Linux Internals

Team Embedded

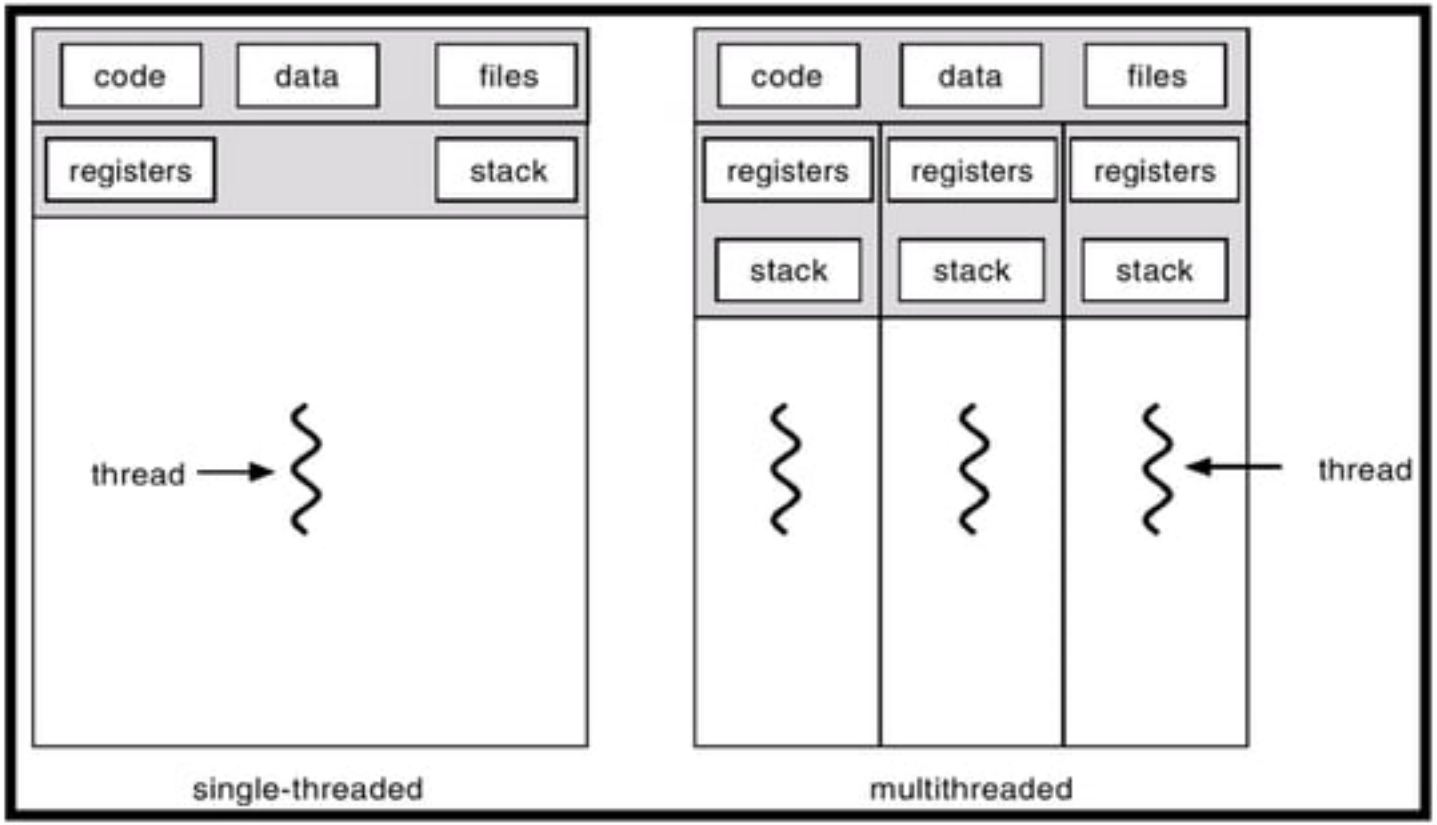
Day 3

Threads

What is Thread?

