

Linux Internals & Networking

System programming using Kernel interfaces

Team Emertxe



Contents



Linux Internals & Networking

Contents

- .Introduction
- .Transition to OS programmer
- .System Calls
- .Process
- .IPC
- .Signals
- .Networking
- .Threads
- .Synchronization
- .Process Management
- .Memory Management



Threads

A decorative horizontal bar at the bottom of the slide. It features a gradient from bright pink on the left to deep purple on the right. On the far right, there is a graphic of two overlapping chevrons pointing to the right, with the front chevron in a darker purple and the back one in a lighter pink.

Threads

Introduction to Threads



- .Threads, like processes, threads are a mechanism that permits an application to perform multiple tasks concurrently.
- .A single process can contain multiple threads.
- .Threads independently executes the same program, and they all share the same global memory, including the initialized data, uninitialized data, and heap segments.
- .The threads in a process can execute concurrently.
- .On a multiprocessor system, multiple threads can execute parallel.
- .If one thread is blocked on I/O, other threads are still eligible to execute.
- .The Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute
- .Threads are a finer-grained unit of execution than processes
- .One thread can create additional threads; all these threads run the same program in the same process
- .But each thread may be executing a different part of the program at any given time

Team Emertxe





• It is difficult to share information between processes.

- Since the parent and child don't share memory, some form of inter process communication is needed in order to exchange information between processes.

• Process creation with `fork()` is relatively expensive.

- Even with the copy-on-write technique, the need to duplicate various process attributes such as page tables and file descriptor tables means that a `fork()` call is still time-consuming.



- Sharing information between threads is easy and fast.
 - It is just a matter of copying data into shared (global or heap) variables.
 - In order to avoid the problems that can occur when multiple threads try to update the same information, the synchronization techniques are employed.
- Takes less time to create a new thread in an existing process than to create a brand new process
 - Thread creation is faster because many of the attributes that must be duplicated in a child created by fork() are instead shared between threads.
 - In particular, copy-on-write duplication of pages of memory is not required, nor is duplication of page tables.
- Switching between threads is faster than a normal context switch

Team Emertxe



Threads

Compilation



.Use the following command to compile the programs using thread libraries

```
.$ gcc -o <output_file> <input_file.c> -lpthread
```

Threads

Creation



.The **pthread_create** function creates a new thread

Function	Meaning
<pre>int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)</pre>	<ul style="list-style-type: none">✓ A pointer to a pthread_t variable, in which the thread ID of the new thread is stored✓ A pointer to a thread attribute object. If you pass NULL as the thread attribute, a thread will be created with the default thread attributes✓ A pointer to the thread function. This is an ordinary function pointer, of this type: void* (*) (void*)✓ A thread argument value of type void *. Whatever you pass is simply passed as the argument to the thread function when thread begins executing

Threads

Creation



- A call to **pthread_create** returns immediately, and the original thread continues executing the instructions following the call
- Meanwhile, the new thread begins executing the thread function
- Linux schedules both threads asynchronously
- Programs must not rely on the relative order in which instructions are executed in the two threads



- 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()**
 - 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.
- Calling **pthread_exit()** is equivalent to performing a **return** in the thread's start function, with the difference 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.



- Each thread within a process is uniquely identified by a thread ID.
- This ID is returned to the caller of `pthread_create()`, and a thread can obtain its own ID using `pthread_self()`.
- Thread IDs are useful within applications for the following reasons:
 - Various Pthreads functions use thread IDs to identify the thread on which they are to act.
- Examples of such functions include `pthread_join()`, `pthread_detach()`, `pthread_cancel()`, and `pthread_kill()`.
- In some applications, it can be useful to tag dynamic data structures with the ID of a particular thread.



- Occasionally, it is useful for a sequence of code to determine which thread is executing it.
- Also sometimes we may need to compare one thread with another thread using their IDs
- Some of the utility functions help us to do that

Function	Meaning
<code>pthread_t pthread_self()</code>	✓ Get self ID
<code>int pthread_equal(pthread_t threadID1, pthread_t threadID2);</code>	✓ Compare threadID1 with threadID2 ✓ If equal return non-zero value, otherwise return zero

Threads

Joining



- It is quite possible that output created by a thread needs to be integrated for creating final result
- So the main program may need to wait for threads to complete actions
- The `pthread_join()` function helps to achieve this purpose

Function	Meaning
<code>int pthread_join(pthread_t thread, void **value_ptr)</code>	<ul style="list-style-type: none">✓ Thread ID of the thread to wait✓ Pointer to a void* variable that will receive thread finished value✓ If you don't care about the thread return value, pass NULL as the second argument.



