Multithreaded Programming

On Linux

Prayas Mohanty (Red Hat Certified Instructor)
Red Hat Certification ID: 100-005-594

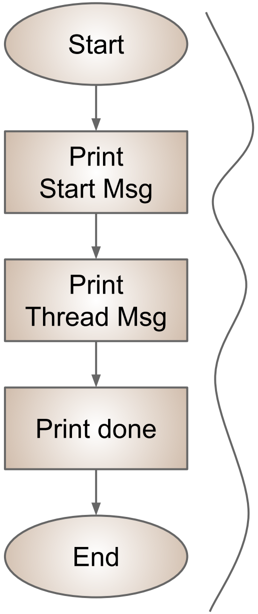
Objective

- What is Thread
    - What is Multithreaded Programming
- What is the Difference between thread & process
- What is thread attributes & how to use.
- How to use shared resources / data in thread
- How to Develop a multithreaded application using POSIX Library.
- How to handle Errors in Threads
- What is Thread Synchronization.
    - What is race condition.
    - What is critical section.
    - How to apply synchronization mechanisms (mutex, semaphore).

Prerequisite of Participants

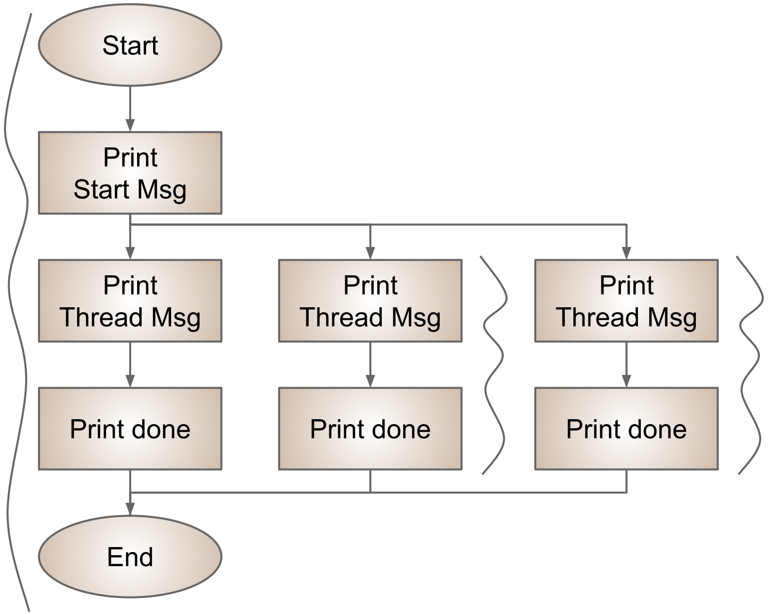
- Having Knowledge on Linux Platform
- Having familiarity with Linux Command line
- Basic Knowledge on vi Editor
- Proper Knowledge on C programming
- Basic Knowledge on Process Handling in C

What is a Thread



- A thread is a single sequential flow of control within a program.
    - It is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
    - It is an execution unit that has its own program counter, a stack and a set of registers that reside in a process.
- The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process.
    - Linux is quite capable of running multiple threads simultaneously inside a process. Indeed, all processes have at least one thread of execution inside a process.

What is multithreading



- Multithreading is a model of program execution that allows for multiple threads to be created within a process, executing independently but concurrently sharing process resources.
- When the original UNIX and POSIX library routines were designed, it was assumed that there would be only a single thread of execution in any process.
- Think of it as the application's version of multitasking.

Thread Example

```
include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //Header file for sleep().

void *myThreadFun(void *vargp) {
        printf("Hello World \n");
        sleep(1);
        printf("done\n");
        return NULL;
}

int main() {
        pthread t thread id1,thread id2,thread_id3;
        printf("Welcome Before Thread\n");
        pthread create(&thread id1, NULL, myThreadFun, NULL);
        pthread create(&thread id2, NULL, myThreadFun, NULL);
        pthread create(&thread id3, NULL, myThreadFun, NULL);
        pthread join(thread id1, NULL);
        pthread join(thread id2, NULL);
        pthread join(thread_id3, NULL);
        exit(0);
}
```

- Including the file pthread.h provides you with other definitions and prototypes that you will need in your code.
- Finally, you need to ensure that you include the appropriate thread header file and link with the appropriate threads library that implements the pthread function.
- During Compilation provide pthread library.

