**ICS 431 Operating Systems**

**Lab 05: Threads Creation and Execution**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Objective :**

This lab examines aspects of **threads** and **multiprocessing** (and **multithreading**). The primary objective of this lab is to implement the Thread Management Functions:

**Creating Threads**

**Terminating Thread Execution**

**Passing Arguments To Threads**

**Thread Identifiers**

**Joining Threads**

**Detaching / Undetaching Threads**

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**What is thread ?**

A **thread** is a semi-process, that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

**What are pthreads?**

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.

**Pthreads** are defined as a set of **C language programming types and procedure calls**. Vendors usually provide a Pthreads implementation in the form of a **header/include file** and a library which you **link** with your program.

**The pthreads API :**

The subroutines which comprise the Pthreads API can be informally grouped into three major classes:

**Thread management:** The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

**Mutexes:** The second class of functions deal with a coarse type of synchronization, called a "mutex", which is an abbreviation for "**mutual exclusion**". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

**Condition variables:** The third class of functions deal with a finer type of synchronization - based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

**Naming conventions:** All identifiers in the threads library begin with **pthread\_**

|  |  |
| --- | --- |
| **pthread\_** | **Threads themselves and miscellaneous subroutines** |
| **pthread\_attr** | **Thread attributes objects** |
| **pthread\_mutex** | **Mutexes** |
| **pthread\_mutexattr** | **Mutex attributes objects.** |
| **pthread\_cond** | **Condition variables** |
| **pthread\_condattr** | **Condition attributes objects** |
| **pthread\_key** | **Thread-specific data keys** |

**Thread Management Functions :**

The function **pthread\_create** is used to create a new thread, and the function **pthread\_exit** is used by a thread to terminate itself. The function **pthread\_join** is used by a thread to wait for termination of another thread.

|  |  |
| --- | --- |
| Function: | int **pthread\_create**  (  pthread\_t \* threadhandle, /\* Thread handle returned by reference \*/  pthread\_attr\_t \*attribute, /\* Special Attribute for starting thread, may be NULL \*/ void \*(\*start\_routine)(void \*), /\* Main Function which thread executes \*/ void \*arg /\* An extra argument passed as a pointer \*/  ); |
| Info: | Request the **PThread** library for **creation** of a new thread. The return value is **0** on **success**. The return value is **negative** on **failure**. The **pthread\_t** is an abstract datatype that is used as a handle to **reference** the thread. |

|  |  |
| --- | --- |
| Function: | void **pthread\_exit**  (  void \*retval /\* return value passed as a pointer \*/  ); |
| Info: | This Function is used by a thread to **terminate**. The return value is passed as a **pointer**. This pointer value can be anything so long as it does not exceed the size of (void \*). Be careful, this is system dependent. You may wish to return an address of a structure, if the returned data is very large. |

|  |  |
| --- | --- |
| Function: | int **pthread\_join**  (  pthread\_t threadhandle, /\* Pass threadhandle \*/  void \*\*returnvalue /\* Return value is returned by ref. \*/  ); |
| Info: | Return **0** on **success**, and **negative** on **failure**. The returned value is a **pointer** returned by **reference**. If you do not care about the return value, you can pass **NULL** for the second argument. |

**Thread Initialization :**

**Include the pthread.h library :**

**#include <pthread.h>**

**Declare a variable of type pthread\_t :**

**pthread\_t the\_thread**

**When you compile, add -lpthread to the linker flags :**

**cc or gcc threads.c -o threads -lpthread**

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Initially, threads are created from within a process. Once created, threads are peers, and may create other threads. Note that an "initial thread" exists by default and is the thread which runs main.**

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Terminating Thread Execution :**

## int pthread\_cancel(pthread\_t thread)

**pthread\_cancel** sends a cancellation request to the thread denoted by the **thread** argument. If there is no such thread, **pthread\_cancel** fails. Otherwise it returns **0**. A cancel is a mechanism by which a calling thread informs either itself or the called thread to terminate as quickly as possible. Issuing a cancel does not guarantee that the canceled thread receives or handles the cancel. The canceled thread can delay processing the cancel after receiving it. For instance, if a cancel arrives during an important operation, the canceled thread can continue if what it is doing cannot be interrupted at the point where the cancel is requested.

The programmer may specify a termination status, which is stored as a void pointer for any thread that may join the calling thread.

**There are several ways in which a Pthread may be terminated:**

The thread returns from its starting routine (the main routine for the initial thread). By default, the Pthreads library will reclaim any system resources used by the thread. This is similar to a process terminating when it reaches the end of main.

The thread makes a call to the pthread\_exit subroutine (covered below).

