

Table of Contents

Example 1:	2
Example 2:	2
Example 3:	4
Example 4:	5
Example 5:	7
Fixing using semaphores (Bonus):.....	11

Example 1:

Running the compiled, we get the following:

```
KhaleDahhasi@lamp ~/lab6$ ./example1
Parent: My process# ---> 1354
Parent: My thread # ---> 140498892134208
Child: Hello World! It's me, process# ---> 1354
Child: Hello World! It's me, thread # ---> 140498892130048
Parent: No more child thread!
```

Figure 1 Output of example 1.

In this program the parent creates a child thread and make it run a method where it would print the process id and thread id. As for the parent, after creating the child, it would also print its own process id and thread id. Expectations are that the process id would be the same but the thread id would differ. This is correct as shown in Figure 1 where the parent printed that their process id is 1354 and as for the child it printed that their process id is also 1354 which is true as threads are local to the process and they all share the same process's memory as its defined that A Thread Group is a set of threads, all executing inside the same process. But what would differ is the thread id because this id represents different threads and they should be different or otherwise we couldn't recognize threads from each other. Where in Figure 1 it can be seen that the thread id for the parent differs from that of the child.

Example 2:

Running the compiled program, we get the following:

```
KhaleDahhasi@lamp ~/lab6$ ./example2
Parent: Global data = 5
Child: Global data was 10.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
```

Figure 2 first-try output of example 2.

```
KhaleDahhasi@lamp ~/lab6$ ./example2
Parent: Global data = 5
Child: Global data was 5.
Child: Global data is now 15.
Parent: Global data = 15
Parent: End of program.
```

Figure 3 second-try output of example 2.

(parent | child)

In this program, there would be a global integer initialized first at 5. The parent thread first creates a child thread and after that it prints the value of the global integer. And the child prints the global integer as well. After printing, the parent changes the global integer to 10 and then waits for the child to finish and in parallel, the child changes the global integer to 15 and prints the global integer and finishes. Finally, the parent prints the global integer one last time.

Does the program gives the same output every time?

In such a program, some of the prints may not be constant on all trials. As it depends on how fast the child thread is created, started, and running. In Figure 2 we can see that on the first try running the program, the child thread wasn't fast enough to print the global integer when it was equal to its first value which is 5. What happened instead is that the parent thread was faster in both printing the first line and then changing the global integer to 10 and thus why the child printed it as 10 instead of 5. As for the second try shown in Figure 3, the child thread was fast enough to print the global integer as 5 before the parent changed it.

Do the threads share one copy of this global integer?

global data is indeed shared between threads of the same process which is why this behavior even happened as parent may change the value for the child or vice versa.

Example 3:

Running the compiled program, we get the following:

```
KhaleDahhasi@lamp ~/lab6$ ./example3
I am the parent thread
I am thread #1, My ID #139668927719168
I am thread #0, My ID #139668936111872
I am thread #2, My ID #139668919326464
I am thread #3, My ID #139668910933760
I am thread #4, My ID #139668902434560
I am thread #6, My ID #139668885649152
I am thread #8, My ID #139668868863744
I am thread #9, My ID #139668860471040
I am thread #7, My ID #139668877256448
I am thread #5, My ID #139668894041856
I am the parent thread again
```

Figure 4 first-try output of example 3.

```
KhaleDahhasi@lamp ~/lab6$ ./example3
I am the parent thread
I am thread #1, My ID #140030710945536
I am thread #2, My ID #140030702552832
I am thread #0, My ID #140030719338240
I am thread #3, My ID #140030694160128
I am thread #4, My ID #140030685660928
I am thread #5, My ID #140030677268224
I am thread #6, My ID #140030668875520
I am thread #7, My ID #140030660482816
I am thread #8, My ID #140030652090112
I am thread #9, My ID #140030643697408
I am the parent thread again
```

Figure 5 second-try output of example 3.

In this program, the parent thread prints the first line then creates multiple children threads, all these child threads would print their own id then finally the parent prints the last line.

Do the output lines come in the same order every time? Why?

No, it can be seen in Figure 4 and Figure 5 that the order at which the threads are given time by the schedule to run is different. As in, the scheduler, depending on the policy and the present state of other processes on the ready queue would give different results. Meaning that at the first try, it happened to be that order of threads that were chosen in that order by the system scheduler and on the second try another order was chosen. Note: this program uses the default system scheduler.

Example 4:

Running the compiled program, we get the following:

```
KhaleDahhasi@lamp ~/lab6$ ./example4
First, we create two threads to see better what context they share...
Set this_is_global to: 1000
Thread: 140257742817024, pid: 1728, addresses: local: 0X4CEFFEDC, global: 0XE4D0807C
Thread: 140257742817024, incremented this_is_global to: 1001
Thread: 140257734424320, pid: 1728, addresses: local: 0X4C6FEEDC, global: 0XE4D0807C
Thread: 140257734424320, incremented this_is_global to: 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
Parent: pid: 1728, local address: 0X763A38A8, global address: 0XE4D0807C
Child : pid: 1731, local address: 0X763A38A8, global address: 0XE4D0807C
Child : pid: 1731, set local_main to: 13; this_is_global to: 23
Parent: pid: 1728, local_main = 17, this_is_global = 17
```

Figure 6 example 4 output.

