King Abdulaziz University – Faculty of Engineering - EE-463

Concurrent Process Creation and Execution

Multi-Threading

Operating Systems Lab#6

Name	ID
Abdulrahman Ayman Mekwar	**1937268**

```c
/* Thread creation and termination example */

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

void *PrintHello(void *p) {
    printf("Child: Hello World! It's me, process# ---> %d\n", getpid());
    printf("Child: Hello World! It's me, thread # ---> %ld\n", pthread_self());
    pthread_exit(NULL);
}


main() {
    pthread_t tid;
    pthread_create(&tid, NULL, PrintHello, NULL);
    printf("Parent: My process# ---> %d\n", getpid());
    printf("Parent: My thread # ---> %ld\n", pthread_self());
    pthread_join(tid, NULL);
    printf("Parent: No more child thread!\n");
    pthread_exit(NULL);
}
```

QUESTION NO#1: Run the above program, and observe its output

Output:

Parent: My process# ---> 195
Parent: My thread # ---> 140257018722112
Child: Hello World! It's me, process# ---> 195
Child: Hello World! It's me, thread # ---> 140257018717760
Parent: No more child thread!

QUESTION NO#2: Are the process ID numbers of parent and child threads the same or different? Why?

Your second question first: In a multi-threaded program, all threads share the same process ID. This is because threads are actually a part of a process. Each process has its own unique process ID, and all threads created by that process will share the same process ID. This is one of the key differences between processes and threads. Processes are independent execution units with their own memory space, while threads are lighter-weight execution units that share the same memory space within a process.

```c
/* Thread global data example */

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

/* This data is shared by all the threads */
int glob_data = 5;

/*This is the thread function */
void *change(void *p) {
    printf("Child: Global data was %d.\n", glob_data);
    glob_data = 15;
    printf("Child: Global data is now %d.\n", glob_data);
}

main() {
    pthread_t tid;
    pthread_create(&tid, NULL, change, NULL);
    printf("Parent: Global data = %d\n", glob_data);
    glob_data = 10;
    pthread_join(tid, NULL);
    printf("Parent: Global data = %d\nParent: End of program.\n", glob_data);
}
```

QUESTION NO#3: Run the above program several times; observe its output every time. A sample output follows

Output:

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

Other Output:

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

QUESTION NO#4: Does the program give the same output every time? Why?

No, the program doesn't necessarily give the same output every time, and this is due to the nature of threading and the scheduling of threads by the operating system.

In a multithreaded program, the order in which threads are executed is determined by the operating system's scheduler, which can be non-deterministic. This means that you can't predict the order of execution with certainty.

In this program, there's a race condition with the global variable 'glob_data'. Both the main thread and the newly created thread are trying to access and modify this variable. Depending on the order in which the threads are scheduled, one of the following scenarios might occur:

1. The main thread might print the initial value of 'glob_data', then change it, and then the new thread might run, print the updated value, change it again, and then the main thread might print the final value.

2. The new thread might run first, print the initial value of 'glob_data', change it, and then the main thread might run, print the updated value, change it again, and then print the final value.

3. The main thread might print the initial value of 'glob_data', and then the new thread might run, print the updated value, change it, and the main thread may finally update the value and print it.

These are just a few possible scenarios. The exact order of execution can vary from run to run. This is why multithreaded programs often need to use synchronization mechanisms, like mutexes or semaphores, to control access to shared data and ensure that operations are carried out in a consistent and predictable manner.

QUESTION NO#5: Do the threads have separate copies of glob_data?

No, threads within the same process do not have separate copies of global data. Threads share the same address space, which means that they have access to the same memory and can both read and modify the same variables. This is one of the fundamental properties of threads, and it differentiates them from processes.

In this example, 'glob_data' is a global variable, which means it's accessible from anywhere in the program, including from inside any threads that you create. When one thread changes the value of 'glob_data', that change is visible to all other threads, because they're all looking at the exact same piece of memory

This shared access can lead to issues when multiple threads want to read or write the same variable at the same time, which is known as a race condition. To avoid race conditions, you often need to use synchronization mechanisms, like mutexes or semaphores, to ensure that only one thread can access a piece of data at a time.

```c
/* Multi-threaded example */

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

#define NUM_THREADS 10

/*This data is shared by the thread(s) */
pthread_t tid[NUM_THREADS];

/*This is the thread function */
void *runner(void *param);

int main(int argc, char *argv[]) {
    int i;
    pthread_attr_t attr;
    printf("I am the parent thread\n");

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* set the scheduling algorithm to PROCESS(PCS) or SYSTEM(SCS) */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* set the scheduling policy - FIFO, RR, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);

    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, (void *) i);

    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);

    printf("I am the parent thread again\n");
    return 0;
}

/* Each thread will begin control in this function */
void *runner(void *param) {
    int id;
    id = (int) param;

    printf("I am thread #%d, My ID #%lu\n", id, tid[id]);
    pthread_exit(0);
}
```

