Threading in C

Zeyneb KURT

Outline

- What is a thread?
- Why do we need threads?
- Difference between threads and processes
- Problems with Threads
- Identifying a thread
- Creating a thread
- Terminating a thread
- Examples

Threading in C

- Threads/ Processes are the mechanism by which you can run multiple code segments at a time,
- A thread of execution is the smallest sequence of program instructions that can be managed independently by a scheduler
- A process can have multiple threads of execution.
- Threads appear to run concurrently; the kernel schedules them asynchronously, interrupting each thread from time to time to give others chance to execute.
- This asynchronous execution brings in the capability of each thread handling a particular work or service independently.
- Multiple threads running in a process handle their services which overall constitutes the complete capability of the process.

Use of Threads - Examples

- Graphical User Interfaces (GUIs)
  - The GUI is usually put on a separate thread from the "app engine"
  - GUI remains responsive even if app blocks for processing
- Web Browser Tabs
  - Each tab is managed by a separate thread for rendering
  - Web pages render "simultaneously" (e.g. while one page is printed out, another page can be downloaded concurrently)
  - Note: Google Chrome actually uses a separate process per tab

Why Threads are Required?

- Why do we need multiple threads in a process? Why can't a process with only one (default) main thread be used in every situation.
- To answer this lets consider an example:
  - Suppose there is a process, that receiving real time inputs and corresponding to each input it has to produce a certain output.
  - If the process does not involve multiple threads, then the whole processing in the process becomes synchronous.
  - This means that the process takes an input, processes it, and produces an output.

Why Threads are Required -2

- The limitation in the above design is that the process cannot accept an input until its done processing the earlier one.
- In case processing an input takes longer than expected, then accepting further inputs goes on hold.
- We could solve the above example with a socket server process that can accept input connection, process them and provide the socket client with output.
- While processing any input, if the server process takes more than expected time and in the meantime another input (connection request) comes to the socket server then the server process would not be able to accept the new input connection as its already stuck in processing the old input connection.
- This may lead to a connection time out at the socket client which is not at all desired.
- This shows that synchronous model of execution cannot be applied everywhere hence the asynchronous model of execution would be required, which is implemented by using threads.

Difference between threads and processes

- Processes do not share their memory space, while threads executing under same process share the memory space.
- Processes have independent open file descriptors, while threads have shared open file descriptors
- Processes execute independent of each other and the synchronization between processes is taken care by kernel only; on the other hand, thread synchronization has to be taken care by the process under which the threads are executing
- Context switching between threads is fast as compared to context switching between processes
- The interaction between 2 processes is achieved only through the standard inter-process communication, while threads executing under the same process can communicate easily as they share most of the resources like memory, text segment etc

Problems with Threads - I

- Many operating system does not implement threads as processes rather they see threads as part of parent process.
- What would happen if a thread execs a new binary (exe)?
- This scenario may have dangerous consequences e.g. the whole parent process could get replaced with the address space of the newly exec'd binary.
- So, an exec from any of the thread would stop all the threads in the parent process. This is not at all desired.
- This problem is a design issue and design for applications should be done in a way that least problems of this kind arise.
- Debugging with threads is difficult.
- Too many threads may reduce the performance.

Problems with Threads -2

- Another problem that may arise is the concurrency:
  - Since threads share all the segments (except the stack) and can be preempted at any stage by the scheduler before any global variable or data structure that can be left in inconsistent state by preemption of one thread could cause severe problems when the next high priority thread executes the same function & uses the same variables or data structures.
  - For the above problem: using locking mechanisms programmer can lock a chunk of code inside a function so, when a context switch* happens, next thread is not able to execute the same code until the locked code block inside the function is unlocked by the previous thread.
- *context switch: is the process of storing the state of a thread, so, it can be restored and execution resumed from the same point later. This allows multiple threads to share a single CPU.

Identifying a thread

- Each thread is identified by an ID, which is known as Thread ID
- Thread ID is quite different from Process ID.
- A Thread ID is unique in the context of current process, while a Process ID is unique across the system.
- A process ID is an integer value but the thread ID is not necessarily an integer value. It could be a structure and represented by type pthread_t.
- A process ID can be printed very easily while a thread ID is not easy to print.
- The header file, which needs to be included to access thread functions and pthread_t type, is: #include<pthread.h>

Creating a Thread (pthread_create)

- pthread_create function in pthread.h file, is used to create a thread.
- The syntax and parameters details are given as follows:

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

- pthread_t *thread: It is the pointer to a pthread_t variable which is used to store thread id of new created thread.
- const pthread_attr_t *attr: It is the pointer to a thread attribute object which is used to set thread attributes, NULL can be used to create a thread with default arguments.
- void *(*start_routine) (void *): It is the pointer to a thread function; this function contains the code segment which is executed by the thread.
- void *arg: It is the thread functions argument of the type void*, you can pass what is necessary for the function using this parameter.
- int (return type): If thread created successfully, return value will be 0 otherwise pthread_create will return an error number of type int.

