

# Linux Kernel Development

Internal training

Claudio Scordino

Software Engineer, PhD

[claudio.scordino@gmail.com](mailto:claudio.scordino@gmail.com)

October 16, 2015



# Outline

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples



## Kernel



## Kernel drivers

# Unix structure

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

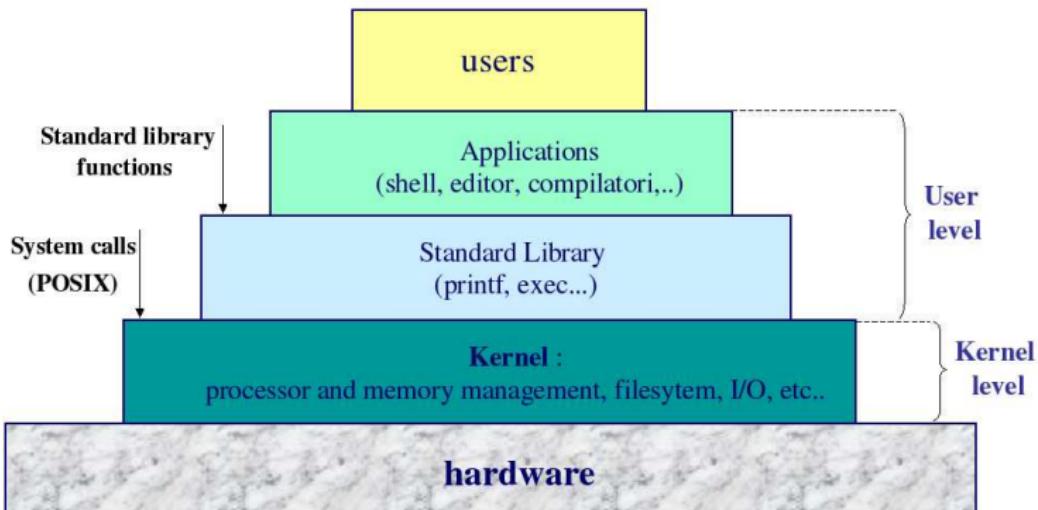
[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)



# Privilege levels

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Modern processors can run in at least two different modes (*privilege levels*)
- Hardware feature introduced to protect the OS from faulty and malicious programs
- Linux uses only the highest and the lowest privilege levels
- The kernel of the OS runs in the highest level (called **Kernel or Supervisor Mode**)
- The applications run in the lowest level (called **User Mode**)
- The CPU switches to Kernel Mode whenever
  - A hardware device raises an interrupt
  - An application triggers an exception or invokes a system call
- Exceptions are synchronous interrupts

# Goals of the kernel

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Provide to applications a standard API
  - Easy to use
  - To guarantee portability among different operating systems
- Optimize and arbitrate use and access of system resources
- Grant data security and protection from misbehaving or buggy programs
- Manage failures

# Kernel's role

## Kernel

[Intro](#)  
[Compiling](#)  
[Code](#)  
[Debugging](#)  
[Linked lists](#)  
[Scheduling](#)  
[Linux filesystem](#)  
[/dev filesystem](#)  
[proc filesystem](#)  
[sys filesystem](#)  
[Syscalls](#)  
[Real-Time](#)  
[Boot time](#)

## Kernel drivers

[Modules](#)  
[Memory](#)  
[Concurrency](#)  
[Char devices](#)  
[I/O](#)  
[Platforms](#)  
[Using sysfs](#)  
[Timers](#)  
[Int. handlers](#)  
[Bottom halves](#)  
[Examples](#)

## 1. Process management

- Scheduling of processes and inter-process communication
- Virtualized processor

## 2. Memory management

- Virtual addressing space
- Virtualized memory

## 3. Filesystems: file abstraction

## 4. Device control: *device drivers* for every peripherals installed on the system (e.g. hard drive, keyboard, floppy drive, etc.)

## 5. Networking: delivering of data packets across processes and network interfaces

# Kernel's role

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)  
[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)  
[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

## 1. Process management

- Scheduling of processes and inter-process communication
- Virtualized processor

## 2. Memory management

- Virtual addressing space
- Virtualized memory

## 3. Filesystems: file abstraction

## 4. Device control: *device drivers* for every peripherals installed on the system (e.g. hard drive, keyboard, floppy drive, etc.)

## 5. Networking: delivering of data packets across processes and network interfaces

# Kernel's role

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

## 1. Process management

- Scheduling of processes and inter-process communication
- Virtualized processor

## 2. Memory management

- Virtual addressing space
- Virtualized memory

## 3. Filesystems: file abstraction

4. Device control: *device drivers* for every peripherals installed on the system (e.g. hard drive, keyboard, floppy drive, etc.)
5. Networking: delivering of data packets across processes and network interfaces

# Kernel's role

## Kernel

[Intro](#)  
[Compiling](#)  
[Code](#)  
[Debugging](#)  
[Linked lists](#)  
[Scheduling](#)  
[Linux filesystem](#)  
[/dev filesystem](#)  
[proc filesystem](#)  
[sys filesystem](#)  
[Syscalls](#)  
[Real-Time](#)  
[Boot time](#)

## Kernel drivers

[Modules](#)  
[Memory](#)  
[Concurrency](#)  
[Char devices](#)  
[I/O](#)  
[Platforms](#)  
[Using sysfs](#)  
[Timers](#)  
[Int. handlers](#)  
[Bottom halves](#)  
[Examples](#)

## 1. Process management

- Scheduling of processes and inter-process communication
- Virtualized processor

## 2. Memory management

- Virtual addressing space
- Virtualized memory

## 3. Filesystems: file abstraction

## 4. Device control: *device drivers* for every peripherals installed on the system (e.g. hard drive, keyboard, floppy drive, etc.)