The thread is canceled by another thread via the pthread\_cancel routine (not covered here).

The thread receives a signal that terminates it

The entire process is terminated due to a call to either the exec or exit subroutines.

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Thread Attributes :**

Threads have a number of attributes that may be set at creation time. This is done by filling a thread attribute object **attr** of type **pthread\_attr\_t**, then passing it as second argument to **pthread\_create**. Passing **NULL** is equivalent to passing a thread attribute object with all attributes set to their **default values**.

Attribute objects are consulted only when creating a new thread. The same attribute object can be used for creating several threads. Modifying an attribute object after a call to **pthread\_create** does not change the attributes of the thread previously created.

## int pthread\_attr\_init (pthread\_attr\_t \*attr)

**pthread\_attr\_init** initializes the thread attribute object **attr** and fills it with default values for the attributes. Each attribute **attrname** can be individually set using the function **pthread\_attr\_setattrname** and retrieved using the function **pthread\_attr\_getattrname**.

## int pthread\_attr\_destroy (pthread\_attr\_t \*attr)

**pthread\_attr\_destroy** destroys the attribute object pointed to by **attr** releasing any resources associated with it. **attr** is left in an undefined state, and you must not use it again in a call to any pthreads function until it has been reinitialized.

## int pthread\_attr\_setattr (pthread\_attr\_t \*obj, int value)

Set attribute **attr** to value in the attribute object pointed to by **obj**. See below for a list of possible attributes and the values they can take. On success, these functions return 0.

**int pthread\_attr\_getattr (const pthread\_attr\_t \*obj, int \*value)**  Store the current setting of **attr** in **obj** into the variable pointed to by **value**. These functions always return 0.

**The following thread attributes are supported:**

**`detachstate'**

Choose whether the thread is created in the joinable state (value PTHREAD\_CREATE\_JOINABLE) or in the detached state (PTHREAD\_CREATE\_DETACHED). The default is PTHREAD\_CREATE\_JOINABLE.

In the joinable state, another thread can synchronize on the thread termination and recover its termination code using pthread\_join, but some of the thread resources are kept allocated after the thread terminates, and reclaimed only when another thread performs pthread\_join on that thread. In the detached state, the thread resources are immediately freed when it terminates, but pthread\_join cannot be used to synchronize on the thread termination. A thread created in the joinable state can later be put in the detached thread using pthread\_detach.

**`schedpolicy'**

Select the scheduling policy for the thread: one of SCHED\_OTHER (regular, non-realtime scheduling),

SCHED\_RR (realtime, round-robin) or SCHED\_FIFO (realtime, first-in first-out). The default is SCHED\_OTHER. The realtime scheduling policies SCHED\_RR and SCHED\_FIFO are available only to processes with superuser privileges. pthread\_attr\_setschedparam will fail and return ENOTSUP if you try to set a realtime policy when you are unprivileged. The scheduling policy of a thread can be changed after creation with pthread\_setschedparam.

**`schedparam'**

Change the scheduling parameter (the scheduling priority) for the thread. The default is 0. This attribute is not significant if the scheduling policy is SCHED\_OTHER; it only matters for the realtime policies SCHED\_RR and SCHED\_FIFO. The scheduling priority of a thread can be changed after creation with pthread\_setschedparam.

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Thread Identifiers :** **pthread\_self ( )**

Returns the unique thread ID of the calling thread.

The returned data object is opaque can not be easily inspected. **pthread\_equal (thread1, thread2)**

Compares two thread IDs:

If the two IDs are different 0 is returned, otherwise a non-zero value is returned.

Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs.

**Example: Pthread Creation and Termination:**

## Lab5\_1.c

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

void \*kidfunc(void \*p) {

    printf ("Kid ID is ---> %d\n", getpid());

    }

int main (void) {

    pthread\_t kid;

    pthread\_create(&kid, NULL, kidfunc, NULL);

    printf("Parent ID is ---> %d\n", getpid());

    pthread\_join(kid, NULL);

    printf("No more kid!\n") ;

}

**Sample output**

**Parent ID is ---> 29085 Kid ID is ---> 29085 No more kid!**

**Are the process id numbers of parent and child thread the same or different?**

**No, because the thread shares data with the process.**

## Lab5\_2.c

#include <stdio.h>

#include <pthread.h>

int glob\_data = 5 ;

void \*kidfunc(void \*p)

{

    printf ("Kid here. Global data was %d.\n", glob\_data) ;

    glob\_data = 15 ;

    printf ("Kid Again. Global data was now %d.\n", glob\_data) ;

}

int main(void)

{

    pthread\_t kid ;

    pthread\_create (&kid, NULL, kidfunc, NULL);

    printf ("Parent here. Global data = %d\n", glob\_data);

    glob\_data = 10;

    pthread\_join (kid, NULL) ;

    printf ("End of program. Global data = %d\n", glob\_data);

}

**Sample output**

### Parent here. Global data = 5

**Kid here. Global data was 10.**

**Kid Again. Global data was now 15.**

### End of program. Global data = 15

**Do the threads have separate copies of glob\_data?**

No, global data accessible in the thread is the same as the process.

