بسم الله الرحمن الرحيم

EE 463 : Operating Systems
Faculty Of Engineering
King Abdulaziz University
Department Of Electrical and Computer Engineering

LAB #6

Concurrent Process Creation and Execution

Multi-Threading

Objective:

Some application programs may have several possible execution sequences, which may be independent, or may depend on each other. Such programs can be made more responsive if the different sequences of execution are written as separate units that can compete for the CPU together with other program units in the system. Such units can be made as part of the same process, (i.e. have the same PCB and share same address space), called Threads; or can have their own Processes.

Modern operating systems support multi-threading and multi-processing in user programs, through numerous data structures and functions provided in high-level programming language libraries.

In this lab you will learn how a program can create multiple threads of execution within a single process. We will examine the following Thread Management Functions:

- Creating Threads,
- Terminating Thread Execution,
- Passing Arguments To Threads,
- Thread Identifiers,
- Joining Threads, and
- Detaching / Undetaching Threads.

You can skip the discussions and go directly to the numbered instructions written in RED

What is a Thread?

A *Thread* is a semi-process, which has its own ***stack***, and executes a given piece of code. Unlike a real process, a thread normally ***shares*** *its memory with other threads* (whereas 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 concurrently (i.e. using time slices, or if the system has several processors, then they execute 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 and UNIX-like 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 programs.

Why Pthreads?

The primary motivation for using Pthreads is to realize potential program *performance gains*:

- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
    - **Overlapping CPU work with I/O**: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU intensive work.
    - **Priority/real-time scheduling:** tasks, which are more important, can be scheduled to supersede or interrupt lower priority tasks.
    - **Asynchronous event handling:** tasks, which service events of indeterminate frequency and duration, can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.
- Multi-threaded applications will work on a uniprocessor system; yet naturally take advantage of a multiprocessor system, without recompiling.
- In a multiprocessor environment, the most important reason for using Pthreads is to take advantage of potential parallelism. This will be the focus of the remainder of this session.

The Pthreads API :

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

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

- **Mutexes:** Functions of this class 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:** This 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.

The following table lists the predefined identifier types in the Pthreads API. All identifiers in the threads library begin with **pthread_**

Identifier Type	Object Description
pthread_t	Thread handles
pthread_attr_t	Thread creation attributes
pthread_cond_t	Condition variable Synchronization primitives
pthread_condattr_t	Condition variable creation attributes
pthread_mutex_t	Mutual exclusion synchronization primitives
pthread_mutexattr_t	Mutual exclusion creation attributes
pthread_rwlock_t	Read/Write synchronization primitives
pthread_rwlockattr_t	Read/Write lock attributes
pthread_key_t	Thread-specific data access keys
pthread_once_t	Once time initialization control variables
struct sched_param	Scheduling parameters (priority and policy)

Thread Management Functions:

The function **pthread_create()** is used to create a new thread, and a thread can terminate itself by using the function **pthread_exit()**. A thread can wait for termination of another thread by using the function **pthread_join()**. The syntax of these three functions is shown below:

```
Function:   int pthread_create(pthread_t *threadhandle,  /* Thread handle returned by reference */
                               pthread_attr_t *attribute,  /* Attribute for starting thread or NULL */
                               void *(*function)(void *),  /* Main Function which thread executes */
                                          void *arg); /* An extra argument passed as a pointer */
```

Requests the Pthread library to create a new thread. The return value is **0** on success and negative on failure. The **pthread_t** is an abstract data type for **threadhandle** that is used to reference the newly created thread.

```
Function:   void pthread_exit(void *retval);                    /* return value passed as a pointer */
```

This function is used by a thread to terminate itself. 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,     /* Passes thread handle of target thread */
                             void **returnvalue);    /* Return value is returned by reference */
```

This function suspends execution of the calling thread until the *target thread* terminates, unless the *target thread* has already terminated. Returns **0** on success, and negative on failure. The return value pointer references the value passed to **pthread_exit()** by the terminating thread. If you do not care about the return value, you can pass **NULL** for the second argument.

Thread Initialization:

To enable the use of Pthreads in your program, you have to do the following three steps:

1. Include the pthread.h library by adding the following line to your program:

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

2. Declare a variable of type **`pthread_t`** to store the thread handle(s):

```
pthread_t thread_handle_identifiers;
```

3. When you compile the program, add **`-lpthread`** to the linker flags, as follows:

```
$ cc your_threaded_program.c -o threaded_program -lpthread
```

Initially, threads are created from within a process by calling the **`pthread_create()`** function. Once created, threads are peers (i.e., no parent-child relationship among them), and they may create other threads. Note that an *initial thread* exists in the program by default and it is the thread which runs function **`main()`**.

Thread Termination:

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

- The thread returns from its starting routine – this is the **`main()`** function 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 receives a *signal* that terminates it. The entire process is terminated due to a call to either the **`exec()`** or **`exit()`** functions.
- The thread makes a call to the **`pthread_exit()`** function as specified before.
- The thread is canceled by another thread via the **`pthread_cancel()`** function as follows:

Function: `int pthread_cancel(pthread_t threadhandle); /* Passes thread handle of target thread */`

This function sends a cancellation request to the thread specified by the **`threadhandle`** 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.

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 the 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 threads previously created.

```
Function:   int pthread_attr_init(pthread_attr_t *attr);      /* Passes thread attribute object */
```

This function initializes the thread attribute object **attr** and fills it with default values for thread attributes. It returns **0** on success, and nonzero otherwise.

Each attribute **attrname** can be individually set using the function **pthread_attr_setattrname** and retrieved using the function **pthread_attr_getattrname**.

```
Function:   int pthread_attr_setattr(pthread_attr_t *obj,   /* Passes an attribute object pointer */
                                     int value);  /* Passes the new value of the object */
```

This function sets 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, this function returns **0**.

```
Function:   int pthread_attr_getattr(const pthread_attr_t *obj, /* Pass attribute object pointer */
                                     int *value);/* Return pointer to objct value */
```

This function stores the current setting of **attr** in **obj** into the variable pointed to by **value**. This function always returns **0**.

```
Function:   int pthread_attr_destroy(pthread_attr_t *attr);     /* Pass attribute object pointer */
```

This function destroys the attribute object pointed to by **attr** releasing any resources associated with it. It returns zero after completing successfully. Any other returned value indicates that an error occurred. After the execution of this function, **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.

The following thread attributes are supported:

Thread Attribute Name	Possible Values (default is colored)	Description
detachstate	**PTHREAD_CREATE_JOINABLE**	The thread is created in the joinable state.
	PTHREAD_CREATE_DETACHED	The thread is created in the detached state.

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. If you do not want a thread to be joined, create it as a detached thread.

In the detached state: The thread resources are immediately freed when it terminates and can be reused, 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 state by using **pthread_detach()**. If you do not want the calling thread to wait for the thread to terminate then call the function, **pthread_attr_setdetachstate()**.

Thread Attribute Name	Possible Values (default is colored)	Description

`schedpolicy`	`SCHED_OTHER`
	`SCHED_RR`
	`SCHED_FIFO`
		Regular, non-realtime scheduling.
		Realtime, round-robin scheduling.
		Realtime, first-in first-out scheduling.

In the regular scheduling policy: The new thread uses fixed priority scheduling, where, threads run until preempted by a higher-priority thread or until they block or yield.

The realtime scheduling policies, **`SCHED_RR`** and **`SCHED_FIFO`** are available only to processes with **superuser** privileges. The function, **`pthread_attr_setschedparam()`** will fail and return **`ENOTSUP`** error if you try to set a realtime policy when you are *unprivileged*. The scheduling policy of a thread can be changed after creation by, **`pthread_setschedparam()`**.

Thread Attribute Name	Possible Values (default is colored)	Description
`schedparam`	`0`	Sets scheduling priority (a number).

Changes the scheduling parameter (the scheduling priority) for the thread. 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 by, **`pthread_setschedparam()`**.

Thread Attribute Name	Possible Values (default is colored)	Description
`scope`	`PTHREAD_SCOPE_PROCESS` `PTHREAD_SCOPE_SYSTEM`	Process contention scope. System contention scope.

Sets the scheduling contention scope attribute for the thread to either process, where it is unbound - not permanently attached to LWP, or system.

Thread Attribute Name	Possible Values (default is colored)	Description
`inheritsched`	`PTHREAD_INHERIT_SCHED` `PTHREAD_EXPLICIT_SCHED`	Inherit the scheduling attributes. Set the scheduling attributes explicitly.

In the inherit option: The new thread inherits creating thread's scheduling attributes. The scheduling attributes in this **`attr`** argument shall be ignored.

In the explicit option: The new thread scheduling attributes shall be set to the corresponding values from this **`attr`** argument.

The following thread scheduling attributes defined by IEEE Std 1003.1-2001 are affected by the **`inheritsched`** attribute:

- Scheduling policy (**`schedpolicy`**),
- Scheduling parameters (**`schedparam`**), and
- Scheduling contention scope (**`scope`**).

Thread Identifiers:

When threads are created, the new thread does not get its thread ID, and Pthreads provides two functions that deal with thread IDs:

Function:
```
Pthread_t pthread_self(void);                    /* Takes no arguments */
```

This function returns the unique thread ID of the calling thread. The returned data object is **opaque** and cannot be easily inspected. This is a capability similar to the **`getpid()`** function for processes.

Function:
```
int pthread_equal(pthread_t t1, pthread_t t2);    /* Pass two thread IDs to compare */
```

This function 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 against each other, or to compare a single thread ID against another value.

Dynamic Threaded Package Initialization:

When a multi-threaded program is run, it is usually desirable to execute some initialization code that will run only once in the lifetime of the program. The initialization code can be setup as a function that we can call, the **`init_function`**, which can then be called by a thread to execute. The **`pthread_once()`** function ensures that the **`init_function`** is executed only once.

Function:
```
int pthread_once(pthread_once_t *once_control,  /* once_control = PTHREAD_ONCE_INIT */
                 void (*init_function)(void)); /* Pass an initialization routine */
```

This function executes the **`init_function`** exactly once in a process. The first call to this function by any thread in the process executes the given **`init_function`**, without parameters. Any subsequent call will have no effect. The **`init_function`** routine is typically an initialization routine. The **`once_control`** parameter is a synchronization control structure that requires initialization prior to calling **`pthread_once()`**. For example:

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

On return from **`pthread_once()`**, **`init_function`** shall have completed. The **`once_control`** parameter shall determine whether the associated initialization routine has been called. Upon successful completion, **`pthread_once()`** shall return zero; other-wise, an error number shall be returned to indicate the error.

Passing Arguments to Threads:

The **`pthread_create()`** function permits the programmer to pass one argument to the thread start function. 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()`** function. 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.

Examples:

1. The following program demonstrates thread creation and termination:

```
/* 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);
}
```

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

```
Parent: My process# ---> 8123
Parent: My thread # ---> 1216846144
Child: Hello World! It's me, process# ---> 8123
Child: Hello World! It's me, thread # ---> 1216849040
Parent: No more child thread!
```

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

2. The following program demonstrates thread global data:

```
/* 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);
}
```

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

```
Parent: Global data = 5
Child: Global data was 10.
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?

5) Do the threads have separate copies of **`glob_data`**?

3. The following example demonstrates a multi-threaded program:

```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);
}
```

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

```
I am the parent thread
I am thread #4, My ID #3043994480
I am thread #5, My ID #3035601776
I am thread #6, My ID #3027209072
I am thread #3, My ID #3052387184
I am thread #2, My ID #3060779888
I am thread #7, My ID #3018816368
I am thread #8, My ID #3010423664
I am thread #9, My ID #3002030960
I am thread #1, My ID #3069172592
I am thread #0, My ID #3077565296
I am the parent thread again
```

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

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

```
/* 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);
}
```

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

```
First, we create two threads to see better what context they share...
Set this_is_global to: 1000
Thread: 3070053232, pid: 10487, addresses: local: 0XB6FD438C, global: 0X804A040
Thread: 3070053232, incremented this_is_global to: 1001
Thread: 3078445936, pid: 10487, addresses: local: 0XB77D538C, global: 0X804A040
Thread: 3078445936, 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: 10487, lobal address: 0XBFF5CB0C, global address: 0X804A040
Child : pid: 10490, local address: 0XBFF5CB0C, global address: 0X804A040
Child : pid: 10490, set local_main to: 13; this_is_global to: 23
Parent: pid: 10487, local_main = 17, this_is_global = 17
```

9) Did **`this_is_global`** change after the threads have finished? Why?
10) Are the local addresses the same in each thread? What about the global addresses?
11) Did **`local_main`** and **`this_is_global`** change after the child process has finished? Why?
12) Are the local addresses the same in each process? What about global addresses? What happened?

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

```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);
}
```

13) Run the above program several times and observe the outputs, until you get different results.
14) How many times the line **`tot_items = tot_items + *iptr;`** is executed?
15) What values does **`*iptr`** have during these executions?
16) What do you expect **`Grand Total`** to be?
17) Why you are getting different results?
18) Write your answers to all the above questions with evidence of your experiments and submit them in one .pdf file.