Linux System Programming Part 8 - Threads

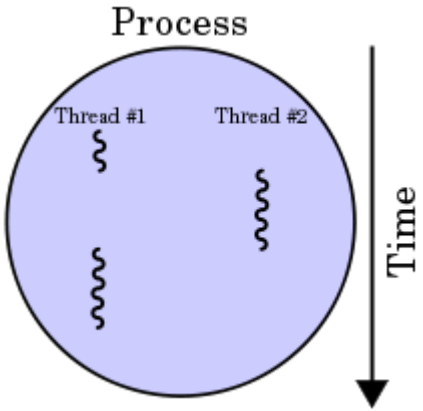
IBA Bulgaria
2018

Threads

A **thread** in computer science is short for a thread of execution. Threads are a way for a program to divide (termed "split") itself into two or more simultaneously (or pseudo-simultaneously) running tasks.

Threads and processes differ from one operating system to another but, in general, a thread is contained inside a process and different threads in the same process share same resources while different processes in the same multitasking operating system do not.

Threads are lightweight, in terms of the system resources they consume, as compared with processes.



libpthreads (POSIX threads)

pthreads defines a set of C programming language types, functions and constants that support applications with requirements for multiple flows of control, called threads, within a process. It is implemented with a pthread.h header and a pthread library.

There are around 100 threads procedures, all prefixed pthread_ and they can be categorized into four groups:

- Thread management - creating, joining threads etc.
- Mutexes
- Condition variables
- Synchronization between threads using read/write locks and barriers

We're only going to cover "Thread management" in this part.

Thread management functions - basics

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

starts a new thread in the calling process

```
int pthread_join(pthread_t thread, void **retval);
```

waits for the thread specified by thread to terminate

```
void pthread_exit(void *retval);
```

terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls **`pthread_join`**(3).

Creating threads

Initially, your **`main()`** program comprises a single, default thread. All other threads must be explicitly created by the programmer. **`pthread_create`** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.

`pthread_create` arguments:

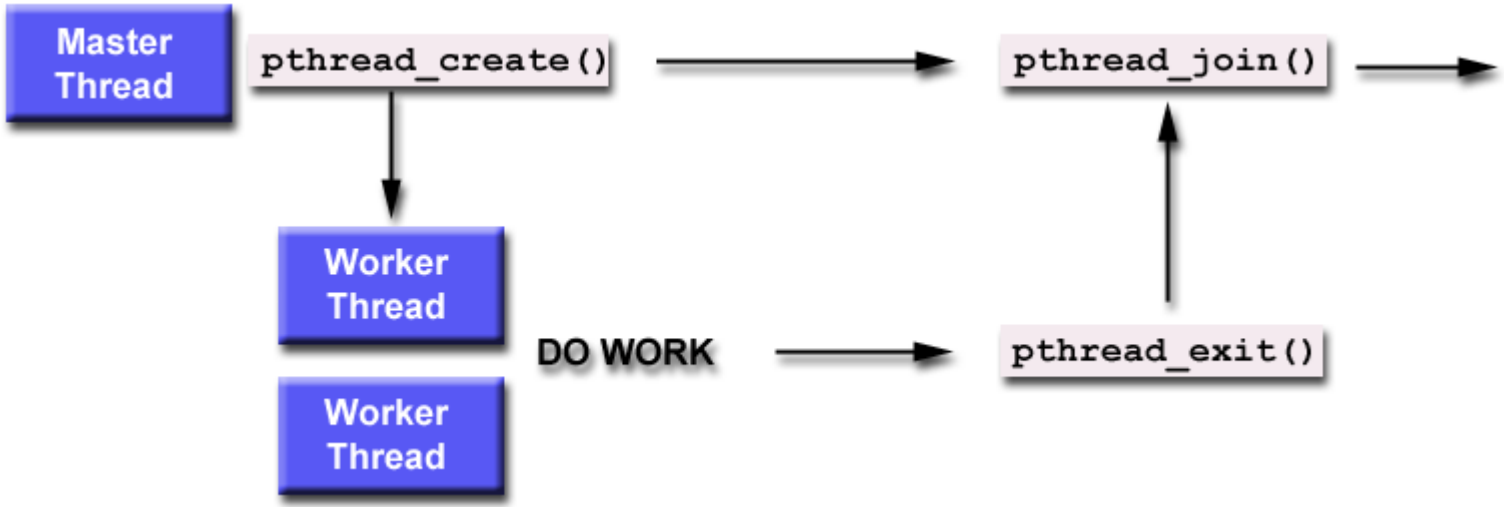
- **`thread`**: An opaque, unique identifier for the new thread returned by the subroutine.
- **`attr`**: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
- **`start_routine`**: the C routine that the thread will execute once it is created.
- **`arg`**: A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

Finishing threads

Terminating Threads & `pthread_exit`():

- There are several ways in which a thread may be terminated:
  - The thread returns normally from its starting routine. Its work is done.
  - The thread makes a call to the **`pthread_exit`** subroutine - whether its work is done or not.
  - The thread is canceled by another thread via the **`pthread_cancel`** routine.
  - The entire process is terminated due to making a call to either the **`exec`()** or **`exit`()**
  - If **main()** finishes first, without calling **`pthread_exit`** explicitly itself

Joining threads



Joining threads

The **`pthread_join`**() subroutine blocks the calling thread until the specified **`threadid`** thread terminates.