## 5. Networking: delivering of data packets across processes and network interfaces

# Kernel's role

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

## 1. Process management

- Scheduling of processes and inter-process communication
- Virtualized processor

## 2. Memory management

- Virtual addressing space
- Virtualized memory

## 3. Filesystems: file abstraction

## 4. Device control: *device drivers* for every peripherals installed on the system (e.g. hard drive, keyboard, floppy drive, etc.)

## 5. Networking: delivering of data packets across processes and network interfaces

# Process context and interrupt context

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- The CPU switches to Kernel Mode when either...
  - An application triggers an interrupt or invokes a system call
    - These synchronous interrupts are usually called "Exceptions"
    - The kernel is said to execute in **process context**
    - The kernel executes on behalf of the process
    - This context is managed by the scheduler
  - A hardware device raises an interrupt
    - The kernel is said to execute in **interrupt context**
    - This context is managed by the CPU
    - This is also the case of interrupts generated by the hardware timer

# Process context and interrupt context

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The CPU switches to Kernel Mode when either...

- An application triggers an interrupt or invokes a system call
  - These synchronous interrupts are usually called "Exceptions"
  - The kernel is said to execute in **process context**
  - The kernel executes on behalf of the process
  - This context is managed by the scheduler

- A hardware device raises an interrupt

- The kernel is said to execute in **interrupt context**
  - This context is managed by the CPU
  - This is also the case of interrupts generated by the hardware timer

# Interrupt context

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- In interrupt context, the kernel cannot make any assumption on the process that is currently in execution
  1. The `current` pointer is meaningless
  2. **Blocking is not allowed!**
    - No sleeping
    - No blocking operations
    - No calls to functions that could sleep (e.g., `kmalloc()`)
  3. No scheduling and no calls to `schedule()`
  4. Access to user space is forbidden
- The function `in_interrupt()` returns nonzero if running in interrupt context. See `asm/hardirq.h`.

# Interrupt context (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The functions of the kernel executed in interrupt context are also said to be *asynchronous*.
- Several types of asynchronous functions:
  1. **interrupt handlers**: each device driver defines an interrupt handler to handle the interrupts coming from the hardware device
  2. deferrable functions (**softirqs** and **tasklets**): usually activated by interrupt handlers, they perform lower priority jobs before returning in process context
  3. **timers**: allow to execute arbitrary functions after predefined delays

# Interrupt context (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The functions of the kernel executed in interrupt context are also said to be *asynchronous*.
- Several types of asynchronous functions:
  1. **interrupt handlers**: each device driver defines an interrupt handler to handle the interrupts coming from the hardware device
  2. deferrable functions (**softirqs** and **tasklets**): usually activated by interrupt handlers, they perform lower priority jobs before returning in process context
  3. **timers**: allow to execute arbitrary functions after predefined delays

# Interrupt context (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The functions of the kernel executed in interrupt context are also said to be *asynchronous*.
- Several types of asynchronous functions:
  1. **interrupt handlers**: each device driver defines an interrupt handler to handle the interrupts coming from the hardware device
  2. **deferrable functions (softirqs and tasklets)**: usually activated by interrupt handlers, they perform lower priority jobs before returning in process context
  3. **timers**: allow to execute arbitrary functions after predefined delays

# Interrupt context (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The functions of the kernel executed in interrupt context are also said to be *asynchronous*.
- Several types of asynchronous functions:
  1. **interrupt handlers**: each device driver defines an interrupt handler to handle the interrupts coming from the hardware device
  2. deferrable functions (**softirqs** and **tasklets**): usually activated by interrupt handlers, they perform lower priority jobs before returning in process context
  3. **timers**: allow to execute arbitrary functions after predefined delays

# Kernel

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The kernel is a huge program
  - More than 10 millions of lines
- The kernel must be compiled and linked before being loaded in RAM
- A User Mode program enters in Kernel Mode by issuing a special `int` instruction
  - In Linux, interrupt `0x80` is reserved to implement system calls
  - The names of the system calls in the code begin with `sys_`
- A program in Kernel Mode can put the CPU back to User Mode by executing the `iret` (Interrupt Return) instruction

# Kernel design

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Monolithic kernel:** single statically linked executable image that runs in a single address space.
  - Simplicity and performance
  - Most Unix systems
- **Microkernel:** the functionality of the kernel is broken down into separate processes, usually called *servers*
  - Direct invocation is not possible
  - Communication handled by message passing
  - More overhead but fault tolerance
  - Example: *Minix*
- **Linux:** borrows both advantages
  - Monolithic kernel
  - Kernel preemption
  - Modular design (*Loadable Kernel Modules*)
  - Support for kernel threads

# Kernel design

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Monolithic kernel:** single statically linked executable image that runs in a single address space.
  - Simplicity and performance
  - Most Unix systems
- **Microkernel:** the functionality of the kernel is broken down into separate processes, usually called *servers*
  - Direct invocation is not possible
  - Communication handled by message passing
  - More overhead but fault tolerance
  - Example: *Minix*
- **Linux:** borrows both advantages
  - Monolithic kernel
  - Kernel preemption
  - Modular design (*Loadable Kernel Modules*)
  - Support for kernel threads

# Kernel design

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Monolithic kernel:** single statically linked executable image that runs in a single address space.
  - Simplicity and performance
  - Most Unix systems
- **Microkernel:** the functionality of the kernel is broken down into separate processes, usually called *servers*
  - Direct invocation is not possible
  - Communication handled by message passing
  - More overhead but fault tolerance
  - Example: *Minix*
- **Linux:** borrows both advantages
  - Monolithic kernel
  - Kernel preemption
  - Modular design (*Loadable Kernel Modules*)
  - Support for kernel threads

