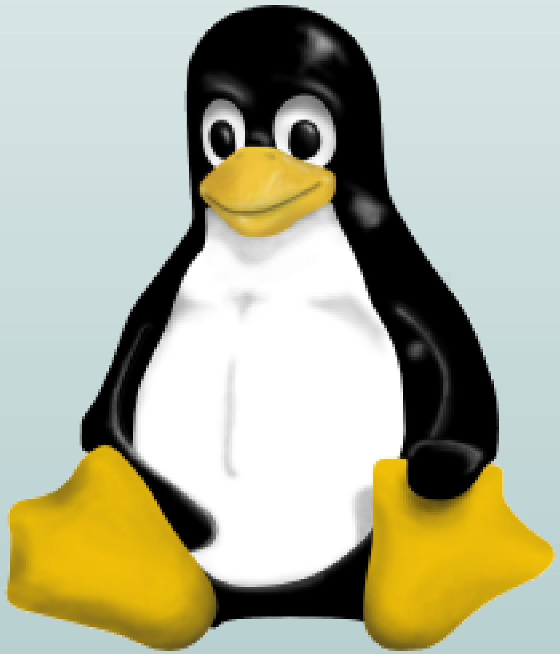
Linux Kernel Hacking Free Course, 3rd edition

D. P. Bovet, M. Cesati
University of Rome “Tor Vergata”

An introduction to Linux



January 18, 2006

What is Linux

What is Linux

- Linux is a POSIX-compliant kernel, although it is not a full Unix-like operating system because it does not include all the Unix applications, such as filesystem utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers, and so on

What is Linux

- Linux is a POSIX-compliant kernel, although it is not a full Unix-like operating system because it does not include all the Unix applications, such as filesystem utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers, and so on
- The GNU project started in 1984, and in particular the gcc compiler/linker has played a crucial role in the development of Linux

What is Linux

- Linux is a POSIX-compliant kernel, although it is not a full Unix-like operating system because it does not include all the Unix applications, such as filesystem utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers, and so on

- The GNU project started in 1984, and in particular the gcc compiler/linker has played a crucial role in the development of Linux

- from the Open Software Foundation website: *Variants of the GNU operating system, which use the kernel Linux, are now widely used; though these systems are often referred to as Linux, they are more accurately called GNU/Linux system*

Privilege levels

Privilege levels

- Modern computers can run in at least two different modes or *privilege levels*

Privilege levels

- Modern computers can run in at least two different modes or *privilege levels*
- This hardware feature has been introduced many years ago to protect the operating system (OS) from faulty programs and to forbid users to access some critical I/O devices such as disks

Privilege levels

- Modern computers can run in at least two different modes or *privilege levels*

- This hardware feature has been introduced many years ago to protect the operating system (OS) from faulty programs and to forbid users to access some critical I/O devices such as disks

- In the IA-32 computers, the 16-bit Code Segment register contains 2 bits that can encode up to 4 different privilege levels: level 0 (most privileged), level 1, level 2, level 3 (less privileged)

What is a kernel

What is a kernel

- The *kernel* of an OS consists of the set of programs that run in a privileged level (for IA-32 computers, a level smaller than 3)

What is a kernel

- The *kernel* of an OS consists of the set of programs that run in a privileged level (for IA-32 computers, a level smaller than 3)

- Some kernels such as the kernel of Windows NT use privilege level 0 for the basic functions and privilege levels 1 and 2 for the I/O drivers

What is a kernel

- The *kernel* of an OS consists of the set of programs that run in a privileged level (for IA-32 computers, a level smaller than 3)

- Some kernels such as the kernel of Windows NT use privilege level 0 for the basic functions and privilege levels 1 and 2 for the I/O drivers

- Linux uses only 2 privilege levels called *User Mode* and *Kernel Mode*

Entering and leaving Kernel Mode

Entering and leaving Kernel Mode

- A User Mode program enters in kernel mode by issuing a special `int` instruction: in Linux, interrupt `0x80` is reserved to implement system calls

Entering and leaving Kernel Mode

- A User Mode program enters in kernel mode by issuing a special `int` instruction: in Linux, interrupt `0x80` is reserved to implement system calls

- Many I/O devices issue interrupts to signal the end of an I/O operation: each of these interrupts puts the CPU into Kernel Mode

Entering and leaving Kernel Mode

- A User Mode program enters in kernel mode by issuing a special `int` instruction: in Linux, interrupt `0x80` is reserved to implement system calls

- Many I/O devices issue interrupts to signal the end of an I/O operation: each of these interrupts puts the CPU into Kernel Mode

- The CPU issues special interrupts called *exceptions* to signal the occurrence of abnormal conditions: overflow, page fault, etc.

