Linux Threads

Purpose

The purpose of this document is to present some functions of the PTHREADS library, which allow creating and managing multiple threads of a process in the Linux operating system.

1. Creating a thread

The function used to create a thread is called *pthread_create* and has the following syntax:

```
#include <pthread.h>
int pthread_create (
      pthread_t* idThread,
      const pthread_attr_t* threadAttributes,
      void* (*thread_routine)(void*),
      void* arg);
```

The meaning of the parameters is:

`idThread`
: The memory address where the identifier of the newly created thread will be stored.

`threadAttributes`
: The memory address of a structure containing values of attributes used when creating the thread. If this parameter is set to NULL, then the default values are used for the thread's attributes.

`thread_routine`
: The memory address of the function that the thread will execute.

`arg`
: The address where the argument passed to the function executed by the thread is found. The argument can have any desired structure.

The function returns 0 in case of a success and a positive value, which represents the error code, in case of failure.

The created thread is executed concurrently with the thread that called the function *pthread_create* and any other threads of the process. It begins its execution by calling `thread_routine` function and it will implicitly end at the end of the function or explicitly by calling the *pthread_exit* function described later.

From the point of view of the relationship that exists between the threads, all threads of a process are identical. The only exception is the main thread, the one that was created when the process was created and that executes the *main* function. Terminating this thread ends the process and all the other threads, too. So in a normal execution, the thread execution function *main* waits for all the other threads to terminate.

The following example shows the way a thread is created.

```
void* thFunction(void* arg) {
      int* val = (int*) arg;

      printf("Thread with argument %d\n", *val);
}

main() {
      pthread_t th1;
      int arg1 = 1;

      pthread_create(&th1, NULL, thFunction, &arg1);
      pthread_join(th1, NULL);
}
```

Important note. When compiling a C program that uses the functions from the PTHREADS library, you have to ask explicitly the compiler to link that library. Consequently, the command line will look like the following one:

```
gcc progr.c -lpthread -o progr.exe
```

2. The thread's identifier

Every thread has an unique ID assigned to it at the moment of its creation. This ID is obtained by the thread that calls the *pthread_create* function. The ID of a thread can be used, for example, by another thread to indicate that the latter one should wait calling function *pthread_join* till the first thread ends its execution. A thread can obtain its own ID calling *pthread_self* function. To compare the IDs of two different threads, function *pthread_equal* can be used. The syntax of these two functions is:

```
#include <pthread.h>
pthread_t pthread_self(void);

int pthread_equal( pthread_t thread1,
                   pthread_t thread2);
```

Function *pthread_self* can be used to get thread's ID independently of the operating system the PTHREAD library is implemented on. On Linux, one can also use *gettid* function to get the identifier allocated to thread by the operating system. Obviously, using such OS specific functions will make the multi-threaded application non portable on other operating systems where the PTHREAD packet is also supported. The syntax of *gettid* function is the following:

```
#include <sys/types.h>
pid_t gettid(void);
```

3. Ending the execution of a thread

Usually a thread terminates its execution when it returns from the function it executes. Another way to voluntarily end a thread is to call function *pthread_exit*, to which one can specify information related to the termination status of the thread.

The syntax of this function is:

```
#include <pthread.h>
void pthread_exit(void *retval);
```

In case the thread ended without explicitly calling function *pthread_exit*, the information about the termination status of that thread will be the one in the return value of the function executed by the thread. So, the information about the termination status of a thread is given by the value of the *retval* parameter of the function *pthread_exit*, in case this function was called, or by the returned value of the function executed by the thread.

Among the attributes of a thread there is one that indicates the system whether to keep or not information about the termination status of a thread. These two alternatives correspond to the situations in which another thread may wait for a thread to end (calling function *pthread_join*) and obtain information about its termination status or not. In the second case, at the moment a thread ends, all its resources are freed. The way the value for the mentioned attribute is set is described in the followings.