- The `pthread_join()` function waits for the thread identified by `thread` to terminate.
- If that thread has already terminated, `pthread_join()` returns immediately.
- If `retval` is a non- NULL pointer, then it receives a copy of the terminated thread's return value—that is, the value that was specified when the thread performed a return or called `pthread_exit()`.
- Calling `pthread_join()` for a thread ID that has been previously joined can lead to unpredictable behavior.
- If a thread is not detached, then we must join with it using `pthread_join()`.
- If we fail to do this, then, when the thread terminates, it produces the thread equivalent of a zombie process.



–Aside from wasting system resources, if enough thread zombies accumulate,



- The task that `pthread_join()` performs for threads is similar to that performed by `waitpid()` for processes.
- Threads are peers. Any thread in a process can use `pthread_join()` to join with any other thread in the process.
- This differs from the hierarchical relationship between processes.
- When a parent process creates a child using `fork()`, it is the only process that can `wait()` on that child.

Threads

Detaching a Thread



- By default, a thread is joinable, meaning that when it terminates, another thread can obtain its return status using `pthread_join()`.
- Sometimes, thread's return status is not needed; we want the system to automatically clean up and remove the thread when it terminates.
- In this case, we can mark the thread as detached, by making a call to `pthread_detach()` specifying the thread's identifier in `thread`.
- Once a thread has been detached, it is no longer possible to use `pthread_join()` to obtain its return status, and the thread can't be made joinable again.

Threads

Passing Data



- The thread argument provides a convenient method of passing data to threads
- Because the type of the argument is **void***, though, you can't pass a lot of data directly via the argument
- Instead, use the thread argument to pass a pointer to some structure or array of data
- Define a structure for each thread function, which contains the “parameters” that the thread function expects
- Using the thread argument, it's easy to reuse the same thread function for many threads. All these threads execute the same code, but on different data

Threads

Return Values



- If the second argument you pass to **pthread_join** is non-null, the thread's return value will be placed in the location pointed to by that argument
- The thread return value, like the thread argument, is of type **void***
- If you want to pass back a single int or other small number, you can do this easily by casting the value to **void*** and then casting back to the appropriate type after calling **pthread_join**

Threads

Attributes



- .Thread attributes provide a mechanism for fine-tuning the behaviour of individual threads
- .Recall that **pthread_create** accepts an argument that is a pointer to a thread attribute object
- .If you pass a null pointer, the default thread attributes are used to configure the new thread
- .However, you may create and customize a thread attribute object to specify other values for the attributes

Threads

Attributes



- .There are multiple attributes related to a
- .particular thread, that can be set during creation
- .Some of the attributes are mentioned as follows:

- Detach state
- Priority
- Stack size
- Name
- Thread group
- Scheduling policy
- Inherit scheduling

Threads

Joinable and Detached



- A thread may be created as a **joinable thread** (the default) or as a **detached thread**
- A joinable thread, like a process, is not automatically cleaned up by GNU/Linux when it terminates
- Thread's exit state hangs around in the system (kind of like a zombie process) until another thread calls **pthread_join** to obtain its return value. Only then are its resources released
- A detached thread, in contrast, is cleaned up automatically when it terminates
- Because a detached thread is immediately cleaned up, another thread may not synchronize on its completion by using **pthread_join** or obtain its return value

Threads

Creating a Detached Thread



- In order to create a detached thread, the thread attribute needs to be set during creation
- Two functions help to achieve this

Function	Meaning
<code>int pthread_attr_init(pthread_attr_t *attr)</code>	<ul style="list-style-type: none">✓ Initializing thread attribute✓ Pass pointer to <code>pthread_attr_t</code> type✓ Returns integer as pass or fail
<code>int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);</code>	<ul style="list-style-type: none">✓ Pass the attribute variable✓ Pass detach state, which can take<ul style="list-style-type: none">• <code>PTHREAD_CREATE_JOINABLE</code>• <code>PTHREAD_CREATE_DETACHED</code>

Threads

Threads Vs Process



•Advantages of Multi threaded approach

- Sharing data between threads is easy. By contrast, sharing data between processes requires more work
- Thread creation is faster than process creation; context-switch time may be lower for threads than for processes.