- ✓ Threads, like processes, are a mechanism to allow a program to do more than one thing at a time
- ✓ As with processes, threads appear to run concurrently
- ✓ Conceptually, a thread exists within a process
- ✓ Threads are a finer-grained unit of execution than processes
- ✓ That 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.

Single & Multi-threaded Processes



Advantages

- ✓ Takes less time to create a new thread in an existing process than to create a brand new process
- ✓ Switching between threads is faster than a normal context switch
- ✓ Threads enhance efficiency in communication between different executing programs
- ✓ No kernel involved

pthread API's

- ✓ GNU/Linux implements the POSIX standard thread API (known as *pthreads*).
- ✓ All thread functions and data types are declared in the header file <pthread.h>.
- ✓ The pthread functions are not included in the standard C library
- ✓ Instead, they are in **libpthread**, so you should add **-lpthread** to the command line when you link your program.

Thread creation

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

- ✓ 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

Thread 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.

How To Compile

- ✓ Use the following command to compile the programs using thread libraries

```
# gcc -o <output> <inputfile.c>  -lpthread
```

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.

Thread Joining

- ✓ The pthread_join() function waits for the thread specified by thread to terminate
- ✓ The function pthread_join, which takes two arguments:
  - Thread ID of the thread to wait for
  - 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.

Thread 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**.

Joinable & Detached threads

- ✓ 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.

Thread 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.

Synchronization

Why Synchronization

✓ Programming with threads is very tricky because most threaded programs are concurrent programs

✓ In particular, there's no way to know when the system will schedule one thread to run and when it will run another

✓ One thread might run for a very long time, or the system might switch among threads very quickly

Why Synchronization

✓Debugging a threaded program is difficult because you cannot always easily reproduce the behavior that caused the problem.

✓You might run the program once and have everything work fine; the next time you run it, it might crash.

✓There's no way to make the system schedule the threads exactly the same way it did before.

Race conditions

✓The ultimate cause of most bugs involving threads is that the threads are accessing the same data.

✓So the powerful aspects of threads can become a danger.

✓ If one thread is only partway through updating a data structure when another thread accesses the same data structure, it's a problem.

✓These bugs are called ***race conditions***; the threads are racing one another to change the same data structure.

The Problem

✓Now suppose that two threads happen to finish a job at about the same time, but only one job remains in the queue.

✓The first thread checks whether **job_queue** is null; finding that it isn't, the thread enters the loop and stores the pointer to the job object in **next_job**.

✓ At this point, Linux happens to interrupt the first thread and schedules the second.

✓The second thread also checks **job_queue** and finding it non-null, also assigns the same job pointer to **next_job**.

✓ By unfortunate coincidence, we now have two threads executing the same job.

Solution - Atomic Operations

✓To eliminate race conditions, you need a way to make operations ***atomic***.

✓An atomic operation is indivisible and uninterruptible; once the operation starts, it will not be paused or interrupted until it completes, and no other operation will take place mean while.

✓In this particular example, you want to check **job_queue**; if it's not empty, remove the first job, all as a single atomic operation.

Mutexes

✓The solution to the job queue race condition problem is to let only one thread access the queue of jobs at a time.

✓GNU/Linux provides ***mutexes***, short for **MUTual EXclusion locks.**

✓A ***mutex*** is a special lock that only one thread may lock at a time.

✓ If a thread locks a mutex and then a second thread also tries to lock the same mutex, the second thread is blocked, or put on hold.

✓Only when the first thread unlocks the mutex is the second thread unblocked—allowed to resume execution.

How To Create Mutex

✓To create a mutex, create a variable of type **pthread_mutex_t** and pass a pointer to it to **pthread_mutex_init.**

✓The second argument to **pthread_mutex_init** is a pointer to a mutex attribute object, which specifies attributes of the mutex.

Locking & Blocking

✓A thread may attempt to lock a mutex by calling **pthread_mutex_lock** on it.

✓If the mutex was unlocked, it becomes locked and the function returns immediately.