An exterior way to force the termination of a thread is for another thread to call function *pthread_cancel*. The syntax is given below:

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

A thread can be ended only by another thread of the same process. The permission to terminate a thread from another thread is given by another of the thread's attributes, which is the one named *cancelability state*. That attribute can be set using the function *pthread_setcancelstate*, whose syntax is given below:

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
```

The possible values for the *state* parameter are:

- `PTHREAD_CANCEL_ENABLE`: the thread can be stopped;
- `PTHREAD_CANCEL_DISABLE`: the thread cannot be stopped.

When a thread having this feature enabled (that it can be stopped by *pthread_cancel*), receives from another thread a request to cancel its execution, it can chose between immediately ending its execution or waiting till it reaches a so called *cancel point*. The way a thread acts when it gets a cancel request is determined by the value of another of its attributes, i.e. *cancelability type*, which can be set using the function *pthread_setcanceltype*. The syntax is as follows:

```
#include <pthread.h>
int pthread_setcanceltype(int type, int *oldtype);
```

The *type* parameter can have the following values, corresponding to the two cases described above:

- `PTHREAD_CANCEL_ASYNCHRONOUS`: immediate cancel;
- `PTHREAD_CANCEL_DEFERRED`: delayed cancel.

Cancel points are those points in the execution of a thread where the system checks if there are cancel requests and the corresponding operations are carried out. The POSIX standard defines as cancel points the following functions: *pthread_join*, *pthread_cond_wait*, *pthread_cond_timedwait*, *pthread_testcancel*, *sem_wait*, *sigwait*, *sleep* and any function that puts the thread in waiting. Other POSIX functions aren't considered cancel points, which means that when called and during their execution the thread cannot end. From the function described above as being cancel points, only the function *pthread_testcancel* represents a cancel point that is set explicitly with that purpose by the programmer. All others are considered to be automatic cancel points.

Function *pthread_testcancel* has the following syntax:

```
#include <pthread.h>

void pthread_testcancel(void);
```

The default values for the two attributes that concern the forced termination of a thread from exterior by another thread, are:

- Cancel state: `PTHREAD_CANCEL_ENABLE`
- Cancel type: `PTHREAD_CANCEL_DEFFERRED`

Terminating a thread from another thread has to be made with a lot of caution because a couple of serious problems may be generated. This is because in the moment the thread receives a cancel request, the thread can contain global data that is in an inconsistent state or hold resources (locks, semaphores etc.) needed by other threads and which must be released before the thread ends. So, those code sequences that contain critical information must be protected against uncontrolled forced endings. This can be done by setting the cancel type to PTHREAD_DEFERRED and by specifying certain functions that have to be executed in the moment the thread ends, either by calling *pthread_exit*, or by calling *pthread_cancel* by another thread. The functions executed at the end of a thread are specific to each thread and its actions, and they mainly should have the purpose of freeing the resources and to bring the used data by the thread into a consistent state. This can be done with the help of the following functions:

- *pthread_cleanup_push*, used to add a new function to the list of functions that need to be executed when the thread terminates;
- *pthread_cleanup_pop*, used to pop the last added function from the list of functions that need to be executed when the thread terminates.

For a certain thread there are more functions that can be set to be executed when the thread ends. The order of execution is done in the stack functioning manner (LIFO).

The syntax for the two functions are:

```
#include <pthread.h>

void pthread_cleanup_push(
            void (*routine)(void*),
            void* arg);

void pthread_cleanup_pop(int execute);
```

The meaning of the parameters is:

`routine`
: The function to be executed at the end of the thread.

`arg`
: The argument passed to this function.

`execute`
: Indicates whether the function popped from the stack has to be executed or not.

An important detail to remember is that these two functions must always be used in pairs because they are defined as macros; *pthread_cleanup_push* uses the open bracket ‘{', while *pthread_cleanup_pop* uses the closing bracket '}'. Unfortunately this defining mode of the two macros imposes some utilization restrictions: they cannot be independently part in different instruction blocks, for example in different for instructions.

The following example illustrates the use of these two functions in the case in which a thread dynamically allocates memory, which has to be freed when thread terminates.

