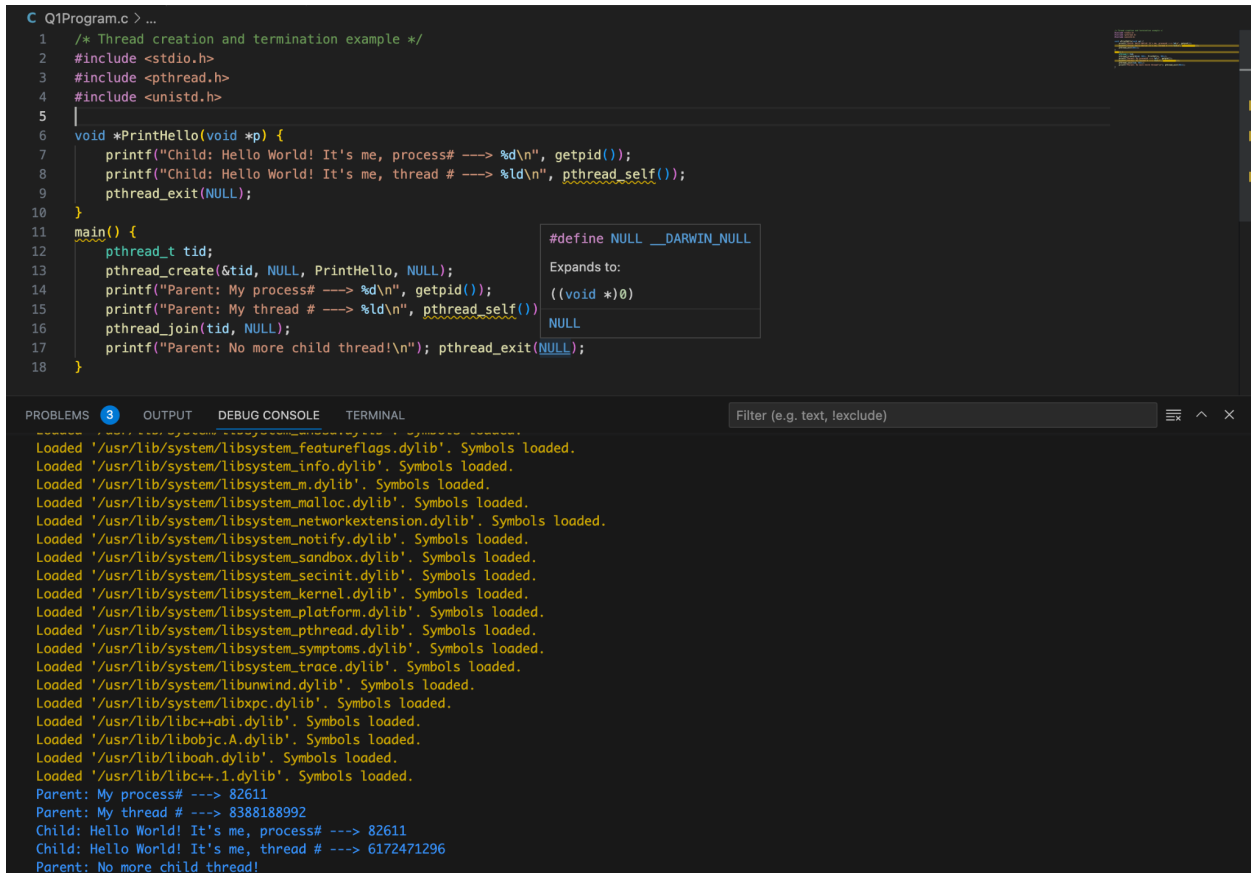


# LAB 6

Question 1 :: The following program demonstrates thread creation and termination:

1) Run the above program, and observe its output:



```
C: Q1Program.c > ...
1  /* Thread creation and termination example */
2  #include <stdio.h>
3  #include <pthread.h>
4  #include <unistd.h>
5
6  void *PrintHello(void *p) {
7      printf("Child: Hello World! It's me, process# ----> %d\n", getpid());
8      printf("Child: Hello World! It's me, thread # ----> %ld\n", pthread_self());
9      pthread_exit(NULL);
10 }
11
12 main() {
13     pthread_t tid;
14     pthread_create(&tid, NULL, PrintHello, NULL);
15     printf("Parent: My process# ----> %d\n", getpid());
16     printf("Parent: My thread # ----> %ld\n", pthread_self());
17     pthread_join(tid, NULL);
18     printf("Parent: No more child thread!\n"); pthread_exit(NULL);
19 }
```

