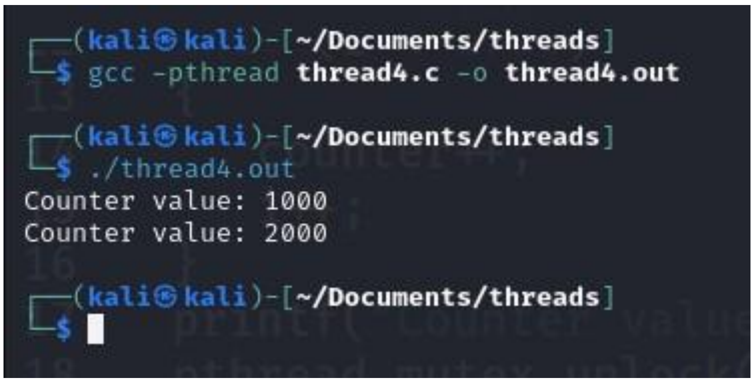
```
┌──(kali㉿kali)-[~/Documents/threads]
└─$ gcc -pthread thread3.c -o thread3.out

┌──(kali㉿kali)-[~/Documents/threads]
└─$ ./thread3.out
Counter value: 1
Counter value: 2
```

When a mutex lock is attempted against a mutex which is held by another thread, the thread is blocked until the mutex is unlocked. When a thread terminates, the mutex does not unless explicitlyunlocked. Nothing happens by default.

```
#include <stdio.h> #include <stdlib.h> #include <pthread.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER; int counter = 0;

void *functionC()
{
int i=0;
pthread_mutex_lock( &mutex1 ); while(i < 1000)
{
counter++; i++;
}
printf("Counter value: %d\n",counter); pthread_mutex_unlock( &mutex1 );
}


int main()
{
int rc1, rc2;
pthread_t thread1, thread2;


if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
```

```
{
printf("Thread creation failed: %d\n", rc1);
}


if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
{
printf("Thread creation failed: %d\n", rc2);
}
pthread_join( thread1, NULL); pthread_join( thread2, NULL); exit(0);
```

}



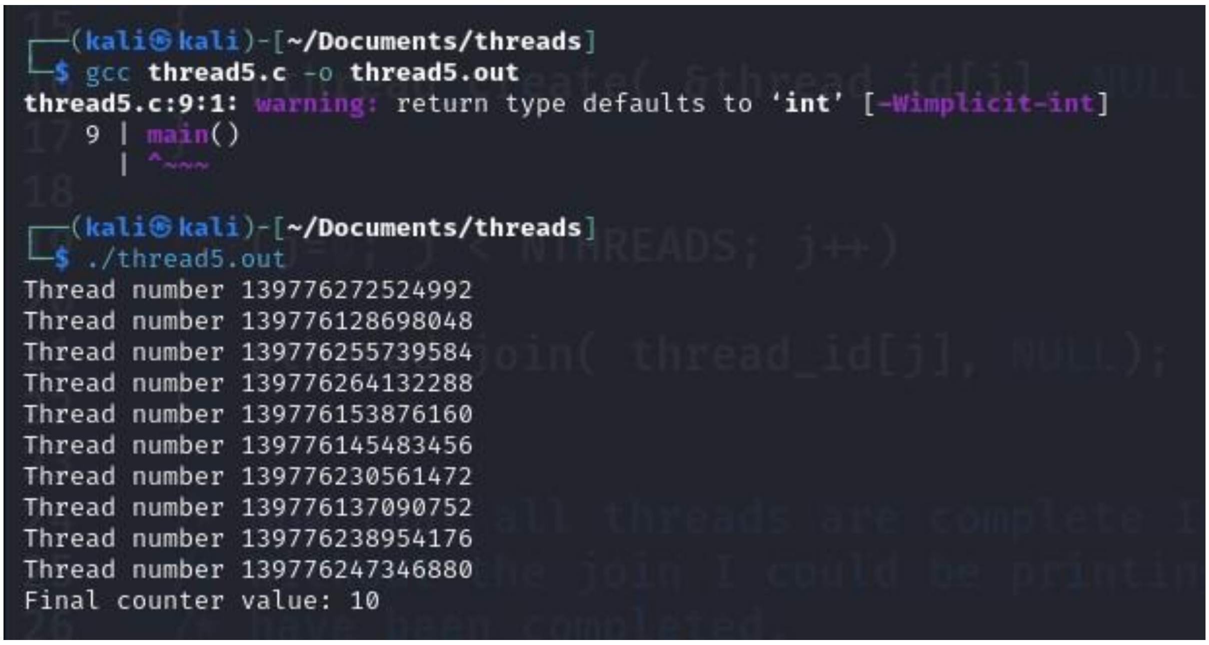
```
┌──(kali㉿kali)-[~/Documents/threads]
└─$ gcc -pthread thread4.c -o thread4.out

┌──(kali㉿kali)-[~/Documents/threads]
└─$ ./thread4.out
Counter value: 1000
Counter value: 2000

┌──(kali㉿kali)-[~/Documents/threads]
└─$ 
```