Process VS Threads

Process means any program is in execution.	Thread means segment of a process.
Process takes more time to terminate.	Thread takes less time to terminate.
It takes more time for creation.	It takes less time for creation.
It also takes more time for context switching.	It takes less time for context switching.
Process is less efficient in term of communication.	Thread is more efficient in term of communication.
Process is isolated.	Threads share memory.
Process use to be heavy weight	A Thread is lightweight as each thread in a process shares code, data and resources.

Thread Attributes

- Attributes are a way to specify behavior that is different from the default.
- When a thread is created with pthread_create an attribute object can be specified.
    - `pthread_create(&thread_id1, NULL, myThreadFun, NULL);`
- The defaults are usually sufficient.
- Attributes are specified only at thread creation time; they cannot be altered while the thread is being used.
- Attribute objects need to get initialize then used during thread creation and subsequently must be destroyed to free up Memory.

Attribute	Default Value	Result
scope	PTHREAD_SCOPE_PROCESS	New thread is unbound - not permanently attached to LWP.
detachstate	PTHREAD_CREATE_JOINABLE	Exit status and thread are preserved after the thread terminates.
stackaddr	NULL	Thread has system-allocated stack address.
stacksize	1 megabyte	New thread has system-defined stack size.
priority		New thread inherits parent thread priority.
inheritsched	PTHREAD_INHERIT_SCHED	Thread inherits parent's scheduling priority.
schedpolicy	SCHED_OTHER	New thread uses Solaris-defined fixed priority scheduling; threads run until preempted by a higher-priority thread or until they block or yield.

Attribute Example

```
void *myThreadFun(void *vargp) {
    printf("Hello World \n");
    sleep(3);
    printf("done\n");
    return NULL;
}

int main() {
    pthread t thread id1,thread_id2,thread_id3;
    pthread attr t attr;
    pthread attr init (&attr);
    pthread attr setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
    printf("Welcome Before Thread\n");
    pthread create(&thread id2, NULL, myThreadFun, NULL);
    pthread create(&thread id3, NULL, myThreadFun, NULL);
    pthread join(thread id2, NULL);
    pthread join(thread id3, NULL);
    pthread create(&thread id1, &attr, myThreadFun, NULL);
    pthread attr destroy(&tattr);
    pthread join(thread_id1, NULL);
    exit(0);
}
```

- Initialize variable (attr in our program) to hold the attribute object by pthred_attr_t data type.
- Use pthread_attr_init(&attr) to initialize object attributes to their default values.
    - The storage is allocated by the thread system during execution.
- Set of get the required attribute using macro to pthread_attr_setXXXX (where XXXX is the actual attribute in our case, it is detachstate).
    - Our program uses the macro PTHREAD_CREATE_DETACHED.
- The Attribute Object created above must get destroyed by pthread_attr_destroy function to free the memory.

Shared Resources

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //Header file for sleep().
#include <pthread.h>
#include <string.h>

char msg[] = "Global Hello Message";

void *myThreadFun(void *vargp) {
        printf("In Thread Message: %s\n", msg);
        strcpy(msg,"Thread Hello Message");
        return NULL;
}