# Kernel modules

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Modules are pieces of code that can be loaded and unloaded into the kernel at runtime
- Extend the functionality of the kernel without the need to reboot the system
- Object code that can be dynamically linked to the running kernel
- Without modules, we would have to build monolithic kernels and add new functionalities directly into the kernel image
- Without modules we have to rebuild the kernel and reboot the system every time we want new functionalities

# Kernel modules

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Modules are pieces of code that can be loaded and unloaded into the kernel at runtime
- Extend the functionality of the kernel without the need to reboot the system
- Object code that can be dynamically linked to the running kernel
- Without modules, we would have to build monolithic kernels and add new functionalities directly into the kernel image
- Without modules we have to rebuild the kernel and reboot the system every time we want new functionalities

# Kernel history

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

1991 Release 0.01 — 72KB compressed

1991 Release 0.12 — GNU License

1992 First distribution: MCC (Manchester Computing Centre)  
Linux based on kernel 0.12

1994 Release 1.0 — 1MB compressed

1995 Release 1.2 — 2.2 MB compressed

1996 Release 2.0 — 5MB compressed

1999 Release 2.2 — 12MB compressed

2001 Release 2.4 — 20MB compressed

2003 Release 2.6 — 40MB compressed

Today **Release 3.18 — 117MB compressed**

# Rates of kernel change

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Linux 2.2.14 (2000): around 2M lines of code
  - Linux 2.4.18 (2002): around 4M lines of code
  - Linux 2.6.11 (2005): around 6M lines of code
- 
- First 6 months of 2.4 develop: -220,000 lines, +600,000 lines
  - First 6 months of 2.6 develop: -600,000 lines, +900,000 lines

# Rates of kernel change (2)

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Rate of change in 2007-2008:

- 4300 lines added
- 1800 lines removed
- 1500 lines modified

per day!

- It's not possible to keep up with this very high rate: put your code in the kernel instead of keeping it outside the kernel!

# Rates of kernel change (2)

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Rate of change in 2007-2008:

- 4300 lines added
- 1800 lines removed
- 1500 lines modified

per day!

- It's not possible to keep up with this very high rate: put your code in the kernel instead of keeping it outside the kernel!

# Rates of kernel change (2)

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Rate of change in 2007-2008:

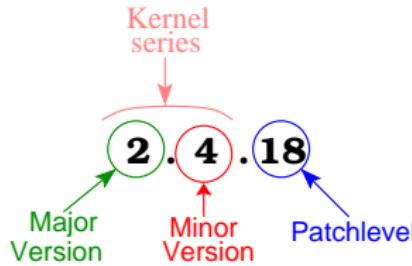
- 4300 lines added
- 1800 lines removed
- 1500 lines modified

per day!

- It's not possible to keep up with this very high rate: put your code in the kernel instead of keeping it outside the kernel!

# Version numbering: 2.4 series

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples



## • Stable kernels

- Production-level releases providing bug fixes or new drivers
- Even minor number

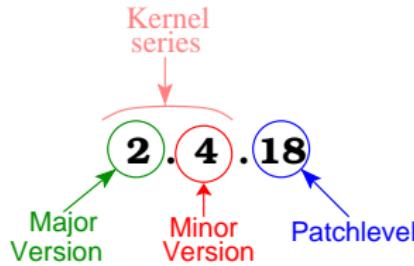
## • Development kernels

- Experiments with new solutions and drastic changes
- Odd minor number

## • Not valid for 2.6 series: see later

# Version numbering: 2.4 series

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
    /dev filesystem  
    proc filesystem  
    sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
  Examples



## • Stable kernels

- Production-level releases providing bug fixes or new drivers
- Even minor number

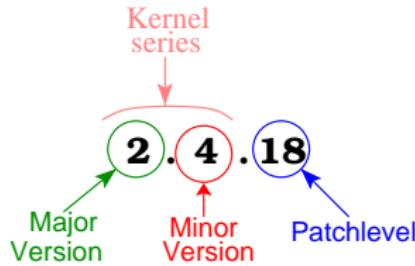
## • Development kernels

- Experiments with new solutions and drastic changes
- Odd minor number

- Not valid for 2.6 series: see later

# Version numbering: 2.4 series

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
    /dev filesystem  
    proc filesystem  
    sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
  Examples



## • Stable kernels

- Production-level releases providing bug fixes or new drivers
- Even minor number

## • Development kernels

- Experiments with new solutions and drastic changes
- Odd minor number

## • Not valid for 2.6 series: see later

# Version numbering: 2.6 series

- Since 2.6 series, it was decided to use only stable kernels (i.e. no 2.7 series) and put experimental code inside stable kernels
- A fourth number first occurred when a grave error, which required immediate fixing, was encountered in 2.6.8's NFS code
  - Not enough changes to legitimize the release of a new minor revision
  - So, 2.6.8.1 was released
- With 2.6.11, this became the new official versioning policy
  - Bug-fixes and security patches are now managed by the fourth number
  - Bigger changes are only implemented in minor revision changes (the third number)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Version numbering: 3.x series

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Since release 3.0, we only have three numbers:
  - The first and second number specify the version (e.g., 3.1)
  - The third number specifies the level of bug fixing