Entering and leaving Kernel Mode

- A User Mode program enters in kernel mode by issuing a special `int` instruction: in Linux, interrupt `0x80` is reserved to implement system calls

- Many I/O devices issue interrupts to signal the end of an I/O operation: each of these interrupts puts the CPU into Kernel Mode

- The CPU issues special interrupts called *exceptions* to signal the occurrence of abnormal conditions: overflow, page fault, etc.

- A program in Kernel Mode can put the CPU back in User Mode by executing the `iret` (Interrupt Return) instruction

The kernel program

The kernel program

- The kernel is a huge program, which must be compiled and linked before being loaded in RAM

The kernel program

- The kernel is a huge program, which must be compiled and linked before being loaded in RAM
- Contrary to ordinary programs (sequential programs), the kernel has the following main characteristics:

The kernel program

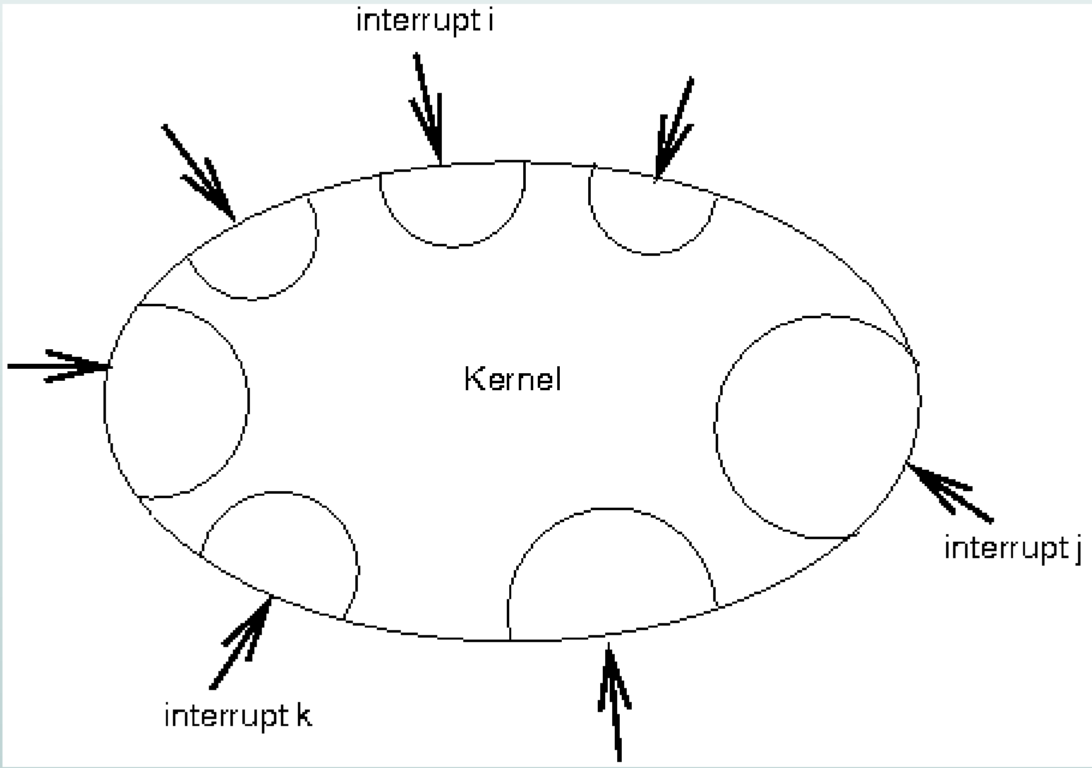
- The kernel is a huge program, which must be compiled and linked before being loaded in RAM
- Contrary to ordinary programs (sequential programs), the kernel has the following main characteristics:
  - It does not have a single entry point; a different entry point must be provided for every type of interrupt recognized by the kernel

The kernel program

- The kernel is a huge program, which must be compiled and linked before being loaded in RAM
- Contrary to ordinary programs (sequential programs), the kernel has the following main characteristics:
  - It does not have a single entry point; a different entry point must be provided for every type of interrupt recognized by the kernel
  - The kernel image produced by the gcc linker cannot be loaded as any other executable file, simply because the loader is not available when booting the system: a more rudimentary technique based on bootstrapping must be used

The kernel program

The kernel program



Which programming language?

Which programming language?

- As all Unix-like OSs, Linux is coded mostly in C

Which programming language?

- As all Unix-like OSs, Linux is coded mostly in C
- C is a high-level programming language developed in 1973 by K. Thompson and D. Ritchie to rewrite a previous version of Unix coded in Assembly

Which programming language?

- As all Unix-like OSs, Linux is coded mostly in C
- C is a high-level programming language developed in 1973 by K. Thompson and D. Ritchie to rewrite a previous version of Unix coded in Assembly
- Some architecture-dependent programs and a few small critical functions are coded in Assembly (roughly 10% of the code)

Which programming language?