Loaded '/usr/lib/system/libsystem\_featureflags.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_info.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_m.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_malloc.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_networkextension.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_notify.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_sandbox.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_secinit.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_kernel.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_platform.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_pthread.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_symptoms.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libsystem\_trace.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libunwind.dylib'. Symbols loaded.  
Loaded '/usr/lib/system/libxpc.dylib'. Symbols loaded.  
Loaded '/usr/lib/libc++abi.dylib'. Symbols loaded.  
Loaded '/usr/lib/libobjc.A.dylib'. Symbols loaded.  
Loaded '/usr/lib/liboah.dylib'. Symbols loaded.  
Loaded '/usr/lib/libc++abi.dylib'. Symbols loaded.  
Parent: My process# ----> 82611  
Parent: My thread # ----> 8388188992  
Child: Hello World! It's me, process# ----> 82611  
Child: Hello World! It's me, thread # ----> 6172471296  
Parent: No more child thread!

Output ::

```
Parent: My process# ----> 82611
Parent: My thread # ----> 8388188992
Child: Hello World! It's me, process# ----> 82611
Child: Hello World! It's me, thread # ----> 6172471296
Parent: No more child thread!
```

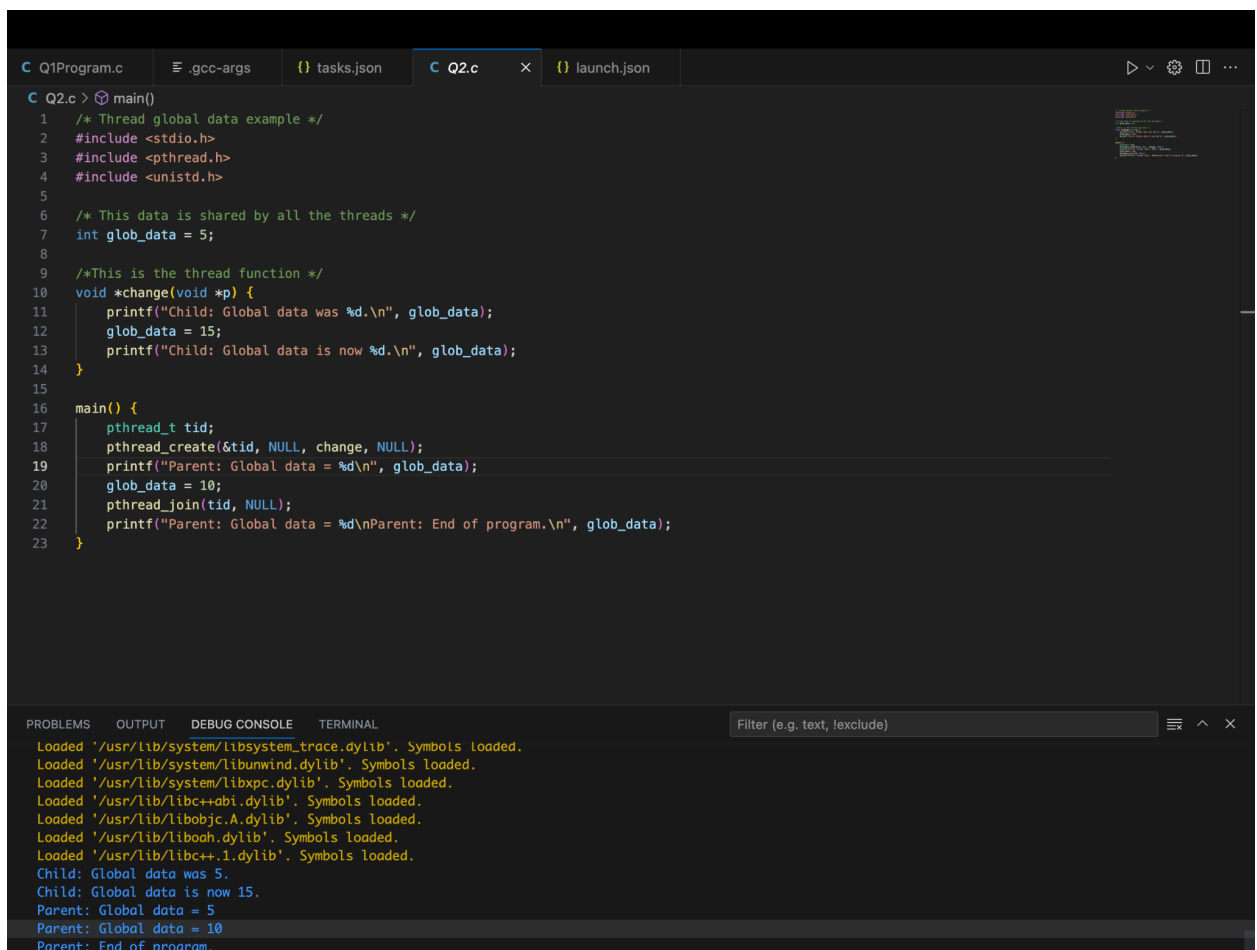
2) Are the process ID numbers of parent and child threads the same or different? Why?