```
typedef struct m {
      int size;
      void* pMem;
} MEM;

void allocate_mem(void* arg) {
        MEM* p = (MEM*) arg;
        p->pMem = malloc(p->size);
}

void release_mem(void* arg) {
        MEM* p = (MEM*) arg;
        if (p->pMem)
                free(p->pMem);
}

void* thFunction(void* arg){
        int oldType;
        MEM  thMem;

        pthread_setcanceltype(
                PTHREAD_CANCEL_DEFERRED, &oldType);

        thMem.size = 100;
        thMem.pMem = NULL;

        pthread_cleanup_push(
                release_mem, (void *) &thMem);

        allocate_mem(&thMem);

          /* do some work with the memory*/

        pthread_cleanup_pop(1);

        pthread_setcanceltype(oldType, NULL);
}
```

4. Waiting for a thread to end

A particular and limited way to synchronize the execution of two different threads is by blocking a thread to wait until another thread ends its execution. This can be done by using the function *pthread_join*. Its syntax is:

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

```
      int pthread_join(pthread_t th, void **thread_return);
```

Calling this function will result in blocking the execution of the calling thread until the end of the *th* thread, passed as the first argument. Information about the termination of the thread for which it waits is organized in a structure whose address will be the one in the *thread_return* parameter, if its value is different from NULL. The type of the *thread_return* parameter is given by the signature of the function executed by the thread: the function returns type void*. If the waiting thread was forced to end, then the value of the parameter *thread_return* is a predefined value stored in the PTHREAD_CANCELED constant.

Note that this function can be called only in case of those threads for which the system holds information about their termination. That property is given by the value of a thread's attribute we describe below.

The function *pthread_join* can successfully be called only once for each thread. If a thread calls the function *pthread_join* and later on another thread calls the same function, the function will return an error code instead of blocking the execution of the thread.

5. Setting the attributes of a thread

A thread has certain properties which we call attributes. Some of these attributes were already discussed, like for example the way threads handle cancel requests from outside.

Other attributes of a thread are:

- the property of the thread to be joined by another thread calling the function *pthread_join*;
- the dimension of the stack;

- placing the stack in the address space of the process;
- attributes related to the scheduling of the thread's execution;

The way to set the attributes of a thread is to build a structure of type *pthread_attr_t* and modify the default values of these fields by using different functions. The address of this structure must then be passed to the function *pthread_create*. If the NULL value is passed, then the thread is created having the default values for its attributes. The functions to create and to destroy such an attribute structure are *pthread_attr_init* and *pthread_attr_destroy*, having the following syntax:

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

For now, the only important attribute is the one related to the possibility for calling the function *pthread_join* for a particular thread. Setting the value of this attribute and obtaining its current value can be done by using the following functions:

```
#include <pthread.h>
int  pthread_attr_setdetachstate(
                  pthread_attr_t *attr,
                  int detachstate);

int  pthread_attr_getdetachstate(
                  const pthread_attr_t* attr,
                  int *detachstate);
```

The possible values for the *detachstate* parameter are:

- `PTHREAD_CREATE_DETACHED`: another thread can't wait till the end of the thread, so implicitly, the information about the termination status of the thread is not kept.

- `PTHREAD_CREATE_JOINABLE`: Can wait for the end of the thread. The information about the termination status of the thread is kept until obtaining them through *pthread_join* function. This is the default value of the attribute of the thread.

There are functions with similar names used to modify and obtain the other attributes of a thread:

- *pthread_attr_getstacksize* and *pthread_attr_setstacksize*;
- *pthread_attr_getstackaddr* and *pthread_attr_setstackaddr* etc.

If a thread is set to be ”detached” then in the moment that thread ends, the system will no longer hold information about its termination status and all the allocated resources will be freed, thus calling function *pthread_join* won't succeed for that thread. A similar behavior can be obtained with the help of the *pthread_detach* function that can be called after creating the thread. Calling this function for a thread for which there is another thread already waiting for it to end, will have no effect and the value of the attribute will remain unchanged. The syntax for this function is:

```
#include <pthread.h>
int pthread_detach(pthread_t th);
```

6. The relation between threads and processes

Behavior when signals are present

One of the problems related to processes that contain more threads is the way the signals are used, so the way in which signals are passed to processes that contain more threads and the reaction of these processes towards signals sent by other processes.

The PTHREADS standard says that a signal can only be sent to a process and not to a thread of that process. This is obvious, since, from the outside, the existence of a thread cannot be detected, because they are part of the internal structure of the process. But in Linux, threads are implemented as special processes (so called *lightweight processes*) so some aspects related to signals have to be clarify.

The response to a received signal, as well as the functions that handle signals are common to all threads of a process and is set using the function *signal* or *sigaction*. On the other hand, defining the filter (mask) to block specific signals is unique to each thread and can be defined separately for each thread. Defining the mask for a thread is done with the help of the function *pthread_sigmask*, and for the whole process with the function *sigprocmask*. A signal sent to a process is received by one of the threads that didn't block receiving that signal, without knowing in advance which one is that.

The following example shows the way a signal and the blocking mask are handled:

```
void sigHandler(int sig) {
 printf("Thread %d handles the signal\n",
        pthread_self());
}

main() {
      pthread_t th1;
      signal(SIGUSR1, sigHandler);
      pthread_create(&th1, NULL, thFunction, NULL);
      pthread_join(th1, NULL);
}

void* thFunction(void* arg) {
      sigset_t sigmask;
      sigemptyset(&sigmask);
      sigaddset(&sigmask, SIGUSR1);
      pthread_sigmask(SIG_SETMASK, &sigmask, NULL);
      while(1);
}
```

The effect of the *fork* function call

Another problem that occurs in the case of processes with multiple threads is when one of the threads of a process calls function *fork*, which creates a new process. The question is whether the threads existing in the parent process will exist in the child process too, when the function *fork* is called? The PTHREADS standard says that even though the child process is identical with the parent process in content, the only active thread in the newly created process will be the one that called the *fork* function. So, the execution of the new process corresponds with the execution of the active thread and will end when this thread ends. Taking into account the implementation of threads in Linux through processes, this behavior is obvious, being created a duplicate only of the process (thread) that calls *fork* function. Because of this kind of behavior of the *fork* function, a couple of problems may appear in the child process concerning the status of those threads that are no longer active in the child. It is possible, for instance, that when creating a new process one of the inactive threads had previously blocked (with the help of a lock) the access to a shared resource. Freeing the lock, thus freeing the access to the shared resource can only be done by the thread that blocked the lock. But this thread being inactive in the child process, it is possible for the active thread to remain definitely blocked waiting for the unblocking of the lock. To avoid such situations, a function named *pthread_atfork* can be used, through which you can specify the functions to be executed, in the parent process and the child process too, before and after the execution of the *fork* function. The syntax for this function is:

```
#include <pthread.h>

pthread_atfork( void (*prepare)(void),
                void (*parent)(void),
                void (*child)(void));
```

The meaning of the parameters is:

`prepare`
: The function that will be executed in the parent process before a new process is created.

`parent`
: The function that will be executed in the parent process before function *fork* finishes its execution.

`child`
: The function that will be executed in the child process before function *fork* finishes its execution.

Function *pthread_atfork* can be called more than one time, setting a list of functions that will be executed in the LIFO order.

7. Problems

1. Write a program that tests whether the threads of a process are executed concurrently or not. The created threads should call a function that contains an infinite loop that prints out the ID of the thread and the ID of the process to which the thread belongs to.
2. Using the program written for the previous problem, test the two different reactions of a thread to a cancel request coming from another thread through the call of the *pthread_cancel* function.
3. Write a program that specifies the maximum number of threads that can simultaneously exist in the same process. To avoid overloading the processor, the created threads will call the function *sleep* inside an infinite loop. To find the wanted number, the value returned by the *pthread_create* function is tested, so to detect the moment when no more threads can be created.
4. Write a server type C program that periodically creates threads that simulate the handling of some requests from clients. The server

threads display a message, wait for a while (using *sleep* function) and then terminate. In the same time, the server also accepts commands introduced from keyboard. Implement the stop command of the server, like when a special key is pressed (for example 'x'): the process should not create new threads and end. But ending the process should be done in a clean state, only after the termination of the existing threads at that time.

5. Modify Problem 4, such that the threads that simulate the handling of the client requests to execute in an infinite loop intensive computational operations (for example calculating the first N prime numbers or just doing "nothing"). When a key is pressed, the server has to display the number of threads created till that moment. Follow how the *computational* threads influence the reaction time of the thread that reads from the keyboard (the *interactive* one).
6. Test the behavior of different threads of the same process that simultaneously access functions that read from the keyboard.
7. Test the effect of the *pthread_join* function called by the only active thread of a child process, in order to wait till the end of a thread in the parent process, existing there when function *fork* is called, but not active in the child process. The question is if the active thread will be blocked in the *pthread_join* or not; if not, what does that function returns?
8. Write a C program that copies the content of a file into another file using more than one thread. The copy is made to places with different dimensions (for example 512 bytes, 1Kb, 2Kb, 4 Kb etc.). To emphasize the need of threads in such a situation and to acknowledge the number of threads for which this operation is efficient, make a comparison between the following implementations of the problem:
    a. one single thread

   b. N threads created at the beginning of the program's execution, each should copy a part of the file
   c. an existing thread should create a new thread only before the file access functions are called (read and write), which may put it in a waiting state

9. Implement the same tests like in the previous problem, but for the case when the part of the file that is to be copied has to be processed before it is copied into the destination file. An example of such a processing might be to write the bytes in the reverse order.

10. It is known that in a *numbers.in* text file, on each line there are two integers. Write the C program that reads each line of the file and creates a new thread for each line read. The thread gets as parameters the two numbers from the line (in the form of a structure). Each thread takes the two parameters, makes their arithmetic average and writes into the *result_threads.out* file on a single line the three numbers and the id of the thread. It should also pass the arithmetic average as a parameter to the *pthread_exit* function. After finishing reading from the *numbers.in* file, the *main* thread will take all the results sent by the created threads and will write them into the *results_main.out* file together with the pid of each created thread. At the end, compare the contents of the two resulted files.

11. Suppose that the *main* thread of a process is blocked, waiting for another thread of the same process to end. You may also suppose that concerning the reaction to the SIGUSR1 signal, a function handle has been established previously and that the thread for which it waits had already masked the SIGUSR1 signal with the help of the function *pthread_sigmask*. Test whether by passing the SIGUSR1 signal to the process, the *main* thread will react or not. If it does, see whether it still waits for the other thread to finish its execution or not.

12. Write two C programs corresponding to two processes, each of them having two threads, such that the communication between the two processes through signals SIGUSR1 and SIGUSR2, to be done at the level of a pair of threads for each signal. Thus, through signal SIGUSR1 only those two threads will communicate that were created first in the two processes, and through signal SIGUSR2 only the other two threads will be able to communicate.

13. Suppose that a global variable *var* from a process is in a consistent state only if it has a predefined value (for instance *var=10*). We also suppose that the process creates more threads that all call the same *threadFunct* function, function that modifies the value of the variable *var*. Modifying the value of the variable can only be done using functions *add(int val)* and *subtract(int val)*. Write the *threadFunct* function that will be executed by the threads, such that when all created threads finish their execution, the value of the variable has to be the initial one; the initial state is considered to be the consistent state of the variable, regardless whether the threads ended their execution in a normal way or they were ended by the *main* thread by calling *pthread_cancel* function.

14. In the context described in the previous problem, make sure that using the *pthread_atfork* function, in the child process the value of the variable *var* is the one corresponding to its consistent state. You may consider that it is possible that the thread that calls the *fork* function could have modified the value of the variable before the call of *fork*.

15. Write a program that would continuously generate threads. Suppose that the allocation of the structure *pthread_t* associated to a thread and the allocation of the parameters of the executed function by the thread is done dynamically. To free the memory allocated to the threads that finished their execution, create another thread called the *garbage collector*.