**""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""** **Multiple Threads :**

The simple example code below creates 5 threads with the **pthread\_create( )** routine. Each thread prints a "Hello World!" message, and then terminates with a call to **pthread\_exit( )**.

### Lab5\_3.c

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM\_THREADS 5

void \*PrintHello(void \*threadid)

{

    printf("\n%d: Hello World!\n", threadid);

    pthread\_exit(NULL);

}

int main(void)

{

    pthread\_t threads[NUM\_THREADS];

    int rc, t;

    for(t = 0; t < NUM\_THREADS; t++)    {

        printf("Creating thread %d\n", t);

        rc = pthread\_create(&threads[t], NULL, PrintHello, (void \*) t);                  if (rc) {

            printf("ERROR; return code from pthread\_create() is %d\n", rc);                   exit(-1);

        }

    }

    pthread\_exit(NULL);

}

**Difference between process and threads :**

The process creates the thread, but the thread takes time to create and function. The thread executes after its creation is done.

## Lab5\_4.c

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

int this\_is\_global;

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 = %d\n", this\_is\_global);

    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",this\_is\_global);

    printf("\n");

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

    local\_main = 17; this\_is\_global = 17;

    printf("Before fork(), local\_main = %d, this\_is\_global = %d\n",local\_main, this\_is\_global);

    pid=fork();

    if(pid == 0) { /\* this is the child \*/

        printf("In child, pid %d: &global: %ls, &local: %ls\n", getpid(), &this\_is\_global, &local\_main);

        local\_main = 13;

        this\_is\_global = 23;

        printf("Child set local main=%d, this\_is\_global=%d\n",local\_main, this\_is\_global);

        exit(0);

    }

    else { /\* this is parent \*/

        printf("In parent, pid %d: &global: %ls, &local: %ls\n", getpid(), &this\_is\_global, &local\_main);

        wait(&status);

        printf("In parent, local\_main=%d, this\_is\_global=%d\n",local\_main, this\_is\_global);

    }

    exit(0);

}

void thread\_func(void \*dummy) {

    int local\_thread;

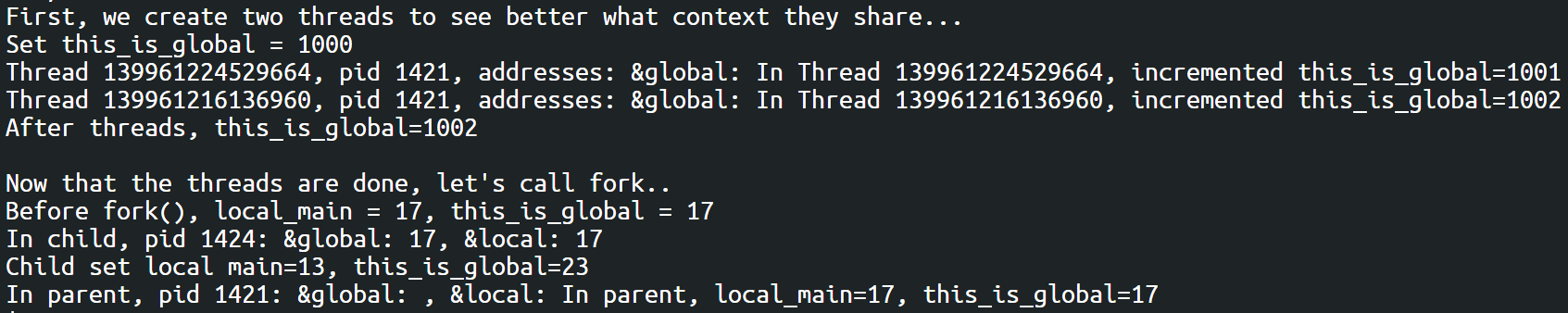
    printf("Thread %ld, pid %d, addresses: &global: %ls, &local: %ls\n", pthread\_self(), getpid(), &this\_is\_global, &local\_thread);

    this\_is\_global++;

    printf("In Thread %ld, incremented this\_is\_global=%d\n", pthread\_self(), this\_is\_global); pthread\_exit(0);

}

**Sample output**



This shows how a process has its own values which is separated from the parent while the parent shares its values with the threads.