Answer :: The process ID numbers of the parent and child threads are the same, both have the process ID 82611. The reason for this is that when a new thread is created using **pthread\_create**, it is created within the same process as the parent thread. The parent process spawns a new thread, but both the parent thread and the child thread share the same process ID. Each process is

assigned a unique process ID (PID) by the operating system, when a new thread is created within a process, the thread shares the same process ID as its parent.

**Question2)** :: The following program demonstrates thread global data:

3) Run the above program several times; observe its output every time. A sample output follows:



The screenshot shows a code editor with a C program named `Q2.c` and its terminal output. The program demonstrates thread global data using a shared variable `glob_data`.

```
1  /* Thread global data example */
2  #include <stdio.h>
3  #include <pthread.h>
4  #include <unistd.h>
5
6  /* This data is shared by all the threads */
7  int glob_data = 5;
8
9  /*This is the thread function */
10 void *change(void *p) {
11     printf("Child: Global data was %d.\n", glob_data);
12     glob_data = 15;
13     printf("Child: Global data is now %d.\n", glob_data);
14 }
15
16 main() {
17     pthread_t tid;
18     pthread_create(&tid, NULL, change, NULL);
19     printf("Parent: Global data = %d\n", glob_data);
20     glob_data = 10;
21     pthread_join(tid, NULL);
22     printf("Parent: Global data = %d\nParent: End of program.\n", glob_data);
23 }
```

The terminal output shows the execution of the program, including library loading messages and the sequence of prints from the child and parent threads:

```
Loaded '/usr/lib/system/libsystem_trace.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libunwind.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libxpc.dylib'. Symbols loaded.
Loaded '/usr/lib/libc++abi.dylib'. Symbols loaded.
Loaded '/usr/lib/libobjc.A.dylib'. Symbols loaded.
Loaded '/usr/lib/liboah.dylib'. Symbols loaded.
Loaded '/usr/lib/libc++1.dylib'. Symbols loaded.
Child: Global data was 5.
Child: Global data is now 15.
Parent: Global data = 5
Parent: Global data = 10
Parent: End of program.
```

Output 1

Child: Global data was 5.  
Child: Global data is now 15.  
Parent: Global data = 5  
Parent: Global data = 10  
Parent: End of program.

Output 2

Parent: Global data = 5

Child: Global data was 5.

Child: Global data is now 15.

Parent: Global data = 15

Parent: End of program.