# Current version numbering

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

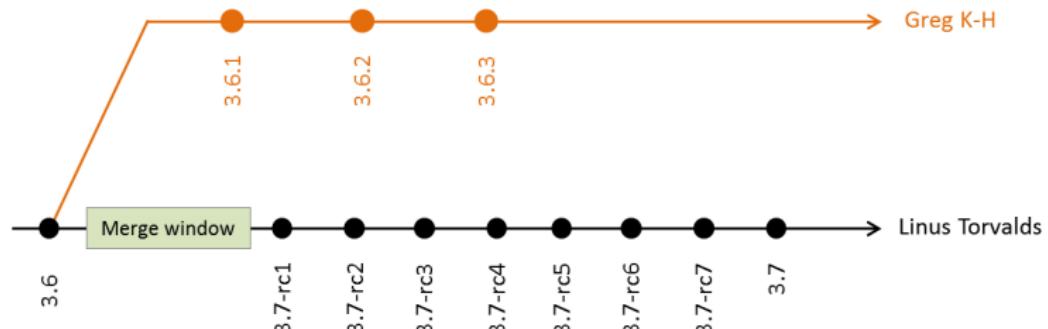
[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)



# Changes from one release to the other

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Changes in kernel internal API:

<http://lwn.net/Articles/2.6-kernel-api/>

- Summary of changes:

<http://wiki.kernelnewbies.org/LinuxChanges>

# Linux kernel architecture

About 28,000 source files:

## Kernel

[Intro](#)  
[Compiling](#)  
[Code](#)  
[Debugging](#)  
[Linked lists](#)  
[Scheduling](#)  
[Linux filesystem](#)  
[/dev filesystem](#)  
[proc filesystem](#)  
[sys filesystem](#)  
[Syscalls](#)  
[Real-Time](#)  
[Boot time](#)

## Kernel drivers

[Modules](#)  
[Memory](#)  
[Concurrency](#)  
[Char devices](#)  
[I/O](#)  
[Platforms](#)  
[Using sysfs](#)  
[Timers](#)  
[Int. handlers](#)  
[Bottom halves](#)  
[Examples](#)

<code>arch/</code>	Low-level architecture-specific source code (all code outside <code>arch/</code> should be portable)
<code>block/</code>	Block layer core
<code>COPYING</code>	Linux copying license (i.e., GPL)
<code>CREDITS</code>	Linux main contributors
<code>crypto/</code>	Cryptographic libraries
<code>Documentation/</code>	Kernel documentation
<code>drivers/</code>	Hierarchy of <b>Device drivers</b> . All drivers except those for sound cards!
<code>fs/</code>	Virtual FileSystem (VFS) layer and individual filesystems
<code>include/</code>	Kernel headers
<code>init/</code>	Kernel boot and initialization
<code>ipc/</code>	Interprocess communication code (e.g. semaphores)

# Linux kernel architecture (2)

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

**kernel/**

Core subsystems (e.g. scheduler)

**lib/**

Generic library support routines  
(e.g. `zlib`, `crc32`)

**MAINTAINERS**

Maintainers of each part

**mm/**

Memory management subsystem  
and Virtual Memory (VM)

**net/**

Networking subsystem (e.g. IPv4, IPv6, TCP).  
Protocols, not drivers!

**scripts/**

Scripts used to build the kernel

**security/**

Linux Security Module  
(i.e. capabilities and SELinux)

**sound/**

Sound subsystem

**usr/**

Code to generate initramfs

# Linux kernel version: Makefile

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Kernel version is contained inside the main `Makefile`:

`VERSION = 2`

`PATCHLEVEL = 6`

`SUBLEVEL = 30`

`EXTRAVERSION =`

- On a running kernel, can be seen through either:

■ `uname -a`

■ `cat /proc/version`

- You can easily track your own kernel version by customizing the `EXTRAVERSION` field

# Linux kernel architecture: `arch/`

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Architecture-specific code
- It contains many subdirectories, one for each supported architecture
- Some subdirectories:
  - `alpha`
  - `arm`
  - `avr32`
  - `blackfin`
  - `ia64`
  - `m68k`
  - `m68knommu`
  - `mips`
  - `powerpc`
  - `sparc`
  - `x86`

# Linux kernel architecture: arch/x86/

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

**mach-\*/**

Subdirectories containing code and configuration for each specific board

**boot/**

Assembly files, related to booting Linux

Here the build process puts the compiled kernel

**include/asm/**

Architecture specific headers

**kernel/**

The bulk of the x86 code, including IRQ handling, processes, signals, system calls and pci support

**lib/**

Optimized routines for common tasks  
(e.g. `memcpy`)

**mm/**

x86 specific memory management

# Linux kernel architecture: drivers/

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

Drivers for each supported device

One of the most important parts from a user point of view

Some subdirectories:

<code>block/</code>	Block devices
<code>char/</code>	Character devices ( <code>/dev/null</code> and <code>/dev/zero</code> included)
<code>cpufreq/</code>	Cpu frequency scaling
<code>leds/</code>	Led devices
<code>net/</code>	Drivers for networking cards
<code>pci/</code>	PCI devices
<code>scsi/</code>	SCSI devices
<code>usb/</code>	USB devices

# Linux kernel architecture: fs/

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Virtual FileSystem (VFS) layer and specific filesystems
- Some subdirectories:
  - `ext2/`
  - `ext3/`
  - `jffs2/`
  - `nfs/`
  - `proc/`
  - `reiserfs/`
  - `sysfs/`
  - `vfat/`

# Linux kernel architecture: `include/`

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

## Header files

Some subdirectories:

- `linux/`      Most important part
- `asm`          Symbolic link to *asm-architecture*  
                  created during the build process