Joins:

A join is performed when one wants to wait for a thread to finish. A thread calling routine may launch multiple threads then wait for them to finish to get the results. One wait for the completionof the threads with a join.

Sample code: join1.c

```
#include <stdio.h> #include <pthread.h>

#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER; int counter = 0;

main()
{
pthread_t thread_id[NTHREADS];

int i, j;

for(i=0; i < NTHREADS; i++)
{
pthread_create( &thread_id[i], NULL, thread_function, NULL );
}

for(j=0; j < NTHREADS; j++)
{
pthread_join( thread_id[j], NULL);
}

/* Now that all threads are complete I can print the final result.     */
/* Without the join I could be printing a value before all the threads */
/* have been completed.       */

printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
printf("Thread number %ld\n", pthread_self()); pthread_mutex_lock( &mutex1 );
counter++;
pthread_mutex_unlock( &mutex1 );
}
```



```
┌──(kali㉿kali)-[~/Documents/threads]
└─$ gcc thread5.c -o thread5.out
thread5.c:9:1: warning: return type defaults to 'int' [-Wimplicit-int]
    9 | main()
      | ^~~~

┌──(kali㉿kali)-[~/Documents/threads]
└─$ ./thread5.out
Thread number 139776272524992
Thread number 139776128698048
Thread number 139776255739584
Thread number 139776264132288
Thread number 139776153876160
Thread number 139776145483456
Thread number 139776230561472
Thread number 139776137090752
Thread number 139776238954176
Thread number 139776247346880
Final counter value: 10
```

Condition Variables:

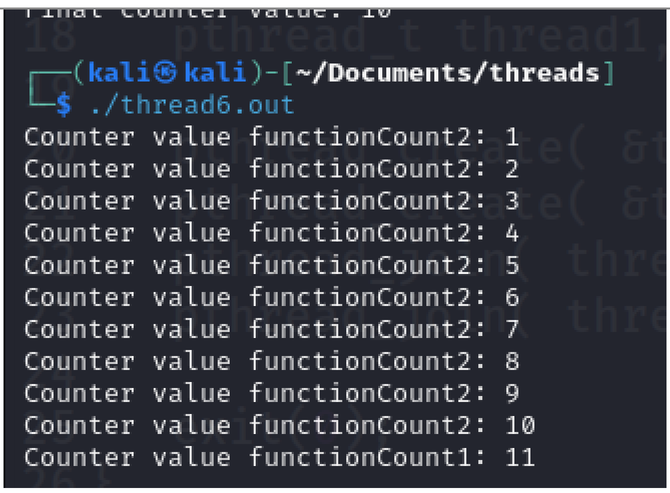
A condition variable is a variable of type pthread_cond_t and is used with the appropriate functions for waiting and later, process continuation. The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true. A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing towait and another thread which may signal the condition before the first thread actually waits on it
resulting in a deadlock. The thread will be perpetually waiting for a signal that is never sent. Anymutex can be used, there is no explicit link between the mutex and the condition variable.

Functions used in conjunction with the condition variable:

- *Creating/Destroying:*
  1. *pthread_cond_init*
  2. *pthread_cond_t cond = PTHREAD_COND_INITIALIZER;*
  3. *pthread_cond_destroy*
- *Waiting on condition:*
  1. *pthread_cond_wait*
  2. *pthread_cond_timedwait - place limit on how long it will block.*
- *Waking thread based on condition:*
  1. *pthread_cond_signal*
  2. *pthread_cond_broadcast - wake up all threads blocked by the specified condition variable.*