#### 4) Does the program give the same output every time? Why?

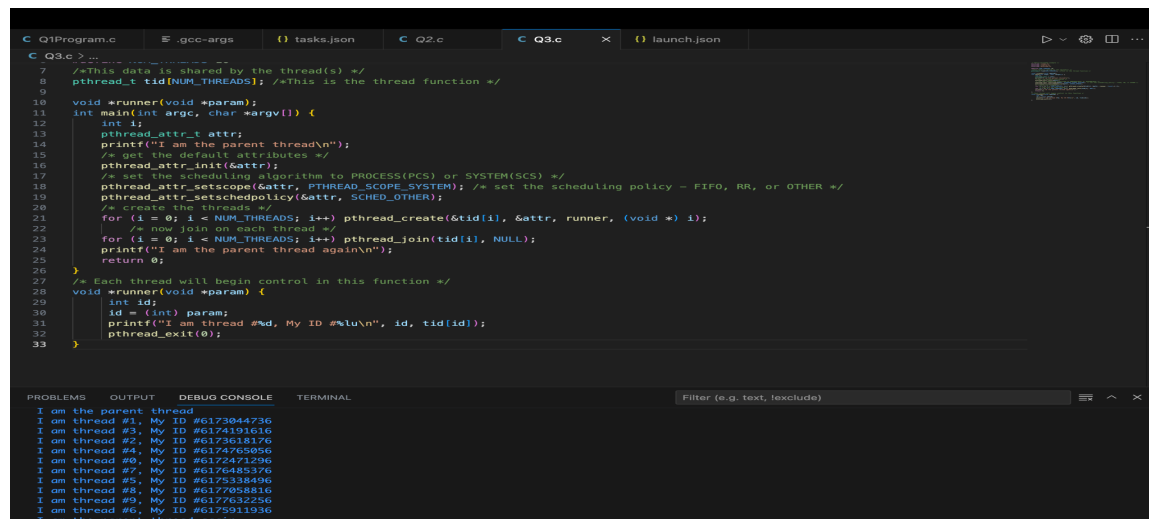
Answer -> The output of the program may not necessarily be the same every time. The behavior of concurrent threads can be non-deterministic, and it depends on the scheduling decisions made by the operating system. The main thread and the child thread both access and modify the global variable `glob\_data`. The order in which these threads execute, and the timing of their execution, can vary. Since both threads are accessing and modifying `glob\_data` concurrently without synchronization mechanisms like mutexes or atomic operations, the behavior is considered a data race. The output is not guaranteed to be predictable or consistent.

#### 5) Do the threads have separate copies of glob\_data?

No, the threads do not have separate copies of glob\_data. Threads within the same process share the same memory space, including global variables. When multiple threads access the same global variable, they are accessing the same memory location.

**Question 3) ::** The following example demonstrates a multi-threaded program:

#### 6) Run the above program several times and observe the outputs:



The screenshot shows a C++ IDE with a file named Q3.c. The code defines a global variable `glob\_data` and a function `runner` that takes a pointer to a thread ID and a parameter. The main function creates 10 threads, each calling `runner` with its own ID and a parameter. The output shows the execution of the parent thread and 10 child threads, each printing their ID and the value of `glob\_data`.

```
C Q3.c > ...
7  /*This data is shared by the thread(s) */
8  pthread_t tid[NUM_THREADS]; /*This is the thread function */
9
10 void *runner(void *param);
11 int main(int argc, char *argv[]) {
12     int i;
13     pthread_attr_t attr;
14     printf("I am the parent thread\n");
15     /* set the default attributes */
16     pthread_attr_init(&attr);
17     /* set the scheduling algorithm to PROCESS(PCS) or SYSTEM(SCS) */
18     pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); /* set the scheduling policy - FIFO, RR, or OTHER */
19     pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
20     /* create the threads */
21     for (i = 0; i < NUM_THREADS; i++) pthread_create(&tid[i], &attr, runner, (void *) i);
22     /* now join on each thread */
23     for (i = 0; i < NUM_THREADS; i++) pthread_join(tid[i], NULL);
24     printf("I am the parent thread again\n");
25     return 0;
26 }
27 /* Each thread will begin control in this function */
28 void *runner(void *param) {
29     int id;
30     id = (int) param;
31     printf("I am thread #%d, My ID #%lu\n", id, tid[id]);
32     pthread_exit(0);
33 }
```

OUTPUT

```
I am the parent thread
I am thread #1, My ID #6173044736
I am thread #2, My ID #6174191616
I am thread #3, My ID #6173618176
I am thread #4, My ID #6174765056
I am thread #5, My ID #6172247256
I am thread #6, My ID #6176485376
I am thread #7, My ID #6175338496
I am thread #8, My ID #6172054816
I am thread #9, My ID #6177632256
I am thread #10, My ID #6175011936
I am the parent thread again
```

#### Output1 ::

I am the parent thread

I am thread #1, My ID #6173044736

I am thread #3, My ID #6174191616

I am thread #2, My ID #6173618176  
I am thread #4, My ID #6174765056  
I am thread #0, My ID #6172471296  
I am thread #7, My ID #6176485376  
I am thread #5, My ID #6175338496  
I am thread #8, My ID #6177058816  
I am thread #9, My ID #6177632256  
I am thread #6, My ID #6175911936  
I am the parent thread again