✓If the mutex was locked by another thread, **pthread_mutex_lock** blocks execution and returns only eventually when the mutex is unlocked by the other thread.

✓More than one thread may be blocked on a locked mutex at one time.

✓When the mutex is unlocked, only one of the blocked threads is unblocked and allowed to lock the mutex; the other threads stay blocked.

Unlocking Mutex

✓A call to **pthread_mutex_unlock** unlocks a mutex.

✓This function should always be called from the same thread that locked the mutex.

Semaphores for Threads

✓A semaphore is a **counter** that can be used to synchronize multiple threads.

✓As with a mutex, GNU/Linux guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition.

✓Each semaphore has a counter value, which is a non-negative integer.

Two basic operations

✓A ***wait*** operation decrements the value of the semaphore by 1. If the value is already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread). When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns.

✓A ***post*** operation increments the value of the semaphore by 1. If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero).

Semaphore Variable

✓A semaphore is represented by a **sem_t** variable.

✓Before using it, you must initialize it using the **sem_init** function, passing a pointer to the **sem_t** variable.

✓The second parameter should be zero, and the third parameter is the semaphore's initial value.

✓If you no longer need a semaphore, it's good to de-allocate it with **sem_destroy**.

Wait & Post operations

- ✓To wait on a semaphore, use **sem_wait**.
- ✓To post to a semaphore, use **sem_post**.