QUESTION NO#6: Run the above program several times and observe the outputs:

Output:

I am the parent thread
I am thread #2, My ID #139954938009152
I am thread #7, My ID #139954896045632
I am thread #6, My ID #139954904438336
I am thread #5, My ID #139954912831040
I am thread #1, My ID #139954946401856
I am thread #0, My ID #139954954794560
I am thread #3, My ID #139954929616448
I am thread #4, My ID #139954921223744
I am thread #8, My ID #139954887652928
I am thread #9, My ID #139954879129152
I am the parent thread again

QUESTION NO#7: Do the output lines come in the same order every time? Why?

No, the output lines from the individual threads don't necessarily come in the same order every time you run the program. The reason for this is due to the nature of multithreaded programming and the scheduling of threads by the operating system.

When you create multiple threads, the order in which they get executed is not guaranteed. This is because the operating system's scheduler decides which threads to run at what time, and this decision can be influenced by various factors like system load, priority of threads, etc. This is called concurrent execution and is a fundamental characteristic of multithreaded systems.

In your program, although you are creating threads in a certain order, they are likely being scheduled to run concurrently. The operating system's scheduler might switch between threads in an unpredictable order, leading to the "I am thread #..., My ID #..." output lines being printed in a different order each time you run the program.

This is a common scenario in multithreaded programming and is often the source of race conditions, where the behavior of a program depends on the relative timing of threads. In these cases, synchronization constructs like mutexes or semaphores are used to ensure that certain sections of code are executed in a particular order.

Others output:

```
I am the parent thread
I am thread #0, My ID #140662544639552
I am thread #1, My ID #140662536246848
I am thread #2, My ID #140662527854144
I am thread #3, My ID #140662519461440
I am thread #6, My ID #140662494152256
I am thread #4, My ID #140662510937664
I am thread #7, My ID #140662485759552
I am thread #5, My ID #140662502544960
I am thread #8, My ID #140662477366848
I am thread #9, My ID #140662393534016
I am the parent thread again

I am the parent thread
I am thread #0, My ID #139869471675968
I am thread #1, My ID #139869463283264
I am thread #2, My ID #139869454890560
I am thread #5, My ID #139869359695424
I am thread #3, My ID #139869446366784
I am thread #6, My ID #139869351302720
I am thread #7, My ID #139869342910016
I am thread #8, My ID #139869334517312
I am thread #9, My ID #139869326124608
I am thread #4, My ID #139869368088128
I am the parent thread again
```

```c
/* Processes vs. threads storage example */

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

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

/*This is the thread function */
void thread_func(void *ptr);

int main() {
    int local_main;
    int pid, status;
    pthread_t thread1, thread2;

    printf("First, we create two threads to see better what context they share...\n");
    this_is_global = 1000;
    printf("Set this_is_global to: %d\n", this_is_global);

    /* create the two threads and wait for them to finish */
    pthread_create(&thread1, NULL, (void*)&thread_func, (void*) NULL);
    pthread_create(&thread2, NULL, (void*)&thread_func, (void*) NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("After threads, this_is_global = %d\n\n", this_is_global);
    printf("Now that the threads are done, let's call fork..\n");

    /* set both local and global to equal value */
    local_main = 17;
    this_is_global = 17;

    printf("Before fork(), local_main = %d, this_is_global = %d\n",
            local_main, this_is_global);

    /* create a child process. Note that it inherits everything from the parent */
    pid=fork();

    if (pid == 0) {                                          /* this is the child */
      printf("Child : pid: %d, local address: %#X, global address: %#X\n",
              getpid(), &local_main, &this_is_global);

      /* change the values of both local and global variables */
      local_main = 13;
      this_is_global = 23;

      printf("Child : pid: %d, set local_main to: %d; this_is_global to: %d\n",
              getpid(), local_main, this_is_global);
      exit(0);
    }
    else {                                                   /* this is the parent */
      printf("Parent: pid: %d, lobal address: %#X, global address: %#X\n",
              getpid(), &local_main, &this_is_global);
      wait(&status);

      /* print the values of both variables after the child process is finished */
      printf("Parent: pid: %d, local_main = %d, this_is_global = %d\n",
              getpid(), local_main, this_is_global);
    }
    exit(0);
}

void thread_func(void *dummy) {
    int local_thread;
    printf("Thread: %lu, pid: %d, addresses: local: %#X, global: %#X\n",
            pthread_self(), getpid(), &local_thread, &this_is_global);

    /* increment the global variable */
    this_is_global++;

    printf("Thread: %lu, incremented this_is_global to: %d\n",
            pthread_self(), this_is_global);

    pthread_exit(0);
}
```