- As all Unix-like OSs, Linux is coded mostly in C
- C is a high-level programming language developed in 1973 by K. Thompson and D. Ritchie to rewrite a previous version of Unix coded in Assembly
- Some architecture-dependent programs and a few small critical functions are coded in Assembly (roughly 10% of the code)
- To improve portability to new hardware platforms, the architecture-dependent code is placed in the `linux/arch` directory

Which programming language?

- As all Unix-like OSs, Linux is coded mostly in C
- C is a high-level programming language developed in 1973 by K. Thompson and D. Ritchie to rewrite a previous version of Unix coded in Assembly
- Some architecture-dependent programs and a few small critical functions are coded in Assembly (roughly 10% of the code)
- To improve portability to new hardware platforms, the architecture-dependent code is placed in the `linux/arch` directory
- A symbolic link called `linux/include/asm` identifies all the architecture-dependent header files for a given hardware platform

Non-standard extensions of C

Non-standard extensions of C

- The gnu C compiler includes several non-standard extensions of C exploited by Linux:

Non-standard extensions of C

- The gnu C compiler includes several non-standard extensions of C exploited by Linux:
  - the `inline` function qualifier

Non-standard extensions of C

- The gnu C compiler includes several non-standard extensions of C exploited by Linux:
  - the `inline` function qualifier
  - the 64-bit `long long` data type

Non-standard extensions of C

- The gnu C compiler includes several non-standard extensions of C exploited by Linux:
  - the `inline` function qualifier
  - the 64-bit `long long` data type
  - the `__attribute__((regparm(3))` function qualifier to pass up to 3 integer parameters using registers instead of the stack (macro `FASTCALL(x)`)

Which kind of Assembly?

Which kind of Assembly?

- Very few Assembly files: `linux/arch/i386/kernel/entry.S`, `linux/arch/i386/kernel/head.S`, ...

Which kind of Assembly?

- Very few Assembly files: `linux/arch/i386/kernel/entry.S`, `linux/arch/i386/kernel/head.S`, ...

- Most of the assembly code is Inline Assembly, that is assembly code embedded in a C function by means of the `asm()` primitive

Which kind of Assembly?

- Very few Assembly files: `linux/arch/i386/kernel/entry.S`, `linux/arch/i386/kernel/head.S`, ...
- Most of the assembly code is Inline Assembly, that is assembly code embedded in a C function by means of the `asm()` primitive
- To improve performances, Extended Inline Assembly is used instead of Inline Assembly

Which kind of Assembly?

- Very few Assembly files: `linux/arch/i386/kernel/entry.S`, `linux/arch/i386/kernel/head.S`, ...
- Most of the assembly code is Inline Assembly, that is assembly code embedded in a C function by means of the `asm()` primitive
- To improve performances, Extended Inline Assembly is used instead of Inline Assembly
- The syntax of Extended Inline Assembly is not obvious

An example of Extended Inline Assembly code

```
static inline int _raw_spin_trylock(spinlock_t *lock)
{
        char oldval;
        __asm__ __volatile__(
                "xchgb %b0,%1"
                :"=q" (oldval), "=m" (lock->slock)
                :"0" (0) : "memory");
        return oldval > 0;
}
```

Expanding the example of Extended Inline Assembly code

```
        xorl %eax, %eax
#APP
        xchgb %al,-4(%ebp)  ; lock variable stored in -4(%ebp)
#NO_APP
        testb %al, %al
        setg %al
        andl $255, %eax
```

The addresses of Linux symbols

The addresses of Linux symbols

- Typically, one fourth of the 4 GB address range is reserved to kernel addresses: all addresses from `0xc0000000` to `0xffffffff` are used by the linker for symbols of the kernel code

The addresses of Linux symbols

- Typically, one fourth of the 4 GB address range is reserved to kernel addresses: all addresses from `0xc0000000` to `0xffffffff` are used by the linker for symbols of the kernel code

- The lower addresses from `0x00000000` to `0xbfffffff` are used by the linker to link User Mode programs

The addresses of Linux symbols

- Typically, one fourth of the 4 GB address range is reserved to kernel addresses: all addresses from `0xc0000000` to `0xffffffff` are used by the linker for symbols of the kernel code

- The lower addresses from `0x00000000` to `0xbfffffff` are used by the linker to link User Mode programs

- When linking the kernel, gcc creates a `System.map` file which lists the addresses assigned to all global kernel symbols. This file is used by debuggers and kernel profilers

The size of Linux

The size of Linux

- Linux becomes bigger and bigger, mostly to support new drivers and to satisfy the requirements of enterprise systems

The size of Linux

- Linux becomes bigger and bigger, mostly to support new drivers and to satisfy the requirements of enterprise systems

- Linux 2.2.14 (2000) includes roughly 2 million lines of code

The size of Linux

- Linux becomes bigger and bigger, mostly to support new drivers and to satisfy the requirements of enterprise systems
- Linux 2.2.14 (2000) includes roughly 2 million lines of code
- Linux 2.4.18 (2002) includes roughly 4 million lines of code

The size of Linux

- Linux becomes bigger and bigger, mostly to support new drivers and to satisfy the requirements of enterprise systems
- Linux 2.2.14 (2000) includes roughly 2 million lines of code
- Linux 2.4.18 (2002) includes roughly 4 million lines of code
- Linux 2.6.11 (2005) includes roughly 6 million lines of code

Goals of Linux

Goals of Linux

World domination!