In this program, the parent sets 'this_is_global' to 1000 and then Two threads are created using. These threads increment the global variable this_is_global and print their respective thread ID, process ID, and addresses of local and global variables.

After the threads finish, the final value of this_is_global is printed.

The program then calls fork() to create a child process.

Both the parent and child processes initially have the same values for the local variable local_main and the global variable this_is_global. The parent process prints its process ID and addresses of local and global variables.

In the child process, the values of local_main and this_is_global are modified, and the child process prints its process ID and addresses of local and global variables.

The parent process waits for the child to finish and prints the final values of local_main and this_is_global.

Did this_is_global change after the threads have finished? Why?

Yes, it can be seen in Figure 6 that the global value that first started at 1000 is indeed incremented by both the two threads for it to be finally equal to 1002. Threads of the same process share the same global variables.

Are the local addresses the same in each thread? What about the global addresses?

The global addresses are indeed the same as they do share the same memory area for global variables. But each thread has its own stack and therefore need separate memory addresses in this aspect.

Did local_main and this_is_global change after the child process has finished? Why?

No, it didn't. when fork is used, a child process is created, and it takes a copy of local and global data from the parent process. They share the same values initially, but they are essentially different copies. So, changing the values in a process would not change it in the other.

Are the local addresses the same in each process? What about global addresses? What happened?

Yes they do share the addresses as shown in the last few lines of Figure 6. But what happened is that for both the parent and the child they have a separate copy for the global and local data. And so when the child modifies the data it doesn't get modified in the parent. Although they are in the same memory address, they are on separate memory pages that holds these copies separate for each process.

Example 5:

Running the compiled program, we get the following:

```
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 11640432
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 17322101
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 23132711
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 11539556
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 9354111
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 14531241
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 17249887
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 21694097
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 27254847
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 8823938
```

Figure 7 output of example 5.

14) How many times the line `tot_items = tot_items + *iptr;` is executed?

In this program the addition to the global variable happens in a for loop where its constant that its 50000 iterations for each thread. We have 50 threads therefore it would happen this many times:
number of iterations for the loop * number of threads = $50000 * 50 = 2500000$ times.

15) What values does `*iptr` have during these executions?

It would take numbers from 1 to 50, coding in a print statement would reveal this.
modifying the `thread_func` method:

```
void thread_func(void *ptr) {  
  
    int *iptr = (int *)ptr;
```



```
printf("iptr= %d\n", *iptr);
```

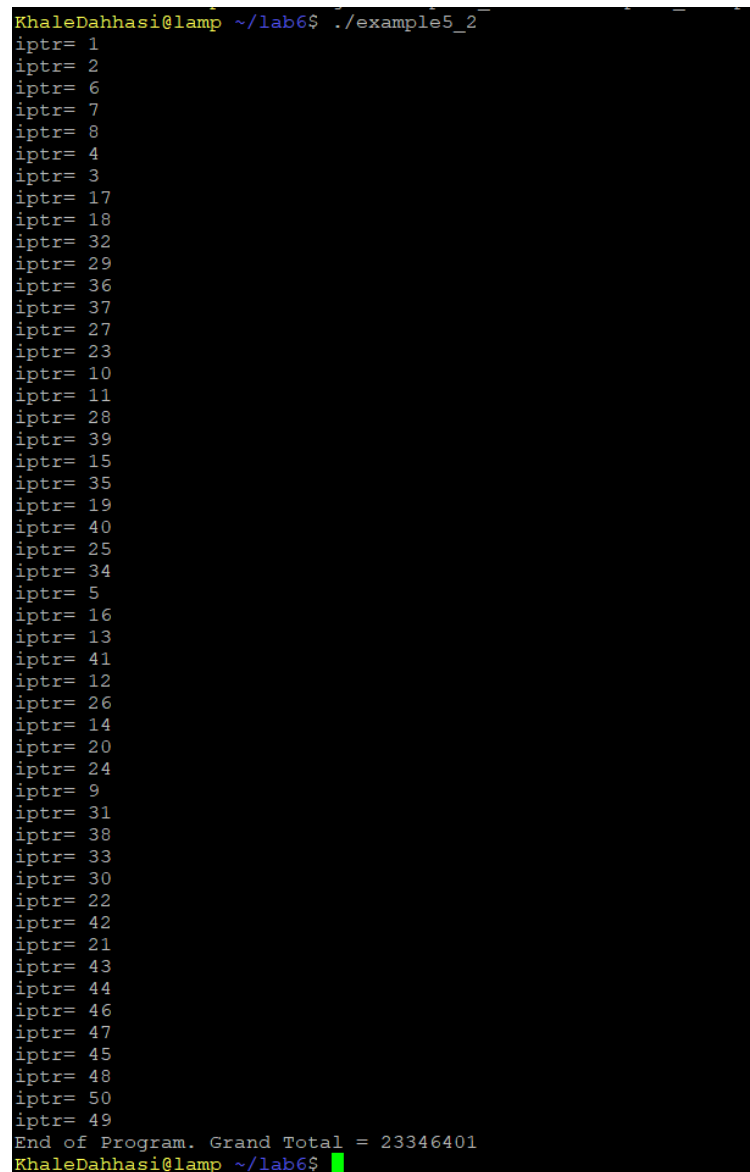
```
int n;
```

```
for(n = 50000; n--; )
```

```
tot_items = tot_items + *iptr; /* the global variable gets modified here */
```

```
}
```