QUESTION NO#8: Run the above program and observe its output. Following is a sample output:

Output:

```
First, we create two threads to see better what context they share...
Set this_is_global to: 1000
Thread: 139946706818624, pid: 3757, addresses: local: 0XE1BF4E34, global:
0X12B25014
Thread: 139946706818624, incremented this_is_global to: 1001
Thread: 139946715211328, pid: 3757, addresses: local: 0XE23F5E34, global:
0X12B25014
Thread: 139946715211328, 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: 3757, lobal address: 0XFD0EEB0C, global address: 0X12B25014
Child : pid: 3763, local address: 0XFD0EEB0C, global address: 0X12B25014
Child : pid: 3763, set local_main to: 13; this_is_global to: 23
Parent: pid: 3757, local_main = 17, this_is_global = 17
```

QUESTION NO#9: Did this_is_global change after the threads have finished? Why

Yes, the value of 'this_is_global' did change after the threads have finished, as can be seen in the output: `After threads, this_is_global = 1002`.

This change occurs because threads share global variables. In your program, each thread increments 'this_is_global' by 1, so the value of the variable increases by 1 for each thread execution. Since two threads are created and each thread increments the value of 'this_is_global', the final value after both threads have executed is 'this_is_global + 2'.

This shared access to global variables among threads is a fundamental characteristic of threading. It allows threads to communicate with each other more easily than separate processes can, but it also means that you have to be careful to avoid race conditions, where the outcome depends on the relative timing of threads' accesses to shared data.

For instance, if both threads tried to increment 'this_is_global' at the same time, they could both read the value, increment it, and write it back, causing one of the increments to be lost. This is one reason why synchronization mechanisms, such as mutexes, are often used when multiple threads need to modify a shared variable.

QUESTION NO#10: Are the local addresses the same in each thread? What about the global addresses?

The local addresses in each thread are typically different because each thread has its own stack, where it keeps its local variables. The local variable in one thread is not shared with any other thread, so it gets a unique address in that thread's stack.

On the other hand, the global addresses are the same in each thread. This is because all threads in a process share the same global memory space. Any global variable declared in the program is accessible to all threads and is stored at the same memory address for all of them.

In your example program output, you can see that the addresses of the global variable 'this_is_global' are the same in both threads ('0X12B25014'), which shows that they are accessing the same global variable. However, the addresses of the local variable 'local_thread' are different in each thread ('0XE1BF4E34' and ' 0XE1BF4E34'), showing that these are separate local variables in each thread's own stack.

This distinct separation of local and global variables among threads is one of the key characteristics of multi-threading and is vital to understanding how to effectively implement and use threads in concurrent programming.

QUESTION NO#11: Did local_main and this_is_global change after the child process has finished? Why?

No, the variables 'local_main' and 'this_is_global' in the parent process do not change after the child process has finished. This is because when a new process is created using fork(), it does not share memory with the parent process. Instead, it gets a complete copy of the parent's memory space. This includes all code, global variables, heap memory, and stack memory.
In this example, after fork(), the child process changes the value of 'local_main' and 'this_is_global'. However, these changes only affect the child process's own copies of these variables. The parent process's copies of 'local_main' and 'this_is_global' remain unchanged. This is evident from the output:

```
Parent: pid: 3757, local_main = 17, this_is_global = 17

Child : pid: 3763, set local_main to: 13; this_is_global to: 23

Parent: pid: 3757, local_main = 17, this_is_global = 17
```

The values of 'local_main' and 'this_is_global' in the parent process remain as '17' and '17', respectively, even after the child process changes its copies of these variables. This demonstrates the isolation between parent and child processes in terms of memory.

It's important to note that while threads of the same process share memory, different processes do not share memory by default. Inter-process communication (IPC) mechanisms are needed for sharing data between different processes, which is typically more complex and slower than sharing data between threads within the same process.

QUESTION NO#12:

In the case of processes, the local addresses and global addresses are different in each process. This is because processes have their own separate memory spaces, which include distinct stacks for local variables and separate memory regions for global variables.

When a process is created through 'fork()', a new address space is allocated for the child process, which is a copy of the parent process's address space. However, the addresses in the child process are different from the addresses in the parent process. This means that the local variables in the child process will have different addresses compared to the parent process.

Similarly, the global variables in each process have their own distinct memory addresses. Each process has its own copy of the global variables, and changes made to the global variables in one process do not affect the values in another process.