Goals of Linux (beside World domination!)

The Linux kernel must provide an execution environment for the users' applications

Goals of Linux (beside World domination!)

The Linux kernel must provide an execution environment for the users' applications in such a way to:

- Allow concurrent execution of many applications at the same time (time sharing)

Goals of Linux (beside World domination!)

The Linux kernel must provide an execution environment for the users' applications in such a way to:

- Allow concurrent execution of many applications at the same time (time sharing)
- Arbitrate access to system resources

Goals of Linux (beside World domination!)

The Linux kernel must provide an execution environment for the users' applications in such a way to:

- Allow concurrent execution of many applications at the same time (time sharing)
- Arbitrate access to system resources
- Grant protection from misbehaving programs

Goals of Linux (beside World domination!)

The Linux kernel must provide an execution environment for the users' applications in such a way to:

- Allow concurrent execution of many applications at the same time (time sharing)
- Arbitrate access to system resources
- Grant protection from misbehaving programs
- Have a small overhead

Goals of Linux (beside World domination!)

The Linux kernel must provide an execution environment for the users' applications in such a way to:

- Allow concurrent execution of many applications at the same time (time sharing)
- Arbitrate access to system resources
- Grant protection from misbehaving programs
- Have a small overhead

A process is traditionally defined as an instance of a program in execution (or, shortly, an execution context)

Process vs thread vs lightweight process vs kernel thread

There is no overall agreement over the meaning of these terms

Process vs thread vs lightweight process vs kernel thread

There is no overall agreement over the meaning of these terms

We'll stick to the following definitions:

- A process is an execution context that can be independently scheduled for execution by the kernel

Process vs thread vs lightweight process vs kernel thread

There is no overall agreement over the meaning of these terms

We'll stick to the following definitions:

- A process is an execution context that can be independently scheduled for execution by the kernel
- A thread is an execution flow inside a multithreaded application

Process vs thread vs lightweight process vs kernel thread

There is no overall agreement over the meaning of these terms

We'll stick to the following definitions:

- A process is an execution context that can be independently scheduled for execution by the kernel
- A thread is an execution flow inside a multithreaded application
- A lightweight process is a process that shares with other processes some critical resources (like physical memory, open files, signal handlers, ...)

Process vs thread vs lightweight process vs kernel thread

There is no overall agreement over the meaning of these terms

We'll stick to the following definitions:

- A process is an execution context that can be independently scheduled for execution by the kernel
- A thread is an execution flow inside a multithreaded application
- A lightweight process is a process that shares with other processes some critical resources (like physical memory, open files, signal handlers, ...)
- A kernel thread is a process that runs only in Kernel Mode, that is, it never executes code of users' applications

Process descriptor

The bunch of information kept by the kernel for each process is rooted in a table called process descriptor

Process descriptor

The bunch of information kept by the kernel for each process is rooted in a table called process descriptor

The process descriptor is a structure of type `task_struct` (or, shortly, `task_t`) composed by more than 100 fields (see `linux/include/linux/sched.h`)

Process descriptor

The bunch of information kept by the kernel for each process is rooted in a table called process descriptor

The process descriptor is a structure of type `task_struct` (or, shortly, `task_t`) composed by more than 100 fields (see `linux/include/linux/sched.h`)

Many of such fields are pointers to additional tables describing, for instance, the memory regions, the open files, the signal handlers, . . .

Process descriptor

The bunch of information kept by the kernel for each process is rooted in a table called process descriptor

The process descriptor is a structure of type `task_struct` (or, shortly, `task_t`) composed by more than 100 fields (see `linux/include/linux/sched.h`)

Many of such fields are pointers to additional tables describing, for instance, the memory regions, the open files, the signal handlers, . . .

The `current` macro yields the address of the descriptor relative to the process currently executed by the CPU

States of a process

The current state of each process is stored in the corresponding descriptor.

