Parallel Programming with the PThread API

- **POSIX Thread tuturials on the WEB:**
  - Getting Started with POSIX Threads (online book): click here
  - PThread Primer (online book): click here

- **The POSIX Threads API**
  - The POSIX Thread API was the result of a ***standardization effort*** to make multi-threaded programs ***portable*** among different computer vendors

    Before the POSIX standard, each computer vendor would implement its own thread library and the resulting programs were not protable across different computer systems.
  - To use the POSIX threads (or "pthreads" for short), include the header file:

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

  - And use the compiler options:

```
CC -mt [other flags] C++-program (Solaris)

g++ -pthread [other flags] C++-program (Linux)
```

  - **Note:**
    - The **C++** language has change a bit in Linux
    - You must use:

```
#include <iostream>

using namespace std;
```

      instead of:

```
#include <iostream.h>
```

      to access the **cin** and **cout** objects.

Creating Threads

- **Creating a new thread of control**
  - **Syntax:**

```
int pthread_create(pthread_t *threadID,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg);
```

Input *Parameters:*

- `attr` = contains the **attributes (properties)** of the ***new* thread**
  - **Thread attributes** are **discussed *later***

- `start_routine` = **name** of the **function** that the ***new* thread** will **execute**
  - This **function** has **one parameter** of the **type** `(void *)`

  - The **return value type** of this **function** is `void *`

    I.e.:

```
void * start_routine( void * arg )
{
    ....
}
```

- `arg` = the **argument** that will be **passed** to the `start_routine` function when it **executes**

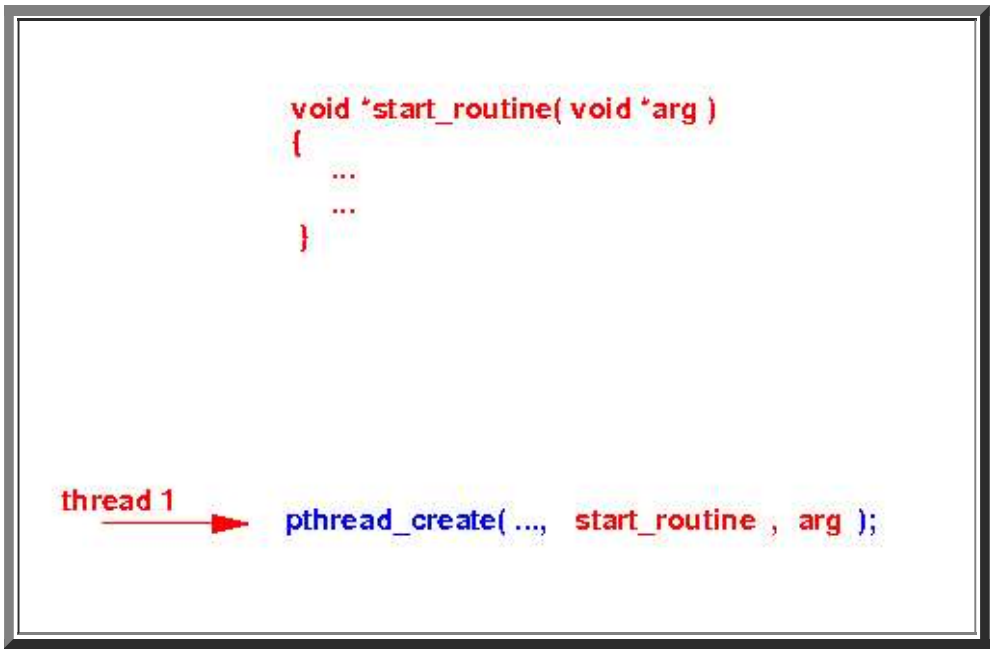
Output parameters

- `threadID` = the **identifier** of the ***new* thread**
  - We will use the `threadID` to **wait** for the **thread** to **finish**

Return value:

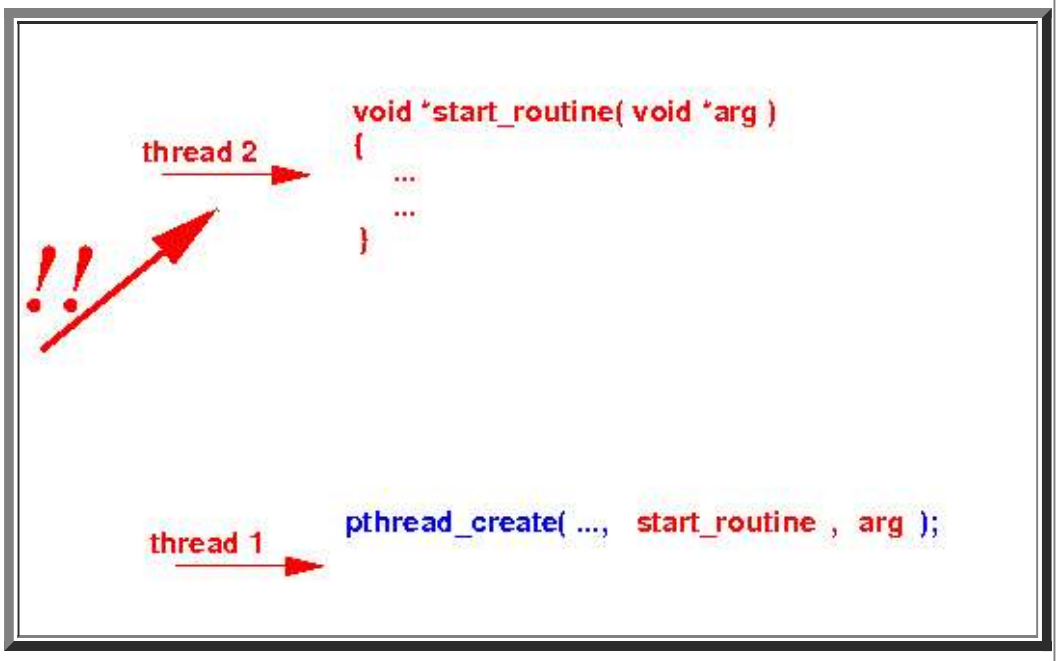
- Returns **0** if **thread** is **created successfully**
- **Otherwise**, returns an **error code**

- **Effect** of the `pthread_create(..., start_routine, ...)` call:
  - **Before** the `pthread_create( )` call:



`Thread 1` is **executing** the `pthread_create( )` call

  - **After** the `pthread_create( )` call:



> Thread 1 will **continue execution** the **statements after** the `pthread_create( )` call
> A ***new*** **thread** `Thread 2` will **start** its **execution** with the **function** `start_routine`

- **Setting attributes for new threads**
  - The attributes for a newly created thread is defined by the user using a special ***bit-variable*** of the type "pthread_attr_t".

    The following definition will define one such variable; and the name of our variable is "attr" (you can call the variable any name that you like):

    ```
    pthread_attr_t    attr;
    ```

  - Once such "pthread_attr_t" variable is defined, you must first ***reset all properties*** to their ***default values*** using the function "pthread_attr_init()":

    ```
    pthread_attr_init(&attr);
    ```

  - The following is a list of "thread attributes" and their default values

Attribute	Default	Meaning of default
contentionscope	PTHREAD_SCOPE_PROCESS	Thread will compete for resources within the process
detachstate	PTHREAD_CREATE_JOINABLE	Thread is ***joinable*** by other threads
stackaddr	NULL	Workspace (stack) used by thread is allocated (reserved) by the operating system
stacksize	NULL	Workspace size used by thread is determined by the operating system
priority	0	Priority of the thread (the lower the priority value, the higher the priority...)
policy	SCHED_OTHER	The scheduling policy is determined by the system
inheritsched	PTHREAD_EXPLICIT_SCHED	scheduling policy and parameters are not inherited but explicitly defined by the attribute object
guardsize	PAGESIZE	size of guard area for a thread's created stack (this area will help

		determine if the thread has exceeded the total amount of space that was allocated for the thread)

- **Changing the default values:**
  - ***There is no need to change MOST of the default values***

    (In fact, my examples always use the **default values**)
  - If you have the need to change some value, you must **invoke specific functions** to **set parameters** in the **attr** variable (after clearing the bits):

    - pthread_attr_setscope(): change contentionscope
    - pthread_attr_setdetachstate(): change detachstate
    - pthread_attr_setstackaddr(): pick your own stack location
    - pthread_attr_setstacksize(): pick your own stack size
    - .. etc

    - See: ***man pthread_attr_init*** for more functions.
    - See: ***man pthread_attr_setscope*** for more details on the pthread_attr_setscope function, etc.