**Output2 ::**

I am the parent thread  
I am thread #0, My ID #6172471296  
I am thread #1, My ID #6173044736  
I am thread #2, My ID #6173618176  
I am thread #3, My ID #6174191616  
I am thread #4, My ID #6174765056  
I am thread #5, My ID #6175338496  
I am thread #6, My ID #6175911936  
I am thread #7, My ID #6176485376  
I am thread #8, My ID #6177058816  
I am thread #9, My ID #6177632256  
I am the parent thread again

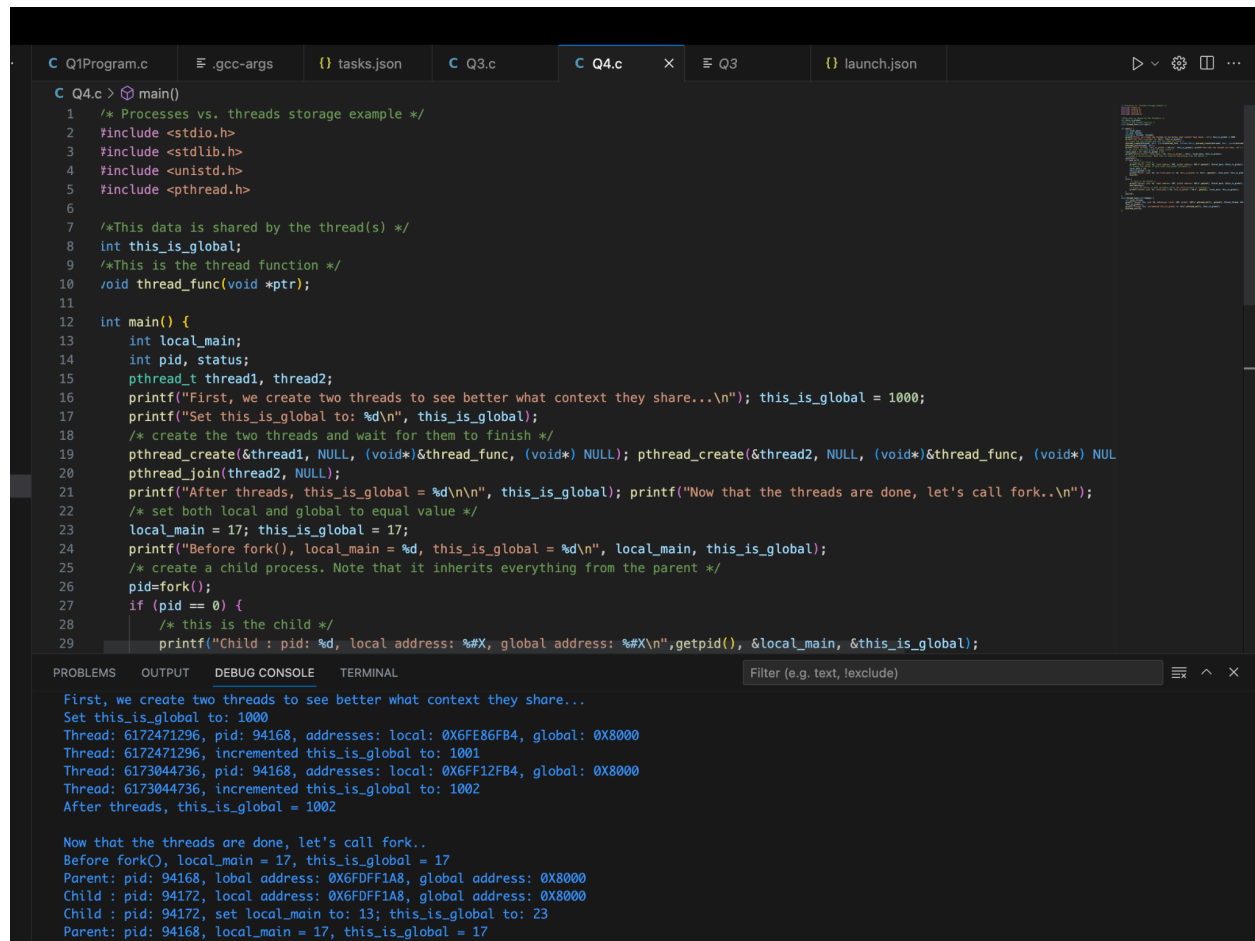
7) Do the output lines come in the same order every time? Why?

