Threads: Basic Concepts

Prof. Naga Kandasamy
ECE Department, Drexel University

This material has been derived from: Michael Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, No Starch Press, San Francisco, 2010.

Threads are a mechanism that permits an application to perform multiple tasks concurrently. A single UNIX process can contain multiple threads, as shown in Fig. 1, in which all these threads are independently executing the same program and all share the same global memory including the initialized and uninitialized data, and heap segments. So, a traditional UNIX process is simply a special case of a multi-threaded process — it is a process that contains just one thread.

The threads in a process can execute concurrently. Therefore, on a multi-processor system, multiple threads can execute in parallel. If one thread is blocked on I/O, other threads are still eligible to execute. Threads offer the following advantage than creating processes to achieve the same level of concurrency.

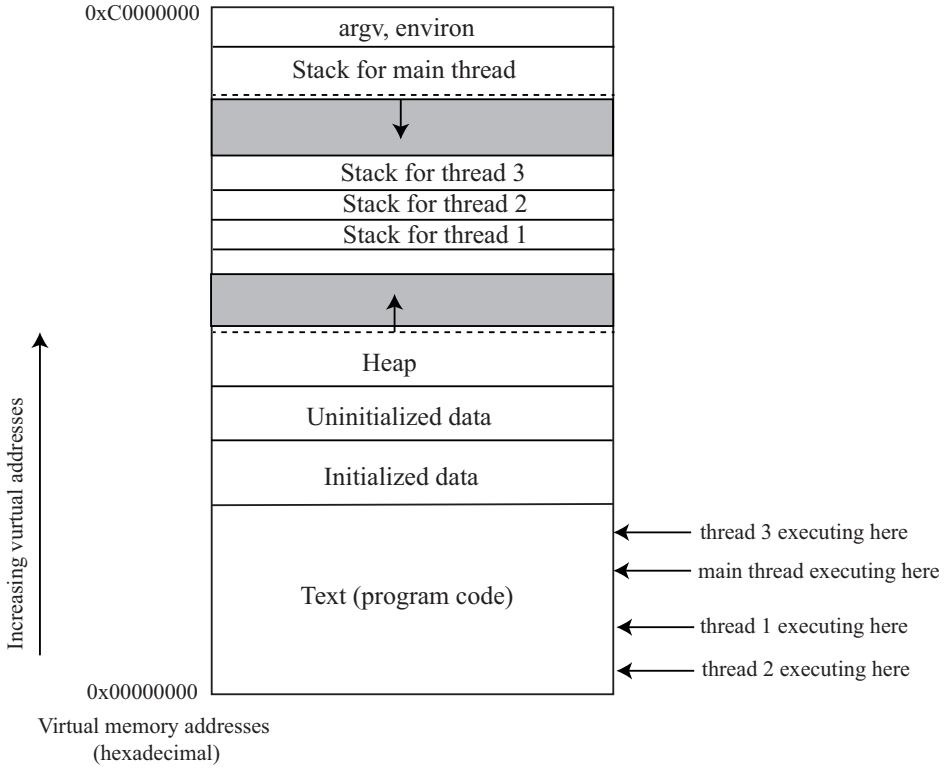

Fig. 1: Four threads executing in the address space of a UNIX process.

- Sharing information between threads is easy and fast; just a matter of copying data into shared (global or heap) variables. However, to avoid race conditions that can occur when multiple threads try to update the same information (via shared variables), we must employ appropriate synchronization mechanisms.
- Thread creation is much faster — up to ten times faster — than creating processes, since many of the attributes that must be duplicated in a child process created using the *fork()* system call are instead shared between threads. In particular, copy-on-write duplication of pages of memory is not required, nor is duplication of page tables.

The *Portable Operating System Interface* (*POSIX*) standard defines the application programming interface (API) for software compatibility with variants of Unix as well as with other operating systems. Linux provides support for the full range of POSIX APIs, including those used for thread creation and management, thread communication, and thread synchronization. The code snippets discussed in the rest of this document use the POSIX API. Threads created using this API are called *POSIX threads* or *pthreads*.

Let us examine the various pthread functions involved in creating and managing threads using code snippets from the file *simple_thread.c* that is available on BBLearn.