Example code: cond1.c

```
#include <stdio.h> #include <stdlib.h> #include <pthread.h>

pthread_mutex_t count_mutex           = PTHREAD_MUTEX_INITIALIZER; pthread_mutex_t
condition_mutex = PTHREAD_MUTEX_INITIALIZER; pthread_cond_t condition_cond =
PTHREAD_COND_INITIALIZER;

void *functionCount1(); void *functionCount2(); int count = 0;
#define COUNT_DONE 10
#define COUNT_HALT1 3
#define COUNT_HALT2 6

main()
{
pthread_t thread1, thread2;

pthread_create( &thread1, NULL, &functionCount1, NULL); pthread_create( &thread2,
NULL, &functionCount2, NULL); pthread_join( thread1, NULL);
pthread_join( thread2, NULL);
exit(0);
}

void *functionCount1()
{
for(;;)
{
pthread_mutex_lock( &condition_mutex );
while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
{
pthread_cond_wait( &condition_cond, &condition_mutex );
}
pthread_mutex_unlock( &condition_mutex );

pthread_mutex_lock( &count_mutex ); count++;
printf("Counter value functionCount1: %d\n",count); pthread_mutex_unlock( &count_mutex );
if(count >= COUNT_DONE) return(NULL);
}
}

void *functionCount2()
{
for(;;)
{
pthread_mutex_lock( &condition_mutex );
if( count < COUNT_HALT1 || count > COUNT_HALT2 )
{
pthread_cond_signal( &condition_cond );
}
pthread_mutex_unlock( &condition_mutex );

pthread_mutex_lock( &count_mutex ); count++;
printf("Counter value functionCount2: %d\n",count); pthread_mutex_unlock( &count_mutex );

if(count >= COUNT_DONE) return(NULL);
}

}
```



```
┌──(kali㉿kali)-[~/Documents/threads]
└─$ ./thread6.out
Counter value functionCount2: 1
Counter value functionCount2: 2
Counter value functionCount2: 3
Counter value functionCount2: 4
Counter value functionCount2: 5
Counter value functionCount2: 6
Counter value functionCount2: 7
Counter value functionCount2: 8
Counter value functionCount2: 9
Counter value functionCount2: 10
Counter value functionCount1: 11
```

Note that functionCount1() was halted while count was between the values COUNT_HALT1 and COUNT_HALT2. The only thing that has been ensures is that functionCount2 will increment the countbetween the values COUNT_HALT1 and COUNT_HALT2. Everything else is random.

The logic conditions (the "if" and "while" statements) must be chosen to insure that the "signal" isexecuted if the "wait" is ever processed. Poor software logic can also lead to a deadlock condition.

Note: Race conditions abound with this example because count is used as the condition and can't belocked in the while statement without causing deadlock. I'll work on a cleaner example *but it is an example of a condition variable.*

Thread Scheduling:

When this option is enabled, each thread may have its own scheduling properties. Schedulingattributes may be specified:

during thread creation

by dynamically by changing the attributes of a thread already created

by defining the effect of a mutex on the thread's scheduling when creating a mutex

by dynamically changing the scheduling of a thread during synchronization

operations.The threads library provides default values that are sufficient for most

cases.

Thread Pitfalls:

Race conditions: While the code may appear on the screen in the order you wish the code to execute, threads are scheduled by the operating system and are executed at random. It cannot be assumed that threads are executed in the order they are created. They may also execute at differentspeeds. When threads are executing (racing to complete) they may give unexpected results (race condition). Mutexes and joins must be utilized to achieve a predictable execution order and outcome.

Thread safe code: The threaded routines must call functions which are "thread safe". This means thatthere are no static or global variables which other threads may clobber or read assuming single
threaded operation. If static or global variables are used then mutexes must be applied or the functions must be re-written to avoid the use of these variables. In C, local variables are dynamicallyallocated on the stack. Therefore, any function that does not use static data or other shared
resources is thread-safe. Thread-unsafe functions may be used by only one thread at a time in a program and the uniqueness of the thread must be ensured. Many non-reentrant functions return a pointer to static data. This can be avoided by returning dynamically allocated data or using caller- provided storage. An example of a non-thread safe function is strtok which is also not re-entrant. The"thread safe" version is the re-entrant version strtok_r.