Answer -> No , The output will not be same everytime as, operating system's thread scheduler determines the order of thread execution based on factors such as the available CPU resources, the scheduling algorithm, and other system load conditions.

The scheduling algorithm used by the operating system may allocate CPU time to threads in a non-deterministic manner. This means that the order in which the threads are executed and the order in which their output is printed can differ from run to run.

**Question 4 ::** The following example demonstrates the difference between processes and threads with regards to how they use memory::

8) Run the above program and observe its output. Following is a sample output:



The screenshot shows a code editor with a C program named Q4.c. The code creates two threads, increments a global variable, and then forks a child process. The terminal output shows the execution flow, including thread creation, global variable updates, and process forking.

```
C Q4.c > main()
1  /* Processes vs. threads storage example */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <pthread.h>
6
7  /*This data is shared by the thread(s) */
8  int this_is_global;
9  /*This is the thread function */
10 void thread_func(void *ptr);
11
12 int main() {
13     int local_main;
14     int pid, status;
15     pthread_t thread1, thread2;
16     printf("First, we create two threads to see better what context they share...\n"); this_is_global = 1000;
17     printf("Set this_is_global to: %d\n", this_is_global);
18     /* create the two threads and wait for them to finish */
19     pthread_create(&thread1, NULL, (void*)&thread_func, (void*) NULL); pthread_create(&thread2, NULL, (void*)&thread_func, (void*) NULL);
20     pthread_join(thread2, NULL);
21     printf("After threads, this_is_global = %d\n\n", this_is_global); printf("Now that the threads are done, let's call fork..\n");
22     /* set both local and global to equal value */
23     local_main = 17; this_is_global = 17;
24     printf("Before fork(), local_main = %d, this_is_global = %d\n", local_main, this_is_global);
25     /* create a child process. Note that it inherits everything from the parent */
26     pid=fork();
27     if (pid == 0) {
28         /* this is the child */
29         printf("Child : pid: %d, local address: %#X, global address: %#X\n",getpid(), &local_main, &this_is_global);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Filter (e.g. text, lexclude)

```
First, we create two threads to see better what context they share...
Set this_is_global to: 1000
Thread: 6172471296, pid: 94168, addresses: local: 0X6FE86FB4, global: 0X8000
Thread: 6172471296, incremented this_is_global to: 1001
Thread: 6173044736, pid: 94168, addresses: local: 0X6FF12FB4, global: 0X8000
Thread: 6173044736, 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: 94168, llobal address: 0X6FDFF1A8, global address: 0X8000
Child : pid: 94172, local address: 0X6FDFF1A8, global address: 0X8000
Child : pid: 94172, set local_main to: 13; this_is_global to: 23
Parent: pid: 94168, local_main = 17, this_is_global = 17
```

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

Set this\_is\_global to: 1000

Thread: 6172471296, pid: 94168, addresses: local: 0X6FE86FB4, global: 0X8000

Thread: 6172471296, incremented this\_is\_global to: 1001

Thread: 6173044736, pid: 94168, addresses: local: 0X6FF12FB4, global: 0X8000

Thread: 6173044736, 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: 94168, llobal address: 0X6FDFF1A8, global address: 0X8000

Child : pid: 94172, local address: 0X6FDFF1A8, global address: 0X8000

Child : pid: 94172, set local\_main to: 13; this\_is\_global to: 23

Parent: pid: 94168, local\_main = 17, this\_is\_global = 17

9) Did `this_is_global` change after the threads have finished? Why?

Answer -> Yes, the value of `this_is_global` changed after the threads have finished. The value changes from 1000 to 1002, after both threads have executed, `this_is_global` becomes 1002.

The reason why `this_is_global` changes after the threads have finished is that all the threads, including the main thread and the child threads, share the same memory space. They can access and modify the same global variables.

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

Answer -> In the given code, the local addresses are not the same in each thread, while the global addresses are the same across all threads.

1. The `local_thread` variable is a local variable within the `thread_func()` function.

- Each thread has its own stack, and local variables are stored on the stack.
- Therefore, the `local_thread` variable has different addresses in each thread.
- The output shows different addresses for `local_thread` in each thread.

2. The `this_is_global` variable is a global variable shared by all threads.

- Global variables are stored in the data segment of the process's memory.
- All threads share the same data segment, so the global variable has the same address in all threads.
- The output shows the same address for `this_is_global` in all threads.

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

Answer-> No, the values of `local_main` and `this_is_global` did not change in the parent process after the child process finished. After the `fork()` system call, a new child process is created. The child process is an exact copy of the parent process, including all variables and their values at the time of the `fork()` call. However, The child process gets its own separate memory space, and any changes made in the child process do not affect the variables in the parent process.

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

Answer

1. **Local Addresses:**

- The `local_main` variable is a local variable within the `main()` function.
- In the parent process, the `local_main` variable has its own stack frame, and its address will be different from the address of `local_main` in the child process.
- Similarly, in the child process, the `local_main` variable has its own stack frame with a different address.

- The output shows different addresses for `local_main` in the parent and child processes.

## 2. Global Addresses:

- The ``this_is_global`` variable is a global variable shared by all processes.
- Global variables are stored in the data segment of each process's memory.
- Each process has its own separate memory space, including its own data segment, so the address of ``this_is_global`` will be different in each process.
- The output shows different addresses for ``this_is_global`` in the parent and child processes.

**Question 5::** The following example demonstrates what happens when multiple threads try to change global data:

**13)** Run the above program several times and observe the outputs, until you get different results.