# Linux 3.18 directories size

```
for i in `ls | xargs -r`; do du -s --apparent-size $i; done
```

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

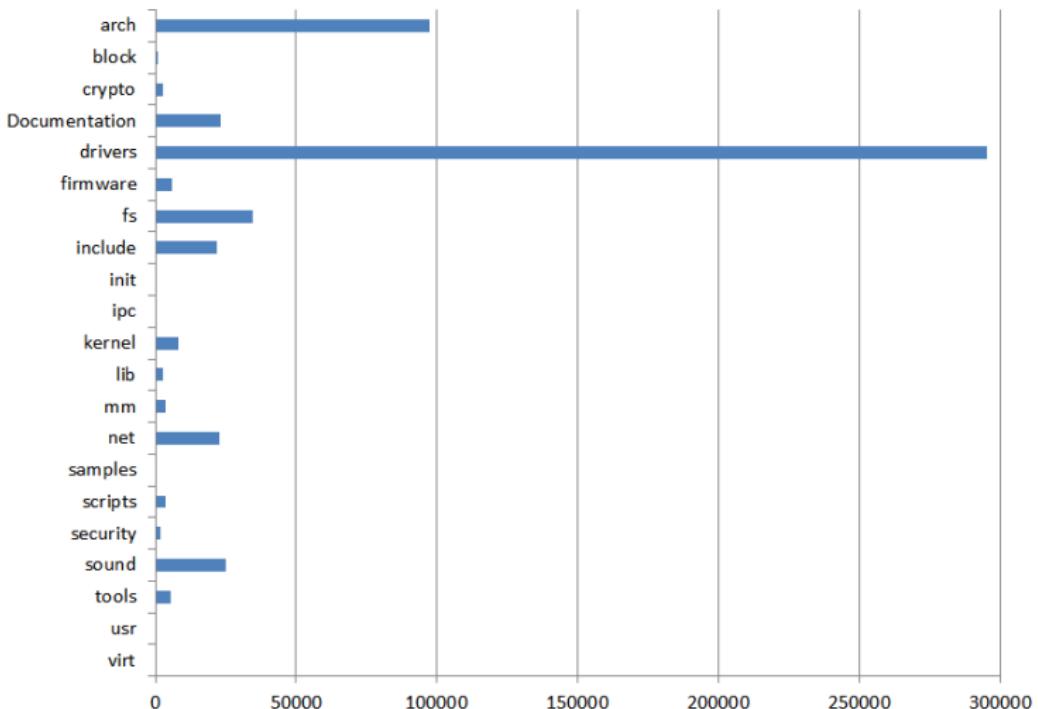
[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)



# Kernel cycle development

## Kernel

[Intro](#)  
[Compiling](#)  
[Code](#)  
[Debugging](#)  
[Linked lists](#)  
[Scheduling](#)  
[Linux filesystem](#)  
[/dev filesystem](#)  
[proc filesystem](#)  
[sys filesystem](#)  
[Syscalls](#)  
[Real-Time](#)  
[Boot time](#)

## Kernel drivers

[Modules](#)  
[Memory](#)  
[Concurrency](#)  
[Char devices](#)  
[I/O](#)  
[Platforms](#)  
[Using sysfs](#)  
[Timers](#)  
[Int. handlers](#)  
[Bottom halves](#)  
[Examples](#)

- Development community is highly technical
- Around 2400 developers
- Famous kernel hackers listed in the [CREDITS](#) file
- Most parts of the kernel have an associated *maintainer*
  - Listed in the file [MAINTAINERS](#)
- Informal design/development
  - Rapid development cycle
  - Discussions and decisions in list, no meeting required
  - No much high-level planning: no deadlines/schedules
  - No design documents
  - Code producers make (most) decisions
  - New ideas best presented as code

# Kernel cycle development

## Kernel

[Intro](#)  
[Compiling](#)  
[Code](#)  
[Debugging](#)  
[Linked lists](#)  
[Scheduling](#)  
[Linux filesystem](#)  
[/dev filesystem](#)  
[proc filesystem](#)  
[sys filesystem](#)  
[Syscalls](#)  
[Real-Time](#)  
[Boot time](#)

## Kernel drivers

[Modules](#)  
[Memory](#)  
[Concurrency](#)  
[Char devices](#)  
[I/O](#)  
[Platforms](#)  
[Using sysfs](#)  
[Timers](#)  
[Int. handlers](#)  
[Bottom halves](#)  
[Examples](#)

- Development community is highly technical
- Around 2400 developers
- Famous kernel hackers listed in the [CREDITS](#) file
- Most parts of the kernel have an associated *maintainer*
  - Listed in the file [MAINTAINERS](#)
- Informal design/development
  - Rapid development cycle
  - Discussions and decisions in list, no meeting required
  - No much high-level planning: no deadlines/schedules
  - No design documents
  - Code producers make (most) decisions
  - New ideas best presented as code

# Join the community!

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Chance to work on a real OS — any part of it that interests you
- Meritocracy:
  - Good ideas and good code are rewarded
  - Payment in kind, self-interest
- Strong trust system: start with small patches for credibility
- Keep your code updated for the current kernel version
- Follow the culture: **have fun!! :)**

# LKML: The Linux Kernel Mailing List

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- <http://vger.kernel.org/>
- All kernel developers (from Linus Torvalds on down) subscribe to this list
- Up to 300 messages a day
- FAQ available at <http://www.tux.org/lkml>
- No registration needed to post messages

# LKML: The Linux Kernel Mailing List (2)

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

Ways for reading messages:

1. Subscribe by sending an email containing  
`subscribe linux-kernel your@email.address`  
in plain text to  
`majordomo@vger.kernel.org`