Example

```
#include <stdio.h>
#include <pthread.h>
/*thread (worker) function definition*/
void* threadFunction(void* args){
    while(1)
        printf("I am threadFunction.\n");
}
int main(){    /*creating a thread id in the main function (main thread) */
   pthread_t id;
   int ret;
   /*creating thread*/
   ret=pthread_create(&id, NULL, &threadFunction, NULL);
   if(ret==0)
        printf("Thread is created successfully.\n");
   else{
        printf("Thread is not created.\n");
        return 0; /*return from main*/
    }
    while(1)
        printf("I am main function.\n");
    return 0;
}
```

* https://www.thegeekstuff.com/2012/03/linux-threads-intro/

How to compile & execute?

- To compile:

```
$ gcc <file-name.c> -o <output-file-name> -lpthread
```

- To run:

```
$ ./<output-file-name>
```

Example

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

pthread_t tid[2];

void* doSomeThing(void *arg){
    unsigned long i = 0;
    pthread_t id = pthread_self();

    if(pthread_equal(id,tid[0]))
        printf("\n First thread
                processing\n");
    else
        printf("\n Second thread
                processing\n");
    for(i=0; i<(0xFFFFFFFF);i++);
    return NULL;
}
```

```
int main(void){
    int i = 0;
    int err;

    while(i < 2) {
        err = pthread_create(&(tid[i]),
             NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread: %s",
                        strerror(err));
        else
            printf("\n Thread created
                        successfully\n");
        i++;
    }
    sleep(5); // this is important!!
    return 0;
}
```

* https://www.thegeekstuff.com/2012/04/create-threads-in-linux/

➢int pthread_equal(pthread_t tid1, pthread_t tid2); takes two thread IDs and returns a non-0 value if both IDs are equal, else it returns 0.

➢pthread_t pthread_self(void); // It is used by a thread for printing its own thread ID.

- Without sleep() function:

```
$ ./threads
Thread created successfully
First thread processing
Thread created successfully
```

- With sleep() function:

```
$ ./threads
Thread created successfully
First thread processing
Thread created successfully
Second thread processing
```

- Without the sleep* function, we did not see the message of "Second thread processing":
- Because just before the second thread is about to be scheduled, the parent thread, from which the two threads were created, completed its execution.
- So that the default thread in which the main() function was running got completed and hence the process terminated as main() returned.
- * sleep: sleep for the specified number of sec. Defined in <unistd.h>

Terminating a thread

- (See the example code) In the code:
- We created two threads using pthread_create()
- The start function for both the threads is same: doSomeThing()
- The threads exit from the start function using the pthread_exit() function with a return value (this is called inside the doSomeThing() function).
- In the main function, after the threads are created, the pthread_join() is called to wait for each thread to complete.
- Once both the threads are complete, their return value is accessed by the second argument in the pthread_join() call.

```c
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
int ret1,ret2;
void* doSomeThing(void *arg) {
    unsigned long i = 0;
    pthread_t id = pthread_self();
    for(i=0; i<(0xFFFFFFFF);i++);
    if(pthread_equal(id,tid[0])) {
        printf("\n 1st thread processing done\n");
        ret1  = 100;
        pthread_exit(&ret1);
    }
    else   {
        printf("\n 2nd thread processing done\n");
        ret2  = 200;
        pthread_exit(&ret2);
    }
    return NULL;
}
```

```c
int main(void)
{
    int i = 0, err;
    int *ptr[2];

    while(i < 2) {
        err = pthread_create(&(tid[i]), NULL,
                &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]",
                strerror(err));
        else
            printf("\n Thread created successfully\n");
        i++;
    }
    pthread_join(tid[0], (void**)&(ptr[0]));
    pthread_join(tid[1], (void**)&(ptr[1]));
    printf("\n return value from first thread is
            %d\n", *ptr[0]);
    printf("\n return value from second thread is
            %d\n", *ptr[1]);
    return 0;
}
```

https://www.thegeekstuff.com/2012/04/terminate-c-thread/

Void Pointers in C

Void pointer

- A void pointer is a pointer that has no associated data type with it.
- It can hold address of any type and can be *casted* to any type.
- Some advantages of the void pointers:
  - Void pointers in C are used to implement generic functions (e.g. qsort() function)
  - Functions such as malloc(), calloc() return void* type. Hence, they can allocate memory for any data type (just because of the void *)
- Some attributes of the void pointers:
  - Standard C does not allow pointer arithmetic with the void*. However, GNU C considers the size of the void is 1 byte.
  - void * cannot be derefenced. The use in the left-side is illegal:

```
int a = 10;
void *ptr=&a;
printf("%d", *ptr); // ILLEGAL
```

```
int a = 10;
void *ptr=&a;
printf("%d", * (int*) ptr); // becomes LEGAL
                               with type casting
```