  - **Example: Changing the default STACKSIZE**
    - The ***only*** default value that you would ever feel the need to change is: ***stacksize***

      The stack is used to store:

      - local variables
      - parameter variables
    - You will need to **increase the stack size** if the function run by a thread uses:

      1. **many local variables**
      2. is ***recursive*** (because the function will create **multiple copies** of parameter and local variables !!!)
    - How to set your own stacksize (workspace size) for your threads:

```
int pthread_attr_setstacksize(pthread_attr_t  *attr,
                              size_t stacksize);
```

Example:

```
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setstacksize(&attr, 10000000); // 10000000 bytes stack
```

- **Joinable and detached threads**
  - As many parallel applications often have interspersed sequential and parallel steps, where the exection of the sequential step must wait until all threads in the parallel step has finished execution; you often find that threads must sometimes ***WAIT*** on each other.
  - The crucial difference between ***joinable*** and ***detached*** threads is precisely this wiating property
  - ***Detached threads*** run independently from other threads and cannot wait for other threads.

    The waiting on other threads is accomplished by using the function ***pthread_join()*** (more later).
  - ***Joinable threads*** may call pthread_join() to wait on another thread.
  - The default value is ***joinable***, you do NOT want to change this (unless you are going to provide some kind of barrier synchronization routine yourself....)

- **Summary thread creation...**
  - The following code fragment summarizes how to create a thread:

```
pthread_t       TID;    /* Thread ID - used for signaling */
pthread_attr_t  attr;   /* Thread attribute values for creation */

int             param;


/* ------------------------------------------------
   Clear out thread attributes (use default values)
   ------------------------------------------------ */
if (pthread_attr_init(&attr) != 0)
{  perror("Problem: pthread_attr_init");
   exit(1);
}

/* ------------------------------------------------------
   Optionally, set thread attributes if needed (often not)
   ------------------------------------------------------ */
if ( pthread_attr_setstacksize(&attr, 10000000) != 0)
{  perror("Problem: pthread_attr_setstacksize");
   exit(1);
}

param = some value....;
```

```
  /* ------------------------------------------------------------
     Create thread (once created, thread starts running my_proc()

     Note: the C operator & means: address of ...
     ------------------------------------------------------------ */
  if (pthread_create(& TID, & attr, my_proc, & param) != 0)
  {  perror("Problem: pthread_create");
     exit(1);
  }
```

We must also provide a function for the thread to execute:

```
  void *my_proc((void *) x)        <---- new thread will begin
  {                                      execution with this function
    ....
  }
```

- ## Example: Hello World in threads

  - ### Program:

```
#include <pthread.h>

/* ================================
   Thread prints "Hello World"
   ================================ */
void *worker(void *arg)
{
  cout << "Hello World !" << endl;

  return(NULL);      /* Thread exits (dies) */
}

/* ============================================
   MAIN: create a trhead and wait for it to finish
   ============================================ */
int main(int argc, char *argv[])
{
   pthread_t  tid;

   /* ---------------------
      Create threads
      --------------------- */
   if ( pthread_create(&tid, NULL, worker, NULL) )
   {
      cout << "Cannot create thread" << endl;
      exit(1);
   }

   cout << "Main waits for thread to finish...." << endl ;

   pthread_join(tid, NULL);

   exit(0);
}
```

- **Example Program:** (Demo above code)

  Example

  - Prog file: click here

  How to run the program:

  > - **Right click** on link and **save** in a scratch directory
  > - To compile: `gcc -pthread HelloWorld.c`
  > - To run: `./a.out`

- **Passing a parameter to a thread**
  - The **function** executed by a **thread** can have **one parameter**