In your example program, after the 'fork()' call, the child process changes the values of 'local_main' and 'this_is_global'. However, these changes only affect the child process's own copies of these variables. The parent process retains its original values. This can be seen in the output:

```
Parent: pid: 3757, local_main = 17, this_is_global = 17

Child : pid: 3763, set local_main to: 13; this_is_global to: 23

Parent: pid: 3757, local_main = 17, this_is_global = 17
```

The addresses of the local variables ('local_main`) are different between the parent and child processes. Similarly, the addresses of the global variable ('this_is_global') are different in each process. This demonstrates the separate memory spaces and address spaces of different processes.

The fork operation creates a new process with its own memory space, allowing each process to have independent values for local and global variable

```c
/* multiple threads changing global data (racing) */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.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 */
void thread_func(void *ptr) {
    int *iptr = (int *)ptr;
    int n;

    for(n = 50000; n--; )
        tot_items = tot_items + *iptr;    /* the global variable gets modified here /*
}

int main() {
    struct tidrec tids[NTIDS];
    int m;

    /* 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);
}
```

QUESTION NO#13: Run the above program several times and observe the outputs, until you get different results

Output:

End of Program. Grand Total = 46063636
End of Program. Grand Total = 35095376
End of Program. Grand Total = 51821105
End of Program. Grand Total = 25911979
End of Program. Grand Total = 20962659
End of Program. Grand Total = 26284554
End of Program. Grand Total = 18903036
End of Program. Grand Total = 45124185
End of Program. Grand Total = 52681594
End of Program. Grand Total = 36225920

QUESTION NO#14:

In the given code, the line 'tot_items = tot_items + *iptr;' is executed in a loop with 50,000 iterations in each thread. The loop runs from 'n = 50000' down to 'n > 0', and the line is executed for each iteration of the loop.

Since there are 50 threads created in the main function ('NTIDS = 50'), each running the loop with 50,000 iterations, the line 'tot_items = tot_items + *iptr;' will be executed a total of 50 * 50,000 = 2,500,000 times.

QUESTION NO#15: What values does *iptr have during these executions

In the given code, each thread's '*iptr' value is passed as a pointer to the 'thread_func' function, and it corresponds to the value of 'tids[m].data' for that specific thread. The 'tids[m].data' value is set in the main function as 'm + 1' for each thread.

Since there are 50 threads created in the main function ('NTIDS = 50'), the '*iptr' values during the executions will range from 1 to 50, corresponding to the respective 'tids[m].data' values.

For example, in the first thread (m = 0), the '*iptr' value will be 1. In the second thread (m = 1), the '*iptr' value will be 2, and so on, until the last thread (m = 49), where the '*iptr' value will be 50.

QUESTION NO#16: What do you expect Grand Total to be?

The final value of 'tot_items' will be the sum of all the values added by each thread over the iterations. Specifically, it will be the sum of the values 1 to 50, repeated 50,000 times for each thread.

The expected value for the "Grand Total" can be calculated as follows:

```
Grand Total = (1 + 2 + 3 + ... + 50) * 50,000
```

The sum of the values 1 to 50 is 1,275, and multiplying it by 50,000 gives us the expected "Grand Total."

Therefore, the expected "Grand Total" in this case would be 1,275 * 50,000 = 63,750,000.

QUESTION NO#17: Why you are getting different results?

There are several reasons why you may be getting different results each time you run the program:

1. **Concurrency and scheduling** : The threads in the program are executed concurrently,and the order in which they are scheduled to run is determined by the operating system's thread scheduler. The scheduler's decisions can be influenced by factors such as system load, thread priorities, and other system events. As a result, the threads may execute in different orders each time, leading to different results.

2. **Race conditions**: The program is vulnerable to race conditions because multiple threads are accessing and modifying the shared variable 'tot_items' simultaneously. When threads race to update the variable, the outcome depends on the relative timing of their operations. The interleaving of thread execution can vary between runs, leading to different results.

3. **Lack of synchronization**: The code does not include any explicit synchronization mechanisms, such as mutexes or atomic operations, to coordinate access to the shared variable. Without proper synchronization, multiple threads can access and modify 'tot_items' concurrently, causing data inconsistencies and unpredictable results.

4. **Non-atomic updates**: The line 'tot_items = tot_items + *iptr;' involves a read- modify-write operation on the 'tot_items' variable. If this operation is not performed atomically, it can result in inconsistent or unexpected behavior when multiple threads try toupdate 'tot_items' simultaneously.

Given these factors, the concurrent and non-deterministic nature of thread execution, the presence of race conditions, and the lack of synchronization, it is expected to observe different results each time the program is run.