Thread Creation

When a program is started, the resulting process consists of a single thread, called the *main* thread. The *pthread_create()* function creates a new thread:

```
int a = 10; /* Global variable stored in the data segment. */
int main(void)
{
    pthread_t main_thread, thr_a;
    main_thread = pthread_self();
    if ((pthread_create(&thr_a, NULL, func_a, NULL)) != 0) {
       printf("pthread_create error\n");
       exit(EXIT_FAILURE);
    }

    pthread_join(thr_a, NULL); /* Wait for thread to finish */

    printf("Value of a is %d\n", a);
    pthread_exit((void *)main_thread);
}
```

The new thread commences execution by calling the function identified in *pthread_create()* — which is *func_a()* in our example. Arguments can be passed to this function as well; *func_a()* takes no arguments in this case and so the last argument in the *pthread_create()* call is set to `NULL`. The code snippet for function *func_a()* executed by our thread is shown below.

```c
void *func_a(void *arg)          /* Function A */
{
    int i;
    thr_a = pthread_self();      /* Obtain thread ID */
    printf("Thread %lu is simulating some processing\n", thr_a);
    for (i = 0; i < 5; i++) {
        a = a + 1;       /* Increment global variable */
        sleep(1);
    }
    pthread_exit((void *)thr_a);
}
```

If the thread is created successfully, a unique identifier for the thread is copied into the variable *thr_a* before *pthread_create()* returns. A thread can also obtain its own ID using *pthread_self()*. We can specify various attributes for the new thread via the second argument to *pthread_create()*. If the attribute parameter is specified as `NULL`, then the thread is created with various default attributes, which is what we will do with all of our programs.

Figure 2 summarizes the high-level structure of a POSIX thread consisting of a pointer to the top-level function executed by the thread, argument list to pass to this function, thread priority, and a pointer to a local stack, specific to each thread, that stores the local variables and nested function calls. Once this structure is created, it can be accessed using a thread identifier.

Consider the following code snippet in which the *pthread_create()* call passes a parameter to the function to be executed by the newly created thread, *func_b()*.

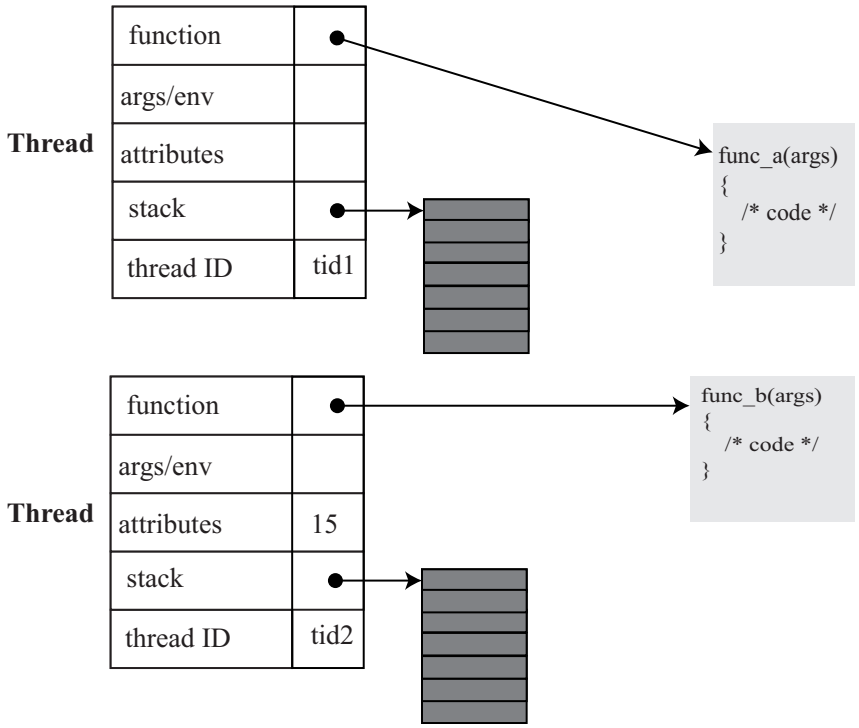

Fig. 2: The POSIX thread structure.

```
int args_for_thread = 5;

if ((pthread_create(&thr_b, NULL, func_b,
                    (void *)args_for_thread)) != 0) {
    printf("pthread_create error\n");
    exit(EXIT_FAILURE);
}
```

The type of the final argument in *pthread_create()* is declared as *void* *, meaning that we can pass a pointer to any type of object to the *func_b()* function. Typically, this pointer points to a global or heap variable, but can also be specified as `NULL` as we did earlier. If we wish to pass multiple arguments to *func_b()*, then we can pass a pointer to a structure containing these arguments as separate fields. Please see the code contained in *multiple_threads.c* for examples.