States of a process

The current state of each process is stored in the corresponding descriptor. The most important process states are the following:

- In execution on a CPU

States of a process

The current state of each process is stored in the corresponding descriptor. The most important process states are the following:

- In execution on a CPU
- Runnable, but currently not in execution

States of a process

The current state of each process is stored in the corresponding descriptor.
The most important process states are the following:

- In execution on a CPU
- Runnable, but currently not in execution
- Waiting for an event or a signal
- Waiting for an event

States of a process

The current state of each process is stored in the corresponding descriptor. The most important process states are the following:

- In execution on a CPU
- Runnable, but currently not in execution
- Waiting for an event or a signal
- Waiting for an event
- Suspended (frozen)

States of a process

The current state of each process is stored in the corresponding descriptor. The most important process states are the following:

- In execution on a CPU
- Runnable, but currently not in execution
- Waiting for an event or a signal
- Waiting for an event
- Suspended (frozen)
- Terminated

States of a process

The current state of each process is stored in the corresponding descriptor. The most important process states are the following:

- In execution on a CPU
- Runnable, but currently not in execution

`TASK_RUNNING`

- Waiting for an event or a signal
- Waiting for an event
- Suspended (frozen)
- Terminated

States of a process

The current state of each process is stored in the corresponding descriptor. The most important process states are the following:

- In execution on a CPU
- Runnable, but currently not in execution
- Waiting for an event or a signal
- Waiting for an event
- Suspended (frozen)
- Terminated

`TASK_RUNNING`

`TASK_INTERRUPTIBLE`

States of a process

The current state of each process is stored in the corresponding descriptor. The most important process states are the following:

• In execution on a CPU	`TASK_RUNNING`
• Runnable, but currently not in execution	
• Waiting for an event or a signal	`TASK_INTERRUPTIBLE`
• Waiting for an event	`TASK_UNINTERRUPTIBLE`
• Suspended (frozen)	
• Terminated	

States of a process

The current state of each process is stored in the corresponding descriptor.
The most important process states are the following:

• In execution on a CPU • Runnable, but currently not in execution	TASK_RUNNING
• Waiting for an event or a signal	TASK_INTERRUPTIBLE
• Waiting for an event	TASK_UNINTERRUPTIBLE
• Suspended (frozen)	TASK_STOPPED
• Terminated	

States of a process

The current state of each process is stored in the corresponding descriptor.
The most important process states are the following:

• In execution on a CPU • Runnable, but currently not in execution	`TASK_RUNNING`
• Waiting for an event or a signal	`TASK_INTERRUPTIBLE`
• Waiting for an event	`TASK_UNINTERRUPTIBLE`
• Suspended (frozen)	`TASK_STOPPED`
• Terminated	`EXIT_ZOMBIE`, `EXIT_DEAD`

States of a process

The current state of each process is stored in the corresponding descriptor. The most important process states are the following:

• In execution on a CPU • Runnable, but currently not in execution	`TASK_RUNNING`
• Waiting for an event or a signal	`TASK_INTERRUPTIBLE`
• Waiting for an event	`TASK_UNINTERRUPTIBLE`
• Suspended (frozen)	`TASK_STOPPED`
• Terminated	`EXIT_ZOMBIE`, `EXIT_DEAD`

Processes in state `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` are equivalently said to be *sleeping* or *blocked*

The double life of processes

Actually, each process has two distinct lives:

1. when the CPU is in User Mode, the process runs code of the user's application or of a system library

2. when the CPU is in Kernel Mode, the process executes code of the kernel

The double life of processes

Actually, each process has two distinct lives:

1. when the CPU is in User Mode, the process runs code of the user's application or of a system library

2. when the CPU is in Kernel Mode, the process executes code of the kernel

Switching from User Mode to Kernel Mode, or vice versa, does not necessarily change the current process!

The double life of processes

Actually, each process has two distinct lives:

1. when the CPU is in User Mode, the process runs code of the user's application or of a system library

2. when the CPU is in Kernel Mode, the process executes code of the kernel

Switching from User Mode to Kernel Mode, or vice versa, does not necessarily change the current process!

Often we say something like: "*the kernel is doing this and that...*"; however, we must never forget that there is always a current process in execution!

Process address space

Each process owns a so-called linear address space: basically, it is the set of linear addresses that can be legally referenced while running in User Mode

Process address space

Each process owns a so-called linear address space: basically, it is the set of linear addresses that can be legally referenced while running in User Mode

Each process has its own mapping between linear addresses and physical addresses stored in the process's page tables

Process address space

Each process owns a so-called linear address space: basically, it is the set of linear addresses that can be legally referenced while running in User Mode

Each process has its own mapping between linear addresses and physical addresses stored in the process's page tables

Therefore, different processes may use the same linear address with different meanings; for instance, the linear address `0x800210f0` could be mapped to the physical address `0x0000d000` for some process and to the physical address `0x038af000` for another process

Process address space

Each process owns a so-called linear address space: basically, it is the set of linear addresses that can be legally referenced while running in User Mode

Each process has its own mapping between linear addresses and physical addresses stored in the process's page tables

Therefore, different processes may use the same linear address with different meanings; for instance, the linear address `0x800210f0` could be mapped to the physical address `0x0000d000` for some process and to the physical address `0x038af000` for another process

When running in Kernel Mode every process makes use of linear addresses in the fourth gigabyte (above `0xc0000000`); the mapping of these "kernel" linear addresses is identical for all processes