2. Subscribe to a Newsgroup from your email client

- E.g., `news.gmane.org`

3. Read the archive through web pages

- `http://www.lkml.org`

# LKML: The Linux Kernel Mailing List (2)

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

## Ways for reading messages:

1. Subscribe by sending an email containing  
`subscribe linux-kernel your@email.address`  
in plain text to  
`majordomo@vger.kernel.org`
2. Subscribe to a Newsgroup from your email client
  - E.g., `news.gmane.org`
3. Read the archive through web pages
  - `http://www.lkml.org`

# LKML: The Linux Kernel Mailing List (2)

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

## Ways for reading messages:

1. Subscribe by sending an email containing  
`subscribe linux-kernel your@email.address`  
in plain text to  
`majordomo@vger.kernel.org`

2. Subscribe to a Newsgroup from your email client
  - E.g., `news.gmane.org`

3. Read the archive through web pages
  - `http://www.lkml.org`

# Mailing list etiquette

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Always use reply-to all
- No HTML
- No encoded or zipped attachments
- ALL CAPS == SHOUTING
- Use < 80-column width
- Keep it technical and professional
- No top-posting
- Possible downside: flames
  - If attacked (flamed) stick with technical points, don't get involved with attacks

# Submitting bug reports

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Kernel error messages can be seen through the `dmesg` shell command
- Fully describe the problem:
  - System output
  - Kernel version
  - Distribution
  - Hardware description
- Send to LKML and to the appropriate maintainer (if any) or post on <http://bugzilla.kernel.org>
- For more info:
  - File `REPORTING-BUGS`
  - File `Documentation/oops-tracing.txt`
  - <http://lwn.net/Articles/139918/>

# Differences wrt normal user-space applications

Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Not a single entry point: a different entry point for any type of interrupt recognized by the kernel
- No memory protection
  - No control over illegal memory access
- Synchronization and concurrency are major concerns
  - Susceptible to race conditions on shared resources!
  - Use spinlocks and semaphores.
- A fault can crash the whole system

# Differences wrt normal user-space applications (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- No libraries to link to
  - Never include the usual header files, like `<stdio.h>`
- Small stack: 4 or 8 KB
  - Do not use large variables, e.g.,  
`char var [32*1024];`
  - Allocate large structures at runtime (`kmalloc`)
- No floating point arithmetic

# Why recompiling the kernel ?

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Why recompiling the kernel ?

- To take advantage of new features
- To support hardware not supported by the stock kernel included in your distribution
- To close potential holes in modules not used
- To tailor the kernel specifically of your hardware, resulting in a performance boost

- See Documentation/Changes for the minimal requirements of compilers (i.e., `gcc`) and utils

- A too new compiler can be as problematic as well as a too old one

# Obtaining a new kernel

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## ● Binaries

- RPM (RedHat), YaST (Suse), Lisa (Caldera), apt-get (Debian and Ubuntu)
- Already compiled
- Very easy to install
- Distribution dependent
- Not 100% customizable

## ● Sources

- Downloadable from <http://www.kernel.org>
- Installation not easy
- Completely customizable
- Distribution independent

# Obtaining a new kernel

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## ● Binaries

- RPM (RedHat), YaST (Suse), Lisa (Caldera), apt-get (Debian and Ubuntu)
- Already compiled
- Very easy to install
- Distribution dependent
- Not 100% customizable

## ● Sources

- Downloadable from <http://www.kernel.org>
- Installation not easy
- Completely customizable
- Distribution independent

# Obtaining the kernel source

- Kernel distributed on <http://www.kernel.org> as
  - Gzip/Bzip2 tarball: `linux-x.y.z.tar.gz`  
Unzip using `tar -xvzf linux-x.y.z.tar.bz2`
  - Incremental patch: `patch-x.y.z`
    - Retains settings from the previous kernel compilation
    - One patch for each patch level
    - Apply the patch to the previous kernel using  
`patch -p1 < patch-x.y.z`
  - Git repository:  
`git clone`  
`git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
- The kernel is typically copied and compiled into the `/usr/src` directory

# Obtaining the kernel source

- Kernel distributed on <http://www.kernel.org> as
  - Gzip/Bzip2 tarball: `linux-x.y.z.tar.gz`  
Unzip using `tar -xvzf linux-x.y.z.tar.bz2`
  - Incremental patch: `patch-x.y.z`
    - Retains settings from the previous kernel compilation
    - One patch for each patch level
    - Apply the patch to the previous kernel using  
`patch -p1 < patch-x.y.z`
  - Git repository:  
`git clone`  
`git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
- The kernel is typically copied and compiled into the `/usr/src` directory

Kernel  
Intro  
**Compiling**  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Obtaining the kernel source

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Kernel distributed on <http://www.kernel.org> as

- Gzip/Bzip2 tarball: `linux-x.y.z.tar.gz`  
Unzip using `tar -xvzf linux-x.y.z.tar.bz2`

- Incremental patch: `patch-x.y.z`

- Retains settings from the previous kernel compilation
- One patch for each patch level
- Apply the patch to the previous kernel using  
`patch -p1 < patch-x.y.z`

- Git repository:

`git clone`

`git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`

- The kernel is typically copied and compiled into the `/usr/src` directory

# Obtaining the kernel source

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Kernel distributed on <http://www.kernel.org> as
  - Gzip/Bzip2 tarball: `linux-x.y.z.tar.gz`  
Unzip using `tar -xvf linux-x.y.z.tar.gz`
  - Incremental patch: `patch-x.y.z`
    - Retains settings from the previous kernel compilation
    - One patch for each patch level
    - Apply the patch to the previous kernel using  
`patch -p1 < patch-x.y.z`
  - Git repository:  
`git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
- The kernel is typically copied and compiled into the `/usr/src` directory

# Compiling the kernel: download

Kernel  
Intro  
**Compiling**  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