```
C Q1Program.c  gcc-args  tasks.json  Q3.c  Q4.c  Q5.c  Q3  launch.json  > ⚙️ □ ...

C Q5.c > main()
1  /* multiple threads changing global data (racing) */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5  #define NTIDS 50
6  /*This data is shared by the thread(s) */
7  int tot_items = 0;
8  struct tidrec {
9      int data;
10     pthread_t id;
11 };
12 /*This is the thread function */
13 void thread_func(void *ptr) {
14     int *iptr = (int *)ptr;
15     int n;
16     for(n = 50000; n--;) {
17         tot_items = tot_items + *iptr;
18     }
19 }
20 int main() {
21     struct tidrec tids[NTIDS];
22     int m;
23     /* create as many threads as NTIDS */
24     for(m=0; m < NTIDS; ++m) {
25         tids[m].data = m+1;
26         /* the global variable gets modified here */
27         pthread_create(&tids[m].id, NULL, (void *) &thread_func, &tids[m].data);
28     }
29     /* wait for all the threads to finish */
30     for(m=0; m<NTIDS; ++m) {
31         pthread_join(tids[m].id, NULL);
32     }
33     printf("End of Program. Grand Total = %d\n", tot_items);
34 }
```

**Output 1::** End of Program. Grand Total = 11583008

**Output 2::** End of Program. Grand Total = 11919280

**Output 3::** End of Program. Grand Total = 12178154

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

There are 50 threads created in the `for` loop, and each thread executes a loop with `n = 50000`. Therefore, the line `tot\_items = tot\_items + \*iptr;` is executed **50000** times in each thread, resulting in a total of  $50 * 50000 = 2,500,000$  executions of the line across all threads.

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

The value of `\*iptr` during the executions of the line `tot\_items = tot\_items + \*iptr;` depends on the specific thread and iteration. Each thread is assigned a unique value in the range of 1 to 50 (`m+1`). The address of this value is passed as the argument `ptr` to the thread function `thread\_func`. Inside the function, `ptr` is casted to an `int` pointer `iptr`, and the value of `\*iptr` is accessed. In each execution of the line, `\*iptr` will have a value from 1 to 50, depending on the specific thread and iteration.

**16) What do you expect Grand Total to be?**

Assuming if there are no race conditions. Each thread performs 50,000 iterations, and within each iteration, it adds the value pointed to by `iptr` to the `tot\_items` variable. Since there are 50 threads, each with a unique value ranging from 1 to 50, the sum of these values would be  $(1 + 2 + 3 + \dots + 50) = 1,275$ . Therefore, the expected "Grand Total" would be  $50,000 \text{ iterations} * 1,275 = 1,250,000$ .

**17) Why you are getting different results?**

Multiple threads are concurrently modifying the shared variable `tot\_items` without any synchronization mechanism. This can result in a race condition where the threads may read and write to `tot\_items` simultaneously, leading to unpredictable and varying results.

The actual value of the "Grand Total" (`tot\_items`) will depend on the specific interleaving and scheduling of the threads during execution. Since there is no synchronization, the order and timing of the operations performed by each thread can vary, leading to different results each time the program is run.