The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to **`pthread_exit`**().

Joining two Threads

- Define the helper permutations generation functions
- Define the threads function:
- Allocate a vector of size **arg**
- Initialize the vector with the first permutation and print it
- While there is next permutation:
- Generate it, print it and synchronize
- Deallocate the vector and exit thread

```c
/* compile string:
        gcc threads.c -o threads -lpthread  */
void print_vect(int * v, int n)
...
int next_permutation(int * v, int n)
...
void * thread_func(void *arg){
   int i;
   int * v;
   int size = * (int *) arg;
   v = malloc(sizeof(int)*size);
   for(i = 0; i < size; i++) v[i] = i+1;
   print_vect(v, size);
   while(next_permutation(v, size)) {
      print_vect(v, size);
      sync();
   }
   free(v);
   pthread_exit(arg);
}
```

Joining two Threads (2)

- In the main function function:
- Start thread function with size 4, exit on error
- Start thread function with size 3, exit on error
- Wait for the first thread and print a notification, exit on error
- Wait for the second thread and print a notification, exit on error
- Print “Done” and quit

```
void *ret;
pthread_t thread1, thread2;
size1 = 4;
result = pthread_create(&thread1, NULL,
                        thread_func, &size1);
if (result != 0) { //ERROR
...
size2 = 3;
result = pthread_create(&thread2, NULL,
                              thread_func, &size2);
if (result != 0) { //ERROR
...
result = pthread_join(thread1, &ret);
if (result != 0) { //ERROR
...
} else printf("thread finished with result %i\n",
              * (int *)ret);
result = pthread_join(thread2, &ret);
if (result != 0) { //ERROR
...
} else printf("thread finished with result %i\n",
              * (int *)ret);
printf("Done\n");
```

Thread management functions - cancellations

`int pthread_cancel(pthread_t thread);`

sends a cancellation request to the thread **`thread`**.

`int pthread_setcancelstate(int state, int *oldstate);`

sets the cancelability state of the calling thread to the value given in **`state`** – **`PTHREAD_CANCEL_ENABLE`** or **`PTHREAD_CANCEL_DISABLE`**

`int pthread_setcanceltype(int type, int *oldtype);`

sets the cancelability type of the calling thread to the value given in **`type`** - **`PTHREAD_CANCEL_DEFERRED`** or **`PTHREAD_CANCEL_ASYNCHRONOUS`**

Thread management functions - types

The **`pthread_cancel`**() function sends a cancellation request to a 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** (enabled/disabled) and **type**:

- **Deferred** (default behaviour) cancelability means that cancellation will be delayed until the thread next calls a function that is a cancellation point (**pthread_testcancel**() and others).

- **Asynchronous** cancelability means that the thread can be canceled at any time (usually immediately, but the system does not guarantee this).

Thread cancellations - basic scenario

- Create a thread
- Try to cancel it
- Wait for it to finish

canceltest.c

```c
int main(int argc, char * argv[])
{
  pthread_t thread;
  pthread_create(&thread, NULL, thread_func, NULL);
  while (i < 1) sleep(1);
  pthread_cancel(thread);
  printf("Requested to cancel the thread\n");
  pthread_join(thread, NULL);
  printf("The thread is stopped.\n");
  return EXIT_SUCCESS;
}
```

Thread cancellability STATE

- Mark the thread as uncancellable
- … while it's working
- … and until we enable cancellations
- … AND reach a cancelability point

canceltest.c

```c
void * thread_func(void * arg)
{
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  for(i=0; i < 4; i++) {
    sleep(1);
    printf("I'm still running!\n");
  }
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  pthread_testcancel();
  printf("YOU WILL NOT STOP ME!!!\n");
}
```

Thread cancellability type

canceltest2.c

- Make the thread cancellable at any time
- But mark it in a noncancelable state for now
- … while it's working
- … and until we enable cancellations
- The thread gets cancelled automatically, no need to reach a cancellation point

```c
void * thread_func(void * arg)
{
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  for(i=0; i < 4; i++) {
    sleep(1);
    printf("I'm still running!\n");
  }
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  printf("YOU WILL NOT STOP ME!!!\n");
}
```

Exercise

Project *ThreadedChat*:

Refactor the network server/client pair of programs from the previous lecture with the following changes:

- The multi-user handling in the server should be done using of **threads**.
- The server stops when one of the clients sends "**!quitserver**" to the server.