1. Download the kernel `linux-x.y.z.tar.gz` from  
`http://www.kernel.org`
2. Copy the kernel into the `/usr/src` directory:  
`cp linux-x.y.z.tar.gz /usr/src/`
3. Unzip the kernel:  
`tar -xvzf linux-x.y.z.tar.gz`
4. Change the directory name:  
`mv linux-x.y.z linux-x.y.z-mycopy`
5. Remove any previous symbolic link:  
`rm -f linux`
6. Create a new symbolic link:  
`ln -s linux-x.y.z-mycopy linux`
7. Enter into the new kernel's directory:  
`cd linux`

# Configuring the kernel: `.config`

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Configuring the kernel means selecting features and device drivers according to your hardware and needs.
- Different features can be included, excluded or compiled as loadable external modules.
- Kernel configuration is written in `.config` file
- Each option can be set in 3 possible ways:

`=y`: Support enabled and included in the kernel

`=m`: Support enabled but compiled as a separate kernel module

*Commented*: Support not enabled

# Configuring the kernel: example of .config

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
#  
# USB Network Adapters  
#  
# CONFIG_USB_USBNET is not set  
# CONFIG_WAN is not set  
CONFIG_PPP=m  
# CONFIG_PPP_MULTILINK is not set  
# CONFIG_PPP_FILTER is not set  
CONFIG_PPP_ASYNC=m  
CONFIG_PPP_SYNC_TTY=m  
CONFIG_PPP_DEFLATE=m  
# CONFIG_PPPOE is not set  
# CONFIG_SLIP is not set  
CONFIG_SLHC=m  
CONFIG_NETCONSOLE=y
```

# Compiling the kernel: configuration

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 8. Some programs allow to configure the kernel:

- `make config` (terminal)
- `make menuconfig` (pseudo-graphical)
- `make xconfig` and `make gconfig` (graphical)

- Options can be answered in 3 possible ways: `Y`, `M` or `N`
- A backup copy of the old configuration is written in the `.config.old` file
- To search a string in `menuconfig` use `/` (like in `vi`)

# Compiling the kernel: make menuconfig

## Kernel

Intro  
Compiling

Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem

sys filesystem  
Syscalls  
Real-Time

Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

I/O  
Platforms  
Using sysfs

Timers  
Int. handlers  
Bottom halves  
Examples

## Linux Kernel v2.6.11.6 Configuration

### Linux Kernel Configuration

Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> includes, <M> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] built-in [ ] excluded <M> module < >

#### Code maturity level options --->

General setup --->

Loadable module support --->

Processor type and features --->

Power management options (ACPI, APM) --->

Bus options (PCI, PCMCIA, EISA, MCA, ISA) --->

Executable file formats --->

Device Drivers --->

File systems --->

Profiling support --->

Kernel hacking --->

[\*]

<Select>

< Exit >

< Help >

# Compiling the kernel: useful commands

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **make help**

- List all available **make** targets

- **make oldconfig**

- Update current config using an existing **.config** as base
  - Useful to upgrade a **.config** file from an earlier kernel release
  - Values for new symbols are asked

# Compiling the kernel: useful commands

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `make help`

- List all available `make` targets

- `make oldconfig`

- Update current config using an existing `.config` as base
  - Useful to upgrade a `.config` file from an earlier kernel release
  - Values for new symbols are asked

# Compiling the kernel: useful commands

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **make allnoconfig**

- Only sets strongly recommended settings to **y**
- Sets all other settings to **n**
- Useful to select only the minimum required set of features and drivers

- **make defconfig**

- Default answer to all options

- **make <micro>\_defconfig**

- Default configuration files available for many boards in the **arch/<arch>/configs/** directory
- This command copies the default configuration from **arch/\$ARCH/configs/<micro>\_defconfig** to **.config**
- The configuration can then be customized through **make menuconfig**

# Compiling the kernel: useful commands

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **make allnoconfig**

- Only sets strongly recommended settings to **y**
- Sets all other settings to **n**
- Useful to select only the minimum required set of features and drivers

- **make defconfig**

- Default answer to all options

- **make <micro>\_defconfig**

- Default configuration files available for many boards in the **arch/<arch>/configs/** directory
- This command copies the default configuration from **arch/\$ARCH/configs/<micro>\_defconfig** to **.config**
- The configuration can then be customized through **make menuconfig**

# Compiling the kernel: useful commands

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `make allnoconfig`

- Only sets strongly recommended settings to `y`
- Sets all other settings to `n`
- Useful to select only the minimum required set of features and drivers

- `make defconfig`

- Default answer to all options

- `make <micro>_defconfig`

- Default configuration files available for many boards in the `arch/<arch>/configs/` directory
- This command copies the default configuration from `arch/$ARCH/configs/<micro>_defconfig` to `.config`
- The configuration can then be customized through `make menuconfig`

# Compiling the kernel: Kconfig

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The configuration and build mechanism is based on **Kconfig**
- The syntax of Kconfig is explained in [Documentation/kbuild/](#).
- Typical structure of a script language

# Compiling the kernel: arch/arm/Kconfig

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
source "init/Kconfig"

#...

choice
prompt "Kernel command line type" if ARM_ATAG_DTB_COMPAT
default ARM_ATAG_DTB_COMPAT_CMDLINE_FROM_BOOTLOADER

config ARM_ATAG_DTB_COMPAT_CMDLINE_FROM_BOOTLOADER
bool "Use bootloader kernel arguments if available"
help
    Uses the command-line options passed by the boot loader instead of
    the device tree bootargs property. If the boot loader doesn't provide
    any, the device tree bootargs property will be used.

config ARM_ATAG_DTB_COMPAT_CMDLINE_EXTEND
bool "Extend with bootloader kernel arguments"
help
    The command-line arguments provided by the boot loader will be
    appended to the the device tree bootargs property.

endchoice
```