The output:



```
KhaleDahhasi@lamp ~/lab6$ ./example5_2
iptr= 1
iptr= 2
iptr= 6
iptr= 7
iptr= 8
iptr= 4
iptr= 3
iptr= 17
iptr= 18
iptr= 32
iptr= 29
iptr= 36
iptr= 37
iptr= 27
iptr= 23
iptr= 10
iptr= 11
iptr= 28
iptr= 39
iptr= 15
iptr= 35
iptr= 19
iptr= 40
iptr= 25
iptr= 34
iptr= 5
iptr= 16
iptr= 13
iptr= 41
iptr= 12
iptr= 26
iptr= 14
iptr= 20
iptr= 24
iptr= 9
iptr= 31
iptr= 38
iptr= 33
iptr= 30
iptr= 22
iptr= 42
iptr= 21
iptr= 43
iptr= 44
iptr= 46
iptr= 47
iptr= 45
iptr= 48
iptr= 50
iptr= 49
End of Program. Grand Total = 23346401
KhaleDahhasi@lamp ~/lab6$
```

Figure 8 output of experimenting to get iptr.

We can conclude that iptr is the thread's order of creation as in the first created thread has this value at = 1 and for the last created thread it has this value at = 50.

16) What do you expect Grand Total to be?

As there would be a loop of 50000 iterations each thread, each time the thread's order of creation is added to itself 50000 times and then this is added to the same for the other threads. If we were to represent this in an equation it would be like the following:

$$\text{Grand total} = \sum_{iptr=1}^{50} (50000 \times iptr)$$

Where iptr starts at 1 and iterate to 50. This would sum up to 63750000.

17) Why you are getting different results?

Due to the race condition that happens where a thread takes the global total that isn't "updated" as in, it doesn't wait until the other threads are done with it. this causes the threads to contradict each other and add to an "old" grand total. The order in which the threads execute and update the grand total is non-deterministic, leading to different results on different runs. The race condition results in inconsistent updates and leads to different final values for the grand total.

Fixing using semaphores (Bonus):

If we added a semaphore to the code, thus protecting the shared value, we would get the correct expected grand total. The following is the modified code:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define NTIDS 50

/* This data is shared by the thread(s) */

int tot_items = 0;

struct tidrec {

    int data;

    pthread_t id;

};

/*This is the thread function */

sem_t semaphore;

void thread_func(void *ptr) {

    int *iptr = (int *)ptr;

    int n;

    for (n = 50000; n--;) {
```

```
    }}

int main() {

    struct tidrec tids[NTIDS];

    int m;

    // Initialize the semaphore

    sem_init(&semaphore, 0, 1);

    /* create as many threads as NTIDS */

    for (m = 0; m < NTIDS; ++m) {

        tids[m].data = m + 1;

        pthread_create(&tids[m].id, NULL, (void

    *)&thread_func, &tids[m].data);

    }

    /* wait for all the threads to finish */

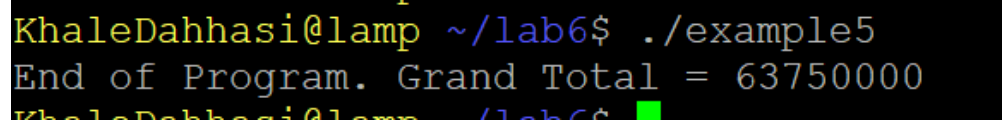
    for (m = 0; m < NTIDS; ++m)

        pthread_join(tids[m].id, NULL);

    printf("End of Program. Grand Total = %d\n",

    tot_items);
```

Output after fixing:

A terminal window with a black background and yellow text. The prompt is 'KhaleDahhasi@lamp ~/lab6\$'. The command './example5' has been executed. The output is 'End of Program. Grand Total = 63750000'. The prompt is partially visible on the next line.

```
KhaleDahhasi@lamp ~/lab6$ ./example5
End of Program. Grand Total = 63750000
KhaleDahhasi@lamp ~/lab6$
```

Figure 9 fixed program's correct output.