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

### Lab5\_5.c

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

int tot\_items = 0 ;

struct kidrec {

int data ;

    pthread\_t id ;

} ;

#define NKIDS 50

void \*kidfunc(void \*p)

{

    int \*ip = (int \*)p ;

    int tmp, n ;

    tmp = tot\_items ;

    for (n = 50000; n--; )

        tot\_items = tmp + \*ip ;

}

int main(void)

{

    struct kidrec kids[NKIDS] ;

    int m ;

    for (m=0; m<NKIDS; ++m)

    {

        kids[m].data = m+1 ;

        pthread\_create (&kids[m].id, NULL, kidfunc, &kids[m].data) ;

    }

    for (m=0; m<NKIDS; ++m)

        pthread\_join (kids[m].id, NULL) ;

    printf ("End of Program. Grand Total = %d\n", tot\_items) ;

}

### 

**Sample output**

Run it several times until you see different output. How many times is the line  **tot\_items = tmp + \*ip ;** executed? What values does **\*ip** have during these executions?

# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Passing Arguments to Threads :**

The **pthread\_create( )** routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the **pthread\_create( )** routine.

All arguments must be passed by reference and cast to **(void \*)**.

Important: threads initially access their data structures in the parent thread's memory space. That data structure must not be corrupted/modified until the thread has finished accessing it.

The following example pass a simple integer to each thread.

**Example: pthread\_create( ) argument passing :**

## Lab5\_6.c

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define NUM\_THREADS 7

char \*messages[NUM\_THREADS];

void \*PrintHello(void \*threadid)

{

    int \*id\_ptr, taskid;

    sleep(1);

    id\_ptr = (int \*) threadid;

    taskid = \*id\_ptr;

    printf("\n %s from thread %d \n\n", messages[taskid], taskid);

    pthread\_exit(NULL);

}

int main( )

{

    pthread\_t   threads[NUM\_THREADS];

    int \*taskids[NUM\_THREADS];

    int rc, t;

    messages[0] = "English: Hello World!";

    messages[1] = "French: Bonjour, le monde!";

    messages[2] = "Spanish: Hola al mundo";

    messages[3] = "Klingon: Nuq neH!";

    messages[4] = "German: Guten Tag, Welt!";

    messages[5] = "Russian: Zdravstvytye, mir!";

    messages[6] = "Japan: Sekai e konnichiwa!";

    messages[7] = "Latin: Orbis, te saluto!";

    for(t=0;t<NUM\_THREADS;t++)

    {

        taskids[t] = (int \*) malloc(sizeof(int));

        \*taskids[t] = t;

        printf("Creating thread %d\n", t);

        rc = pthread\_create(&threads[t], NULL, PrintHello, (void \*) taskids[t]);         if (rc)

        {

            printf("ERROR; return code from pthread\_create() is %d\n", rc);             exit(-1);

        }

    }

    pthread\_exit(NULL);

}

**Sample output**

**Creating thread 0 Creating thread 1 Creating thread 2 Creating thread 3 Creating thread 4 Creating thread 5 Creating thread 6**

### English: Hello World! from thread 0 French: Bonjour, le monde! from thread 1 Spanish: Hola al mundo from thread 2 Klingon: Nuq neH! from thread 3 German: Guten Tag, Welt! from thread 4 Japan: Sekai e konnichiwa! from thread 6 Russian: Zdravstvytye, mir! from thread 5

**Assignments:**

**Problem#1:**

The following **Box #1** program demonstrates a simple program where the **main thread creates another thread** to print out the numbers from 1 to 20. The **main thread waits till the child thread finishes**.

# /\* Box #1: Simple Child Thread \*/

#include <pthread.h>

#include <stdio.h>

void ChildThread(void \*argument)

{

    int i;

    for(i = 1; i <= 20; ++i ) {

        printf(" Child Count - %d\n", i);

    }

    pthread\_exit(0);

}

int main(void)

{

    pthread\_t hThread;

    int ret;

    ret = pthread\_create(&hThread, NULL, (void \*)ChildThread, NULL); /\* Create Thread \*/

    if (ret < 0)   {

        printf("Thread Creation Failed\n");

        return 1;