•Disadvantages of Multi threaded approach

- When programming with threads, we need to ensure that the functions we call are thread-safe or are called in a thread-safe manner.
- A bug in one thread (e.g., modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes.
- Each thread is competing for use of the finite virtual address space of the host process.
- In particular, each thread's stack and thread-specific data consumes a part of the process virtual address space, which is consequently unavailable for other threads.

Team Emertxe



Threads

Canceling the Thread



.The `pthread_cancel()` function sends a cancellation request to the specified thread.

Header	<pthread.h>
Prototype	<code>int pthread_cancel(pthread_t thread);</code>
Return Value	Returns 0 on success, or a positive error number on error

.`pthread_cancel()` returns immediately; that is, it doesn't wait for the target thread to terminate.

Threads

Cancellation State & Type



.The `pthread_setcancelstate()` and `pthread_setcanceltype()` functions set flags that allow a thread to control how it responds to a cancellation request.

Header	<pthread.h>
Prototype	<code>int pthread_setcancelstate(int state , int * oldstate);</code> <code>int pthread_setcanceltype(int type , int * oldtype);</code>
Return Value	Returns 0 on success, or a positive error number on error

.The `pthread_setcancelstate()` function sets the calling thread's cancelability state to the value given in state.

<code>PTHREAD_CANCEL_DISABLE</code>	.The thread is not cancelable. .If a cancellation request is received, it remains pending until cancelability is enabled.
<code>PTHREAD_CANCEL_ENABLE</code>	.The thread is cancelable. .This is the default cancelability state in newly created threads.

Threads

Cancellation State & Type



.If a thread is cancelable (`PTHREAD_CANCEL_ENABLE`), then a cancellation request is determined by the thread's cancelability type, which is specified by the type argument in a call to `pthread_setcanceltype()`.

<code>PTHREAD_CANCEL_ASYNCHRONOUS</code>	.The thread may be canceled at any time
<code>PTHREAD_CANCEL_DEFERRED</code>	.The cancellation remains pending until a cancellation point is reached.

.When a thread calls `fork()`, the child inherits the calling thread's cancelability type and state.

.When a thread calls `exec()`, the cancelability type and state of the main thread of the new program are reset to `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED`, respectively.

`oldtype` can be specified as `NULL`, if previous cancelability type is not needed.

Threads

Cancellation Points



- When cancelability is enabled and deferred, a cancellation request is acted upon only when a thread next reaches a cancellation point.
- A cancellation point is a call to one of a set of functions defined by the implementation.
 - Example: `open()`, `read()`, `write()`, `close()`
- If the thread was not detached, then some other thread in the process must join with it, in order to prevent it from becoming a zombie thread.
- When a canceled thread is joined, the value returned in the second argument to `pthread_join()` is a special thread return value: **PTHREAD_CANCELED**.

Threads

Cleanup Handlers



.Situation: A thread with a pending cancellation were simply terminated when it reached a cancellation point

- shared variables and Pthreads objects (e.g., mutexes) might be left in an inconsistent state, perhaps causing the remaining threads in the process to produce incorrect results, deadlock, or crash.

.To get around this problem,

- Thread can establish one or more cleanup handlers
 - A cleanup handler can perform tasks such as modifying the values of global variables and unlocking mutexes before the thread is terminated.

Cleanup Handlers: functions that are automatically executed if the thread is canceled.

Threads

Cleanup Handlers



- .Each thread can have a stack of cleanup handlers.
- .When a thread is canceled, the cleanup handlers are executed working down from the top of the stack.
- .When all of the cleanup handlers have been executed, the thread terminates.

Cleanup Handlers: functions that are automatically executed if the thread is canceled.

Threads

Cleanup Handlers



.The `pthread_cleanup_push()` and `pthread_cleanup_pop()` functions respectively add and remove handlers on the calling thread's stack of cleanup handlers.

```
#include <pthread.h>

void pthread_cleanup_push(void (* routine )(void*), void * arg );
void pthread_cleanup_pop(int execute );
```

Team Emertxe



Thank You