int main() {
       pthread t thread id1;
       printf("Start of Main Message: %s\n",msg);
       strcpy(msg,"Thread Hello Message");
       pthread create(&thread id1, NULL, myThreadFun, NULL);
       pthread join(thread id1, NULL);
       printf("End of Main Message: %s\n",msg);
       exit(0);
}
```

- Process, group and session IDs
- Open file descriptors
- Signal dispositions
- Text/code segment
- Initialized data, uninitialized data, and heap segments

Resources not shared

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //Header file for sleep().
#include <pthread.h>
#include <string.h>

char msg[] = "Global Hello Message";

void *myThreadFun(void *vargp) {
        printf("Start of Thread Message: %s\n", msg);
        strcpy(msg,"Thread Hello Message");
        printf("End of Thread Message: %s\n", msg);
        return NULL;
}

int main() {
       pthread t thread id1;
       char msg[] = "Main Hello Message";
       printf("Start of Main Message: %s\n",msg);
       pthread create(&thread id1, NULL, myThreadFun, NULL);
       pthread join(thread id1, NULL);
       printf("End of Main Message: %s\n",msg);
       exit(0);
}
```

- Threads shouldn't "share" variables on the stack.
- Using Function/Block Scope Appropriately
  - Single Threaded Programs: Function/block scope means the same identifier in different functions/blocks refers to different entities
  - Multi-Threaded Programs: Distinct stacks means the same identifier in the same function/block in different threads refers to different entities
- Using Global Segments Appropriately
  - Single Threaded Programs: File scope means one identifier can be used in different functions to refer to the same entity
  - Multi-Threaded Programs: File scope means one identifier can be used in different threads to refer to the same entity

Passing Data to Threads

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

void *myThreadFun(void *tmsg) {
        printf("Inside Thread Message: %s \n", (char *) tmsg);
        printf("done\n");
        return NULL;
}

int main() {
        char msg[] = "Message from Main";
        pthread_t thread_id1,thread_id2,thread_id3;
        printf("Welcome Before Thread\n");
        pthread_create(&thread_id1, NULL, myThreadFun, (void *) msg);
        pthread_join(thread_id1, NULL);
        printf("Bye After Thread\n");
        exit(0);
}
```

- The pthread_create accept the arguments for the thread_function as the 4th parameter as follows
    - int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), **void *arg**);
- As it get collected to the thread as a void pointer, it need to get type cast before usage inside the thread function.

Return Data from Threads

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

void *myThreadFun(void *tmsg) {
        printf("Inside Thread Message: %s \n", (char *) tmsg);
        printf("done\n");
        pthread_exit("Thanks from Thread");
}

int main() {
        char msg[] = "Message from Main";
        void *thread_result;
        pthread_t thread_id1,thread_id2,thread_id3;
        printf("Welcome Before Thread\n");
        pthread_create(&thread_id1, NULL, myThreadFun, (void *) msg);
        pthread_join(thread_id1, &thread_result);
        printf("Thread return: %s\n", (char *) thread_result);
        exit(0);
}
```

- Use pthread_exit function to terminate calling thread
    - Usage: void pthread_exit(void *retval);
- It returns a value via retval that (if the thread is joinable).
- This function does not return to the caller.

Cancelling a Threads

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

void *myThreadFun(void *vargp) {
        printf("Hello World \n");
        sleep(3);
        printf("done\n");
        return NULL;
}