    }

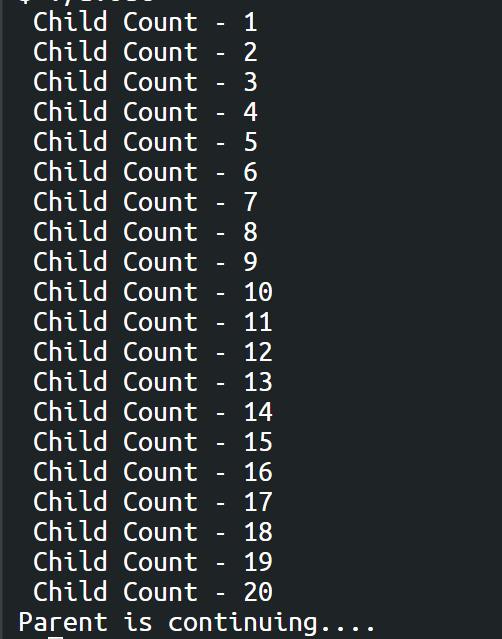
    pthread\_join(hThread, NULL);  /\* Parent waits for  \*/

    printf("Parent is continuing....\n");

    return 0;

}

**Compile and execute the Box #1 program and show the output and explain why is the output so?**



The program will create a thread and calculating to 20 then it will continue after its completion.

**Problem#2:**

In the **Box #2** modify the above **Box #1** program such that the main program passes the **count** as argument to the child thread function and the child thread function prints that many **count** print statements.

# /\* Box #2 : Passing Thread Arguments \*/

**#include <pthread.h>**

**#include <stdio.h>**

**void ChildThread (int argument)**

**{**

**int i;**

**.......................**

**pthread\_exit(0);**

**}**

**int main(void)**

**{**

**pthread\_t hThread;**

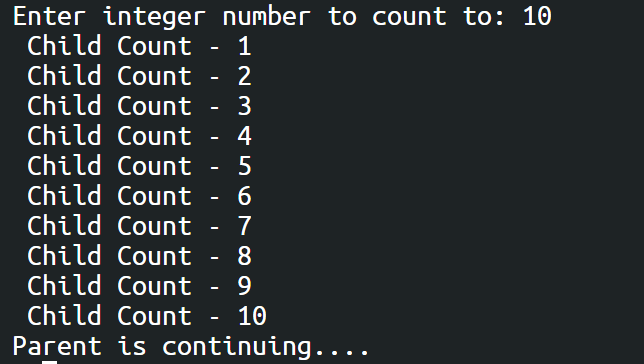
**pthread\_create (.............................);**

**pthread\_join (hThread, NULL);**

**printf ("Parent is continuing....\n"); return 0;**

**}**

**Compile and Execute the Box #2 program and show the output and explain why is the output so?**



I added last argument as “(void \*) x” in pthread\_create, x is an integer that is taken from the user by the console.

**Problem#3:**

**Write a program Box #3 by removing pthread\_exit function from child thread function and check the output? Is it the same as output of Box #2? If so Why? Explain?**

# /\* Box #3: Implicit Thread Exit \*/

**#include <pthread.h>**

**#include <stdio.h>**

**void ChildThread (int argument)**

**{**

**int i;**

**...............................**

# /\* No pthread\_exit function \*/

**}**

**int main(void)**

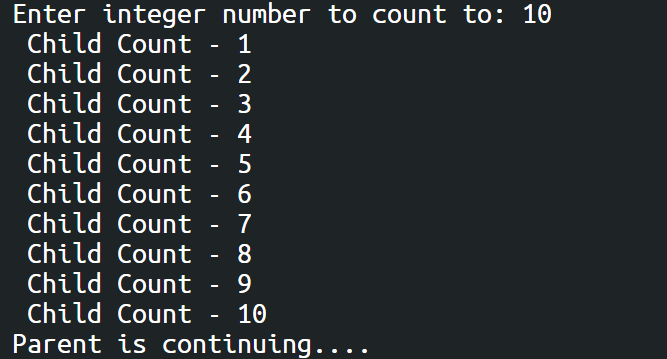
**{**

**pthread\_t hThread;**

**pthread\_create (...........................................);**

**pthread\_join (hThread, NULL);**

**printf ("Parent is continuing....\n"); return 0; }**

.

Yes, it is the same.

Assignment Exercise:

Write a program to find the vector sum C = A + B where A, B and C are one dimensional arrays of type int and of size N (assume max value for N is 100)

The main thread should initialize A and B with random integers between 0 and 10 (this is optional; you may assign these arrays some values by yourself), then creates N threads such that the ith thread calculates C[i] = A[i] + B[i]

Finally, the main thread prints the three arrays; A, B, and C

Hint: Use the function random() to get random integers, apply mod operator (%) to make the generated random number within a range. Use srandom() function to randomize the seed, so that each execution of your program generates a distinct set of random integers.