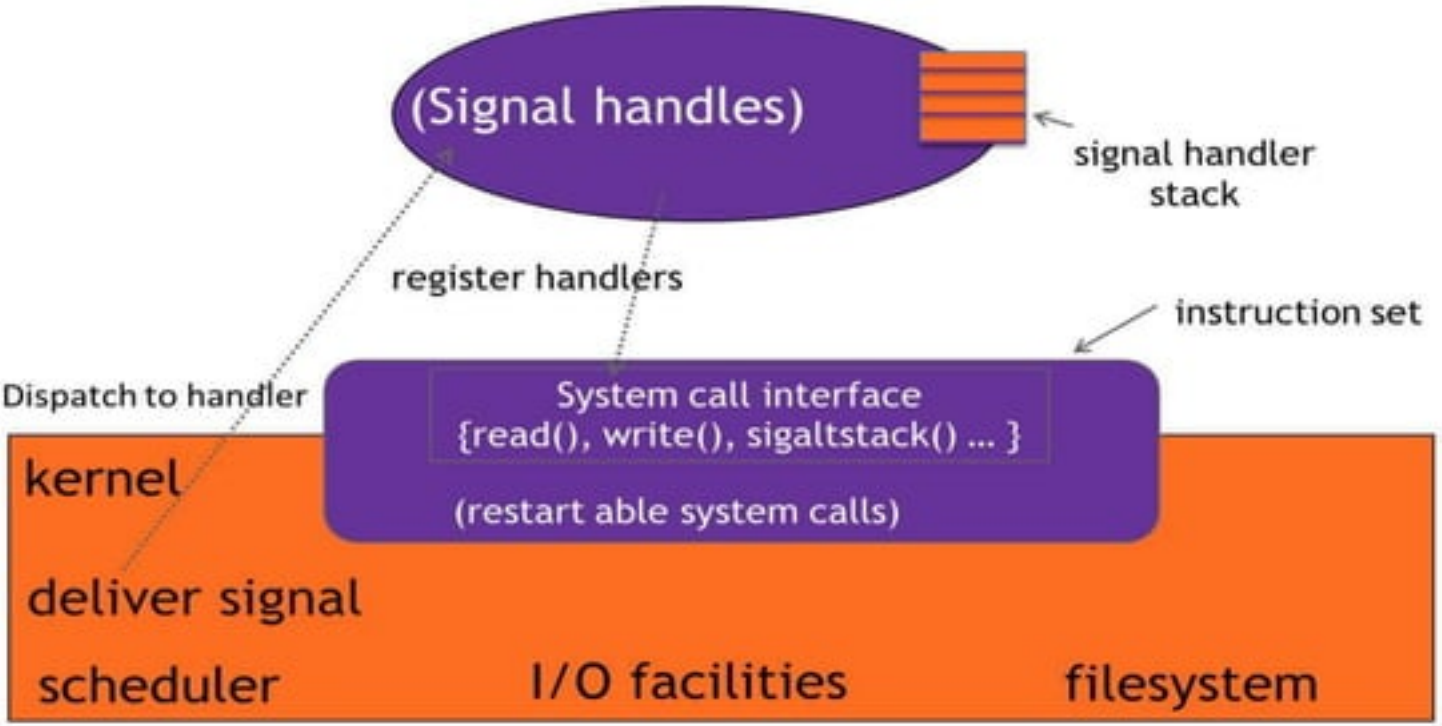
Signals

Introduction

- ✓ Signals are used to notify a process or thread of a particular event.
- ✓ Signals are mechanisms for communicating with and manipulating processes in Linux.
- ✓ Signals make the process aware that something has happened in the system, and the target process should perform some predefined set of actions to keep the system running smoothly.
- ✓ The actions may range from 'self termination' to 'clean-up'.

Signals - Virtual Machine Model

Process X



EMERTXE

Identifying Signals

- ✓ Each signal in Linux has got a name starting with 'SIG' and a number associated with it.
- ✓ For example Signal 'SIGSEGV' has got a number 11.
- ✓ This is defined in /usr/include/bits/signum.h

Where Signals come from?

✓The kernel sends signals to processes in response to specific conditions. For instance, any of these Signals may be sent to a process that attempts to perform an illegal operation :

- SIGBUS (bus error),
- SIGSEGV (segmentation violation),

✓A Process may also send a Signal to another Process.

✓A Process may also send a Signal to itself

How Process Responds to Signals?

- ✓ When a Process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code.
- ✓ For all possible signals, the system defines a default disposition or action to take when a signal occurs. There are four possible default dispositions:
  - Exit: Forces the process to exit.
  - Core: Forces the process to exit and create a core file.
  - Stop: Stops the process.
  - Ignore: Ignores the signal

How To Set Behaviour?

✓The ***sigaction*** function can be used to set a signal disposition whose prototype is:

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

✓The first parameter is the signal number.

✓The next two parameters are pointers to ***sigaction*** structures

✓The first of these contains the desired disposition for that signal number, while the second receives the previous disposition

Note

✓A signal handler should perform the minimum work necessary to respond to the signal, and then return control to the main program (or terminate the program).

✓In most cases, this consists simply of recording the fact that a signal occurred.

✓The main program then checks periodically whether a signal has occurred and reacts accordingly.

Signals and Interrupts

- ✓ Signals can be described as software interrupts.
- ✓ The concept of 'signals' and 'signals handling' is analogous to that of the 'interrupt' handling done by a microprocessor.
- ✓ When a signal is sent to a process or thread, a signal handler may be entered (depending on the current disposition of the signal), which is similar to the system entering an interrupt handler as the result of receiving an interrupt.

Process Termination

✓Normally, a process terminates in one of two ways. Either the executing program calls the exit function, or the program's main function returns.

✓Each process has an exit code: a number that the process returns to its parent. The exit code is the argument passed to the exit function, or the value returned from main.

✓A process may also terminate abnormally, in response to a signal. For instance, the SIGBUS, SIGSEGV, and SIGFPE signals mentioned previously cause the process to terminate.

Stay connected

About us: Emertxe is India's one of the top IT finishing schools & self learning kits provider. Our primary focus is on Embedded with diversification focus on Java, Oracle and Android areas

Emertxe Information Technologies,
No-1, 9th Cross, 5th Main,
Jayamahal Extension,
Bangalore, Karnataka 560046
T: +91 80 6562 9666
E: training@emertxe.com

https://www.facebook.com/Emertxe

https://twitter.com/EmertxeTweet

https://www.slideshare.net/EmertxeSlides

EMERTXE