Mutex Deadlock: This condition occurs when a mutex is applied but then not "unlocked". This causesprogram execution to halt indefinitely. It can also be caused by poor application of mutexes or joins.Be careful when applying two or more mutexes to a section of code. If the first pthread_mutex_lock is applied and the second pthread_mutex_lock fails due to another thread applying a mutex, the firstmutex may eventually lock all other threads from accessing data including the thread which holds
the second mutex. The threads may wait indefinitely for the resource to become free causing a

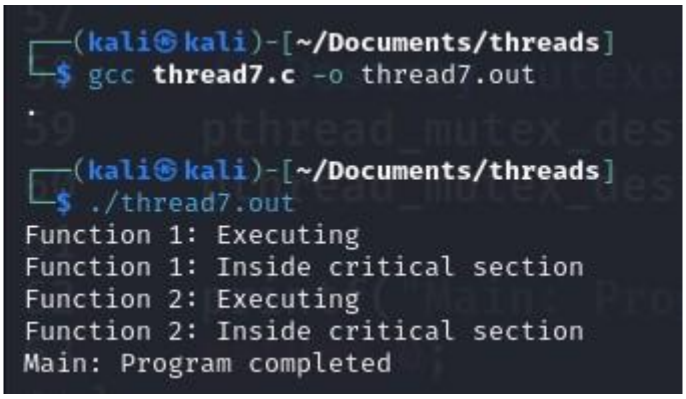
deadlock. It is best to test and if failure occurs, free the resources and stall before retrying.

...

```
pthread_mutex_lock(&mutex_1);
while ( pthread_mutex_trylock(&mutex_2) ) /* Test if already locked */
{
pthread_mutex_unlock(&mutex_1); /* Free resource to avoid deadlock */
...
/* stall here */
...
pthread_mutex_lock(&mutex_1);
}
count++;
pthread_mutex_unlock(&mutex_1); pthread_mutex_unlock(&mutex_2);
The order of applying the mutex is also important. The following code segment illustrates a
potential for deadlock:


void *function1()
{
...
pthread_mutex_lock(&lock1);           - Execution step 1
pthread_mutex_lock(&lock2);           - Execution step 3 DEADLOCK!!!
...
...
pthread_mutex_lock(&lock2); pthread_mutex_lock(&lock1);
...

}


void *function2()
{
...
pthread_mutex_lock(&lock2);           - Execution step 2
pthread_mutex_lock(&lock1);
...
...
pthread_mutex_lock(&lock1); pthread_mutex_lock(&lock2);
...
}


main()
{
...
pthread_create(&thread1, NULL, function1, NULL); pthread_create(&thread2, NULL,
function1, NULL);
}
```

Changed code:

```
#include <pthread.h> #include <stdio.h> #include <stdlib.h>

pthread_mutex_t lock1; // Mutex 1 pthread_mutex_t lock2; // Mutex 2

void *function1()
{
// Simulating some work
printf("Function 1: Executing\n");


pthread_mutex_lock(&lock1);          // Execution step 1
pthread_mutex_lock(&lock2);          // Execution step 3 DEADLOCK!!!


// Simulated critical section for function1
printf("Function 1: Inside critical section\n");


pthread_mutex_unlock(&lock2); pthread_mutex_unlock(&lock1);

return NULL;

}


void *function2()
{
// Simulating some work
printf("Function 2: Executing\n");


pthread_mutex_lock(&lock2);          // Execution step 2
pthread_mutex_lock(&lock1);          // This could cause deadlock if order is mismatched


// Simulated critical section for function2
printf("Function 2: Inside critical section\n");


pthread_mutex_unlock(&lock1); pthread_mutex_unlock(&lock2);

return NULL;
}


int main()
{
pthread_t thread1, thread2;


// Initialize mutexes
pthread_mutex_init(&lock1, NULL); pthread_mutex_init(&lock2, NULL);

// Create threads
pthread_create(&thread1, NULL, function1, NULL); pthread_create(&thread2, NULL,
function2, NULL);


// Wait for threads to finish pthread_join(thread1, NULL); pthread_join(thread2, NULL);

// Destroy mutexes
pthread_mutex_destroy(&lock1); pthread_mutex_destroy(&lock2);

printf("Main: Program completed\n"); return 0;
}
```



```
┌──(kali㉿kali)-[~/Documents/threads]
└─$ gcc thread7.c -o thread7.out
.
┌──(kali㉿kali)-[~/Documents/threads]
└─$ ./thread7.out
Function 1: Executing
Function 1: Inside critical section
Function 2: Executing
Function 2: Inside critical section
Main: Program completed
```

- *If function1 acquires the first mutex and function2 acquires the second, all resources are tiedup and locked.*
- *Condition Variable Deadlock: The logic conditions (the "if" and "while" statements) must bechosen to insure that the "signal" is executed if the "wait" is ever processed.*