Process context and interrupt context

As we have seen, the CPU switches to Kernel Mode when either

- a hardware device raises an interrupt, or
- an application triggers an exception or invokes a system call

Process context and interrupt context

As we have seen, the CPU switches to Kernel Mode when either

- a hardware device raises an interrupt, or
- an application triggers an exception or invokes a system call

In the first case, we say that the kernel is executing in *interrupt context*; in the second case, we say that the kernel is executing in *process context*

Process context and interrupt context

As we have seen, the CPU switches to Kernel Mode when either

- a hardware device raises an interrupt, or
- an application triggers an exception or invokes a system call

In the first case, we say that the kernel is executing in *interrupt context*; in the second case, we say that the kernel is executing in *process context*

While in interrupt context, the kernel cannot make any assumption on the process that is currently in execution. In particular, the kernel cannot

- perform a process switch (thus, it cannot start blocking operations)
- reference memory by means of linear addresses below the fourth gigabyte

Asynchronous functions

The functions of the kernel that can be executed in interrupt context are also said to be *asynchronous*

Asynchronous functions

The functions of the kernel that can be executed in interrupt context are also said to be *asynchronous*

There are several types of asynchronous functions:

- interrupt handlers: each device driver defines an interrupt handler to handle the interrupts coming from the hardware device

Asynchronous functions

The functions of the kernel that can be executed in interrupt context are also said to be *asynchronous*

There are several types of asynchronous functions:

- interrupt handlers: each device driver defines an interrupt handler to handle the interrupts coming from the hardware device
- deferrable functions (softirqs and tasklets): usually activated by interrupt handlers, they perform lower priority jobs before returning in process context

Asynchronous functions

The functions of the kernel that can be executed in interrupt context are also said to be *asynchronous*

There are several types of asynchronous functions:

- interrupt handlers: each device driver defines an interrupt handler to handle the interrupts coming from the hardware device
- deferrable functions (softirqs and tasklets): usually activated by interrupt handlers, they perform lower priority jobs before returning in process context
- timer functions: timers allow to execute arbitrary functions after predefined delays

Scheduling processes

Linux implements a time sharing system: a slice of the CPU time (the so-called time quantum) is assigned to each runnable process

Scheduling processes

Linux implements a time sharing system: a slice of the CPU time (the so-called time quantum) is assigned to each runnable process

The kernel monitors the execution time of the currently running process; when its time quantum expires, the kernel may replace it with another runnable process

Scheduling processes

Linux implements a time sharing system: a slice of the CPU time (the so-called time quantum) is assigned to each runnable process

The kernel monitors the execution time of the currently running process; when its time quantum expires, the kernel may replace it with another runnable process

The scheduler is the kernel program that selects the ‘best” process to run; its entry point is the `schedule()` function (see `linux/kernel/sched.c`)

Scheduling processes

Linux implements a time sharing system: a slice of the CPU time (the so-called time quantum) is assigned to each runnable process

The kernel monitors the execution time of the currently running process; when its time quantum expires, the kernel may replace it with another runnable process

The scheduler is the kernel program that selects the 'best" process to run; its entry point is the `schedule()` function (see `linux/kernel/sched.c`)

Actually, some processes (the so-called real-time processes) are not handled with the time sharing policy, but with a priority-based FIFO or Round-Robin policy

Preemption

The term *preemption* denotes the ability of the kernel to replace the currently running process with another process having higher priority

Preemption

The term *preemption* denotes the ability of the kernel to replace the currently running process with another process having higher priority

All modern, general-purpose OSs sport preemptive multitasking, that is, they have preemptible processes: essentially, when a process runs in User Mode, it can be replaced at any time by the kernel without regards to what it is doing

Preemption

The term *preemption* denotes the ability of the kernel to replace the currently running process with another process having higher priority

All modern, general-purpose OSs sport preemptive multitasking, that is, they have preemptible processes: essentially, when a process runs in User Mode, it can be replaced at any time by the kernel without regards to what it is doing

Some OSs, however, are also kernel preemptive: a process can be replaced even when it executes the code of an exception handler or system call handler (but never when it executes the code of an interrupt handler!)

Preemption

The term *preemption* denotes the ability of the kernel to replace the currently running process with another process having higher priority

All modern, general-purpose OSs sport preemptive multitasking, that is, they have preemptible processes: essentially, when a process runs in User Mode, it can be replaced at any time by the kernel without regards to what it is doing

Some OSs, however, are also kernel preemptive: a process can be replaced even when it executes the code of an exception handler or system call handler (but never when it executes the code of an interrupt handler!)