In the above code snippet, the argument to *func_b()* is specified as an integer — by casting *int* to *void* *, which is permitted by most C compilers and produces the correct result; that is, *int j == (int)((void *)j)*. Since the argument passed to *func_b()* is of the *void* * type, it must be typecast to the correct type — in this case, to *int* — before using it within the function.

```
/* Function B */
void *func_b(void *arg)
{
   int i;
   int args_for_me = (int)arg;
   pthread_t thr_b = pthread_self();

   printf("Thread %lu is using args %d\n", thr_b, args_for_me);
   printf("Thread %lu is processing\n", thr_b);
   for (i = 0; i < 5; i++)
       sleep(2);

   printf("Thread %lu is exiting\n", thr_b);
   pthread_exit((void *)thr_b);
}
```

Thread Termination

The execution of a thread terminates in one of the following ways:

- The thread's start function performs a `return` specifying a return value for the thread.
- The thread calls *pthread_exit()* as shown in the above code snippets. This function terminates the calling thread and specifies a return value that can be obtained in another thread by calling *pthread_join()*. Calling *pthread_join()* is equivalent to performing a return in the thread's start function, with the difference being that *pthread_exit()* can be called from any function that has been called by the thread's start function. If the main thread calls *pthread_exit()* instead of calling *exit()* or performing a `return`, then the other threads continue to execute.

- The thread is canceled using *pthread_cancel()*.
- Any of the threads calls *exit()* or the main thread performs a `return` in the *main()* function, which causes all threads in the process to terminate immediately.

Joining with a Terminated Thread

The *pthread_join()* function waits for the thread identified by its ID to terminate. If that thread has already terminated, *pthread_join()* returns immediately. This operation is called *joining*. By default, a thread is *joinable*, meaning that when it terminates, another thread can obtain its return status using *pthread_join()*. Sometimes, we don't care about the thread's return status but simply want the system to automatically clean up and remove the thread when it terminates. If so, we can mark the thread as *detached* by using the *pthread_detach()* call. Note that detaching a thread does not make it immune to *exit()* in another thread or a *return* in the main thread. In such an event, all running threads are terminated immediately regardless of whether they are joinable or detached.

Thread Attributes

Thread attributes include information such as the location and size of the thread's stack, the thread's scheduling policy and priorities, and whether the thread is joinable or detached.

Thread Synchronization

Semaphore is a programming construct that allows threads to synchronize their accesses to shared memory. Semaphores shared by threads are called *unnamed semaphores*. This type of semaphore does not have a name. Instead it resides in an area of memory shared by threads; for example, on the heap or in a global variable.

Two important functions are needed for unnamed semaphores:

- The *sem_init()* function initializes a semaphore and informs the kernel if the semaphore will be shared between processes or between the threads of a single process.
- The *sem_destroy()* function destroys the semaphore.

The semaphore is an integer whose value is not permitted to fall below zero. If a thread attempts to decrease the value of a semaphore below zero, then depending on the function used, the call either blocks or fails with an error indicating that the operation was not currently possible. Two basic operations can be performed on a semaphore:

- The *sem_wait()* function decrements or locks the semaphore. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement, that is, until the semaphore value rises above zero.
- The *sem_post()* operation increments or unlocks the semaphore. If the semaphore's value consequently becomes greater than zero, then another thread blocked on that semaphore will be woken up and proceed to lock the semaphore.

The following code snippets, extracted from *signalling_using_semaphores.c*, show how semaphores are used to coordinate actions between two threads.

```c
sem_t semaphore;            /* Signals change to the value */
int value = 0;                  /* The value itself */

int main(int argc, char **argv)
{
    pthread_t tid;
    value = 0;              /* Initialize semaphore and value */
    sem_init(&semaphore, 0, 0);

    pthread_create (&tid, NULL, my_thread, NULL);
    sem_wait(&semaphore); /* Probe the semaphore */
    /* Someone changed the value of data to one */
    if (value != 0) {
        printf("Change in variable state was signalled\n");
        printf("Value = %d\n", value);
    }
    pthread_join(tid, NULL);
    pthread_exit(NULL);
}
```

The above code creates the semaphore for signaling between threads. The semaphore is set to be shared between threads executing in the same address space and is initialized with a value of 0. The main thread then blocks on the *sem_wait()* call waiting for the semaphore value to be incremented.

```c
void *my_thread(void *args)
{
    sleep(5);
    value = 1;
    sem_post(semaphore); /* Signal change to blocked main thread */
    pthread_exit(NULL);
}
```

Once the thread has updated *value*, it calls *sem_post()* to increments the semaphore value to greater than zero. This serves as a signal to the main thread to unblock from its *sem_wait()* call.