    Multiple parameters:

    > - **Multiple parameters** can be passed by **passing the *address*** of a **data structure** as the **parameter**

  - To **accomodate different types** of **parameter**, the **casting (type conversion)** operation is usually needed
  - ***Example: passing an integer parameter to a thread***

```
/* ------------------------------------------------------
   Worker thread will receive an INTEGER input parameter
   ------------------------------------------------------ */
void *worker(void *arg)
{
  int *p;      // Pointer (reference, address) to an integer variable
  int x;       // An integer variable

  p = (int *) arg;   // Casting "(void *)" type to "(int *)"
  x = *p


  cout << "Hi, my input parameter is " << x << endl;

  return(NULL);      /* Thread exits (dies) */
}
```

```
 /* ------------------------------------------------------
    Main creates thread and passes param to new thread
    ------------------------------------------------------ */
int main(int argc, char *argv[])
{
  pthread_t       tid;
  int             param;          // Integer

  param = 12345;
```

```
    pthread_create(&tid[i], &attr, worker, & param);
            // This should beinside an if to check error...

    pthread_join(tid, NULL);  // Wait for the thread to end...
}
```

- **Example Program:** (Pass parameter to thread) --- click here

  - Compile with: `g++ -pthread param.C`

- Try changing the type from **int** to **float**
  - Only in main
  - Only in worker
  - In both main and worker
  - You will see that they must agree !

- **Waiting for worker threads to finish....**
  - You have seen it in some examples before, I just want to spell out the effect of the function ***pthread_join()*** explicitly.
  - When a thread ***A*** calls the function

Thread *A* invokes:

```
pthread_join(B, &status);


B = ID of a thread
status = variable to receive the exit state of thread.
```

Then the following will happen:

- The calling thread (***A***) is ***suspended*** until the target thread (***B***) completes

  The thread ***B*** completes if the function (that thread ***B*** executes) returns or thread ***B*** executes the ***pthread_exit()*** function

- Note that thread ***B*** must be ***JOINABLE***

Example: waiting for **1 thread** (**Hello World**)

- **Program:**

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

```
/* ================================
   Thread prints "Hello World"
   ================================ */
void *worker(void *arg)
{
  cout << "Hello World !" << endl;

  return(NULL);      /* Thread exits (dies) */
}

/* ============================================
   MAIN: create a trhead and wait for it to finish
   ============================================ */
int main(int argc, char *argv[])
{
   pthread_t  tid;

   /* ---------------------
      Create threads
      --------------------- */
   if ( pthread_create(&tid, NULL, worker, NULL) )
   {
      cout << "Cannot create thread" << endl;
      exit(1);
   }

   cout << "Main waits for thread to finish...." << endl ;

   pthread_join(tid, NULL);

   exit(0);
}
```

- **Example Program:** (Demo above code)

  Example

  - Prog file: click here

  How to run the program:

  - **Right click** on link and **save** in a scratch directory
  - To compile: `gcc -pthread HelloWorld.c`
  - To run: `./a.out`

- **Example 2:** waiting a ***multiple*** **threads**

```
int main(int argc, char *argv[])
{
   int i, num_threads;
   thread_t tid[100];        /* Thread ID used for thread_join() */
   int   param[100];         /* Parameters for threads */

   num_threads = ... (number of threads to create);

   /* -------------------------
```

```
        Create threads
        ------------------------- */
    for (i = 0; i < num_threads; i = i + 1)
    {
        param[i] = ... ;   // Initialize parameter for thread i

        if ( pthread_create(&tid[i], &attr, worker, & param[i]) != 0 )
        {
            printf("Cannot create thread\n");
            exit(1);
        }
    }

    /* --------------------------------
        Wait for ALL threads to finish
        -------------------------------- */
    for (i = 0; i < num_threads; i = i + 1)
        pthread_join(tid[i], NULL);
}
```

Previous example: thread01.C

- **Example Program:** (Demo above code)

  - Prog file: click here

How to run the program:

- **Right click** on link and **save** in a scratch directory
- To compile: `g++ -pthread thread01.C`
- To run: `./a.out`