Starting from version 2.6, kernel preemption can be enabled or disabled when compiling the kernel

Virtual memory

The virtual memory subsystem is a core component that allows the kernel to:

- allocate *on demand* the physical memory required by the users' applications

Virtual memory

The virtual memory subsystem is a core component that allows the kernel to:

- allocate *on demand* the physical memory required by the users' applications
- reclaim physical memory from the users' applications (in case of memory shortage)

Virtual memory

The virtual memory subsystem is a core component that allows the kernel to:

- allocate *on demand* the physical memory required by the users' applications
- reclaim physical memory from the users' applications (in case of memory shortage)
- temporarily put the data of the slow disk devices in RAM (disk caches)

Virtual memory

The virtual memory subsystem is a core component that allows the kernel to:

- allocate *on demand* the physical memory required by the users' applications
- reclaim physical memory from the users' applications (in case of memory shortage)
- temporarily put the data of the slow disk devices in RAM (disk caches)

The virtual memory subsystem works in chunks of RAM called *pages*

In IA-32, each page is 4096 bytes long

Support to multiprocessor systems

Several kernel components, such as the scheduler and the memory allocator, behave differently according to the type of multiprocessor system

Support to multiprocessor systems

Several kernel components, such as the scheduler and the memory allocator, behave differently according to the type of multiprocessor system:

- Intel CPUs with Hyper-Threading Technology (several logical processing units on the same chip)

Support to multiprocessor systems

Several kernel components, such as the scheduler and the memory allocator, behave differently according to the type of multiprocessor system:

- Intel CPUs with Hyper-Threading Technology (several logical processing units on the same chip)
- Multicore chips (several physical processing units, each with its hardware caches, on the same chip)

Support to multiprocessor systems

Several kernel components, such as the scheduler and the memory allocator, behave differently according to the type of multiprocessor system:

- Intel CPUs with Hyper-Threading Technology (several logical processing units on the same chip)
- Multicore chips (several physical processing units, each with its hardware caches, on the same chip)
- Non-uniform Memory Access (NUMA) systems (several CPUs, each having its own local physical memory)

Support to multiprocessor systems

Several kernel components, such as the scheduler and the memory allocator, behave differently according to the type of multiprocessor system:

- Intel CPUs with Hyper-Threading Technology (several logical processing units on the same chip)
- Multicore chips (several physical processing units, each with its hardware caches, on the same chip)
- Non-uniform Memory Access (NUMA) systems (several CPUs, each having its own local physical memory)

The bottom line is: at any given instant there are several processes in execution

Support to multiprocessor systems

Several kernel components, such as the scheduler and the memory allocator, behave differently according to the type of multiprocessor system:

- Intel CPUs with Hyper-Threading Technology (several logical processing units on the same chip)
- Multicore chips (several physical processing units, each with its hardware caches, on the same chip)
- Non-uniform Memory Access (NUMA) systems (several CPUs, each having its own local physical memory)

The bottom line is: at any given instant there are several processes in execution

Synchronization, or how to make order out of chaos

When programming the kernel it is crucial to forbid execution flows (asynchronous functions, exception and system call handlers) to badly interfere with each other (*race conditions*)

Synchronization, or how to make order out of chaos

When programming the kernel it is crucial to forbid execution flows (asynchronous functions, exception and system call handlers) to badly interfere with each other (*race conditions*)

Even in uniprocessor systems kernel synchronization is a hard necessity: kernel preemption and multiprocessor systems just make things worse

Synchronization, or how to make order out of chaos

When programming the kernel it is crucial to forbid execution flows (asynchronous functions, exception and system call handlers) to badly interfere with each other (*race conditions*)

Even in uniprocessor systems kernel synchronization is a hard necessity: kernel preemption and multiprocessor systems just make things worse

The Linux kernel sports a large number of synchronization primitives: atomic operations, memory barriers, interrupt disabling, deferrable function disabling, per-CPU data, semaphores, read/write semaphores, spin locks, read/write spin locks, Read-Copy-Update (RCU), seqlocks, mutexes (in the forthcoming version 2.6.16), and others

Device driver developer’s easy receipt for synchronization

The large number of synchronization primitives is only due to efficiency reasons: the kernel must keep the time spent while waiting for a resource to a minimum

As a matter of fact, most synchronization primitives have been introduced in order to allow some kernel core components to scale well when Linux is used in large enterprise systems

Device driver developer's easy receipt for synchronization

The large number of synchronization primitives is only due to efficiency reasons: the kernel must keep the time spent while waiting for a resource to a minimum

As a matter of fact, most synchronization primitives have been introduced in order to allow some kernel core components to scale well when Linux is used in large enterprise systems

Actually, device driver developers can just use a handful of primitives:

1. semaphores (or better mutexes, when they will be available)

Device driver developer's easy receipt for synchronization

The large number of synchronization primitives is only due to efficiency reasons: the kernel must keep the time spent while waiting for a resource to a minimum

As a matter of fact, most synchronization primitives have been introduced in order to allow some kernel core components to scale well when Linux is used in large enterprise systems

Actually, device driver developers can just use a handful of primitives:

1. semaphores (or better mutexes, when they will be available)

2. spin locks (optionally coupled with interrupt disabling)

Semaphores

Semaphores can be used to protect any data structure that can be accessed only in process context

Semaphores

Semaphores can be used to protect any data structure that can be accessed only in process context

In other words, semaphores can be thought as primitives aimed to control the accesses to the resources shared among the processes in the system

Semaphores

Semaphores can be used to protect any data structure that can be accessed only in process context

In other words, semaphores can be thought as primitives aimed to control the accesses to the resources shared among the processes in the system

While a process is waiting on a busy semaphore, it is blocked (put to sleep in state `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`) and replaced by another runnable process on the CPU

Semaphores

Semaphores can be used to protect any data structure that can be accessed only in process context

In other words, semaphores can be thought as primitives aimed to control the accesses to the resources shared among the processes in the system

While a process is waiting on a busy semaphore, it is blocked (put to sleep in state `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`) and replaced by another runnable process on the CPU

Basically, a semaphore cannot be used in interrupt context!

Using a semaphore as a MUTEX

To allocate a semaphore to be used as a MUTEX (one process at a time):

```
DECLARE_MUTEX(foo_semaphore);
```

Using a semaphore as a MUTEX

To allocate a semaphore to be used as a MUTEX (one process at a time):

```
DECLARE_MUTEX(foo_semaphore);
```

To acquire the semaphore:

```
down_interruptible(&foo_semaphore);
```

or

```
down(&foo_semaphore);
```

Using a semaphore as a MUTEX

To allocate a semaphore to be used as a MUTEX (one process at a time):

```
DECLARE_MUTEX(foo_semaphore);
```

To acquire the semaphore:

```
down_interruptible(&foo_semaphore);
```
or
```
down(&foo_semaphore);
```

To release the semaphore:

```
up(&foo_semaphore);
```

Spin locks

Spin locks are used to protect data structures that can possibly be accessed in interrupt context

Spin locks

Spin locks are used to protect data structures that can possibly be accessed in interrupt context

A spin lock is a MUTEX implemented by an atomic variable having only two possible values: *locked* and *unlocked*

Spin locks

Spin locks are used to protect data structures that can possibly be accessed in interrupt context

A spin lock is a MUTEX implemented by an atomic variable having only two possible values: *locked* and *unlocked*

When the CPU must acquire a spin lock, it reads the value of the atomic variable and sets it to *locked*; if the variable was already *locked* before the read-and-set operation, then the whole step is repeated

Spin locks

Spin locks are used to protect data structures that can possibly be accessed in interrupt context

A spin lock is a MUTEX implemented by an atomic variable having only two possible values: *locked* and *unlocked*

When the CPU must acquire a spin lock, it reads the value of the atomic variable and sets it to *locked*; if the variable was already *locked* before the read-and-set operation, then the whole step is repeated

Therefore, a process waiting for a spin lock is never blocked! (However, if the kernel is preemptive, the process may be replaced by another runnable process)

Read carefully the instructions before using!

When using spin locks it's easy to cause deadlocks

Read carefully the instructions before using!

When using spin locks it's easy to cause deadlocks

Some important points to remember:

- If the data structure protected by the spin lock is accessed also in interrupt context, we must disable the interrupts before acquiring the spin lock

Read carefully the instructions before using!

When using spin locks it's easy to cause deadlocks

Some important points to remember:

- If the data structure protected by the spin lock is accessed also in interrupt context, we must disable the interrupts before acquiring the spin lock
- The kernel automatically disables the kernel preemption once a spin lock has been acquired

Read carefully the instructions before using!

When using spin locks it's easy to cause deadlocks

Some important points to remember:

- If the data structure protected by the spin lock is accessed also in interrupt context, we must disable the interrupts before acquiring the spin lock
- The kernel automatically disables the kernel preemption once a spin lock has been acquired
- In uniprocessor systems, the atomic variable is not really useful: when compiling for uniprocessor systems, that variable is simply optimized away (but spin lock primitives are still necessary!)

Using a spin lock

To allocate and initialize a spin lock:

```
spinlock_t foo_lock;
spin_lock_init(&foo_lock);
```
[unlocked]

Using a spin lock

To allocate and initialize a spin lock:

```
spinlock_t foo_lock;
spin_lock_init(&foo_lock);
```
[unlocked]

To disable interrupts and acquire the spin lock:

```
spin_lock_irqsave(&foo_lock, flags);
```
[locked]

Using a spin lock

To allocate and initialize a spin lock:

```
spinlock_t foo_lock;
spin_lock_init(&foo_lock);
```
[unlocked]

To disable interrupts and acquire the spin lock:

```
spin_lock_irqsave(&foo_lock, flags);
```
[locked]

To release the spin lock and restore the previous interrupt status:

```
spin_unlock_irqrestore(&foo_lock, flags);
```
[unlocked]