int main() {
        pthread_t thread_id1,thread_id2,thread_id3;
        printf("Welcome Before Thread\n");
        pthread_create(&thread_id1, NULL, myThreadFun, NULL);
        pthread_cancel(thread_id1);
        pthread_create(&thread_id2, NULL, myThreadFun, NULL);
        pthread_create(&thread_id3, NULL, myThreadFun, NULL);
        pthread_join(thread_id1, NULL);
        pthread_join(thread_id2, NULL);
        pthread_join(thread_id3, NULL);
        exit(0);
}
```

- Use pthread_cancel to send a cancellation request to a thread.
    - Usage: int pthread_cancel(pthread_t thread);
- Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability state and type.
- On success, pthread_cancel() returns 0 and on error, it returns a nonzero error number.

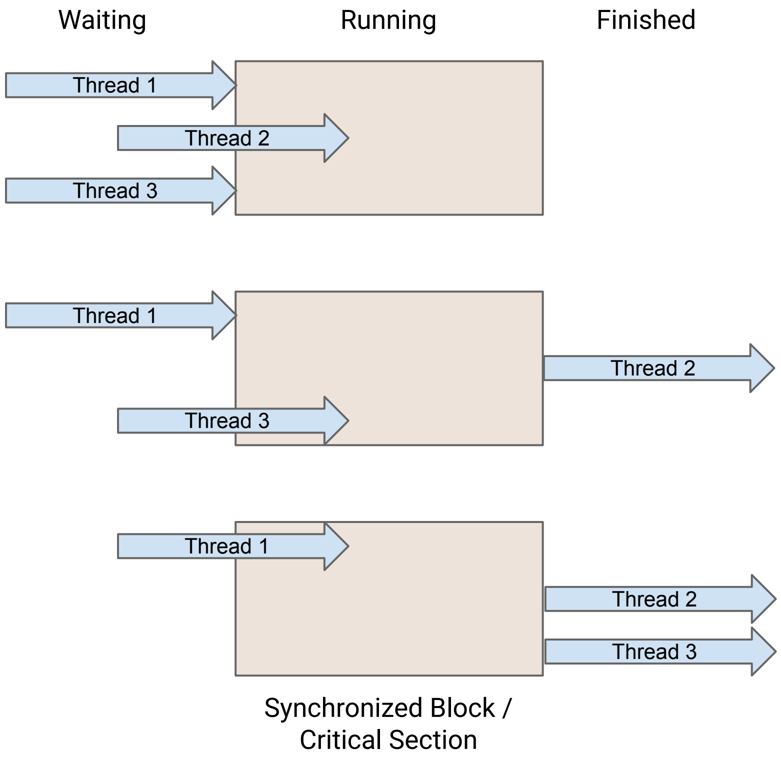
Handling Error in Threads

```
void *myThreadFun(void *vargp) {
        printf("Hello World \n");
        return NULL;
}

int main() {
        pthread_t thread_id1,thread_id2,thread_id3;
        pthread_attr_t attr;
        pthread_attr_init (&attr);
        pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
        if (pthread_create(&thread_id1, &attr, myThreadFun, NULL)) {
                perror("Thread creation failed");
                exit(EXIT_FAILURE);
        }
        pthread_create(&thread_id2, NULL, myThreadFun, NULL)  ;
        pthread_join(thread_id2, NULL);
        if (pthread_join(thread_id1, NULL)){
                perror("Thread Join failed");
                printf("Error No: %d\n",errno);
                printf("Error Message: 
%s\n",strerror(errno));
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

- On success, pthread_create() and pthread_join() returns 0; on error, it returns an error number.
- The errno variable host this value
  - Must include errno.h
- strerror function returns the proper string for the corresponding errno.
  - Must include string.h
- perror function prints custom message with error string(strerror).

What is Thread Synchronization



- Thread Synchronization is a mechanism which ensures that two or more concurrent process or threads do not execute some particular section of program especially critical section.
- In this technique one thread executes the critical section of a program and other thread wait until the first thread finishes execution.
- Access to critical section is controlled by a synchronization techniques.
    - If proper synchronization techniques are not applied, it may cause a **race condition** where the values of variables may be unpredictable.

Race condition

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int balance = 0;
void *mypThreadFun(void *targ) {
        int b;
        printf("Hello Balance \n");
        b = balance;
        b += 10;
        balance = b;
        printf("done\n");
}
int main() {
        int i;
        pthread t thread id[200];
        printf("Balance Before Thread: %d\n", balance);
        for (i=0; i<200; i++) {
           pthread_create(&thread_id[i], NULL, myThreadFun, NULL);
        }
        for (i=0; i<200; i++) {
                pthread_join(thread_id[i], NULL);
        }
        printf("Balance After Thread: %d\n", balance);
        exit(0);
}
```

- A race condition occurs when two threads access a shared variable at the same time.
    - The first thread reads the variable, and the second thread reads the same value from the variable.
    - Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable.
    - The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.
- Race conditions can be avoided by proper thread synchronization in **critical sections**.

Critical Sections

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int balance = 0;
void *mypThreadFun(void *targ) {
        int b;
        printf("Hello Balance \n");
        b = balance;
        b += 10;
        balance = b;
        printf("done\n");
}
int main() {
        int i;
        pthread_t thread_id[200];
        printf("Balance Before Thread: %d\n", balance);
        for (i=0; i<200; i++) {
           pthread_create(&thread_id[i], NULL, myThreadFun, NULL);
        }
        for (i=0; i<200; i++) {
                pthread_join(thread_id[i], NULL);
        }
        printf("Balance After Thread: %d\n", balance);
        exit(0);
}
```

- In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior.
    - So parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access.
- This protected section is of the code is referred as synchronized block or critical section.
- The critical section is a code segment where the shared variables can be accessed.

Synchronization Mechanisms

- Using Mutex:
  - A mutex provides mutual exclusion.
  - Strictly speaking, a mutex is a locking mechanism used to synchronize access to a resource.
  - Only one task (thread or process) can acquire the mutex.
  - It means there is ownership associated with a mutex, and only the owner can release the lock (mutex).
- At any point of time, only one thread can work with the critical section.

- Using Semaphore:
  - A semaphore provides generalized Synchronization.
  - In lieu of a single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources).
  - A semaphore can be associated with these four buffers.
  - The consumer and producer can work on different buffers at the same time.
  - Semaphore is signaling mechanism (“I am done, you can carry on” kind of signal).
- When a specific number (1,2,3,...,N) of threads want to work mutually over a section then Semaphore is the Solution.

Synchronization Using Mutex in Thread

```
pthread mutex t lock;
int balance = 0;
void mypThreadFun(void *targ) {
        int b;
        printf("Hello Balance \n");
        pthread mutex_lock(&lock);
        b = balance;
        b += 10;
        balance = b;
        pthread mutex unlock(&lock);
        printf("done\n");
}
int main() {
        int i;
        void *myThreadFun = &mypThreadFun;
        pthread t thread id[200];
        pthread mutex init(&lock,NULL);
        printf("Balance Before Thread: %d\n", balance);
        for (i=0; i<200; i++) {
            pthread_create(&thread_id[i], NULL, myThreadFun, NULL); }
        for (i=0; i<200; i++) { pthread_join(thread_id[i],NULL);}
        pthread mutex destroy(&lock);
        printf("Balance After Thread: %d\n", balance);
        exit(0);
}
```

- initialize a mutex Variable
    - pthread_mutex_t lock;
- initialize a mutex
    - pthread_mutex_init(&lock,NULL);
- Lock Mutex before Critical Section
    - pthread_mutex_lock(&lock);
- Unlock Mutex after Critical Section
    - pthread_mutex_unlock(&lock);
- Destroy Mutex to release memory
    - pthread_mutex_destroy(&lock);

Thread and Mutex specific coding guidelines

- Handle errors after every thread call.
- Parent should wait for all joinable child threads to exit and then exit.
- Do not pass stack variable as thread parameter rather allocate and pass a pointer to heap block.
- Do not return variable in stack, rather use static variable or return a pointer to heap block. Parent thread to free the allocated memory after use.
- Use pthread_exit() to return from thread.
- Do not rely on thread output sequence
- Hold lock for very short duration.
- Release locks after use.
- Do not attempt lock on an already acquired lock.

Summary

- What is Thread
  - What is Multithreaded Programming
- What is the Difference between thread & process
- What is thread attributes & how to use.
- How to use shared resources / data in thread
- How to Develop a multithreaded application using POSIX Library.
- How to handle Errors in Threads
- What is Thread Synchronization.
  - What is race condition.
  - What is critical section.
  - How to apply synchronization mechanisms (mutex, semaphore).

Any Questions ?

Thank you!