# Compiling the kernel: Makefiles

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
# CPU-specific support
obj-$(CONFIG_SOC_AT91RM9200) += at91rm9200.o at91rm9200_time.o
obj-$(CONFIG_SOC_AT91SAM9260) += at91sam9260.o
obj-$(CONFIG_SOC_AT91SAM9261) += at91sam9261.o
obj-$(CONFIG_SOC_AT91SAM9263) += at91sam9263.o
obj-$(CONFIG_SOC_AT91SAM9G45) += at91sam9g45.o
```

# Compiling the kernel: build

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 9. Compile the kernel by typing

`make dep && make bzImage && make modules` [2.2 and 2.4]

or

`make` [since 2.6]

# Compiling the kernel: build (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

10. The new kernel image is in `arch/architecture/boot/`:

- **bzImage:**

- Specific for x86 architecture with PC-style BIOS
- Typically copied as `/boot/vmlinuz-x.y.z`
- zlib (not bzip2) compressed image  
(bz just means “big zipped”)
- Kernel is uncompressed into high memory (over 1MB)

- **zImage:**

- Default image on ARM
- zlib compressed image
- Kernel is uncompressed into low memory

When linking the kernel, gcc also creates:

- **vmlinux:** stand-alone uncompressed monolithic ELF image

- **System.map:** file containing the addresses of all global symbols (used by debuggers and kernel profilers).

# Compiling the kernel: build (3)

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 11. Install modules by typing

`make modules_install`

- The modules are copied to `/lib/modules/x.y.z`
- It is possible to specify where to copy modules:  
`make INSTALL_MOD_PATH=<directory> modules_install`
- Module dependencies are automatically computed from exported symbols, and saved in  
`/lib/modules/x.y.z/modules.dep`

# Cross-compiling the kernel

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- When cross-compiling the kernel, **ARCH** and **CROSS\_COMPILE** variables must be set:

- ARCH** specifies the architecture
- CROSS\_COMPILE** contains the **first part** of the name of compiler tools

- Example:

```
make ARCH=arm CROSS_COMPILE=arm-linux-
```

- Typically, these variables are automatically set by the scripts (e.g., Evelin BSP or Itib)

# Cross-compiling the kernel

Kernel

Intro

**Compiling**

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- When cross-compiling the kernel, **ARCH** and **CROSS\_COMPILE** variables must be set:

- ARCH** specifies the architecture
- CROSS\_COMPILE** contains the **first part** of the name of compiler tools

- Example:

```
make ARCH=arm CROSS_COMPILE=arm-linux-
```

- Typically, these variables are automatically set by the scripts (e.g., Evelyn BSP or Itib)

# Cross-compiling the kernel

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- When cross-compiling the kernel, `ARCH` and `CROSS_COMPILE` variables must be set:

- `ARCH` specifies the architecture
  - `CROSS_COMPILE` contains the **first part** of the name of compiler tools

- Example:

```
make ARCH=arm CROSS_COMPILE=arm-linux-
```

- Typically, these variables are automatically set by the scripts (e.g., Evelin BSP or Itib)

# Programming language

## Kernel

Intro  
Compiling

## Code

Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Like all Unix-like OSs, Linux is coded mostly in C
  - No C++ (see <http://www.tux.org/lkml/#s153>)
- No access to the C library
  - No `printf`: use `printk`:  
`printk(KERN_ERR "This is an error!");`
- Not coded in ANSI C
  - Both ISO C99 and gcc-specific extensions used
    - A few alternate compilers are supported (e.g., Intel and Marvell)
  - 64-bit `long long` data type
  - Inline functions to reduce overhead:  
`static inline void foo (...);`
  - Branch annotation:  
`if (likely(pippo)) {  
 /*...*/  
}`

# Programming language

## Kernel

Intro  
Compiling  
**Code**

Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Like all Unix-like OSs, Linux is coded mostly in C
  - No C++ (see <http://www.tux.org/lkml/#s153>)
- No access to the C library
  - No `printf`: use  `printk`:  
 `printk(KERN_ERR "This is an error!");`
- Not coded in ANSI C
  - Both ISO C99 and gcc-specific extensions used
    - A few alternate compilers are supported (e.g., Intel and Marvell)
  - 64-bit `long long` data type
  - Inline functions to reduce overhead:  
 `static inline void foo (...);`
  - Branch annotation:  
 `if (likely(pippo)) {`  
 `/*...*/`  
}

# Programming language (2)

## Kernel

Intro

Compiling

### Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Need GNU `gcc`

- At least version 3.2 (see [Documentation/Changes](#))
  - Intel compiler works too (on Intel platforms only)

- Few small critical functions coded in Assembly (around 10% of the code)

- Architecture-dependent code placed in
    - `arch/<arch>/include/asm/`  
(previously in `include/asm-<arch>/`)
    - `include/asm-generic/`
  - Inline assembly (`asm` primitive)

# Programming language (2)

Kernel

Intro

Compiling

**Code**

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Need GNU `gcc`
  - At least version 3.2 (see [Documentation/Changes](#))
  - Intel compiler works too (on Intel platforms only)
- Few small critical functions coded in Assembly (around 10% of the code)
  - Architecture-dependent code placed in
    - `arch/<arch>/include/asm/`  
(previously in `include/asm-<arch>/`)
    - `include/asm-generic/`
  - Inline assembly (`asm` primitive)

# Programming language (3)

Kernel  
Intro  
Compiling  
**Code**

Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

Result of `sloccount` for Linux 3.18:

Totals grouped by language (dominant language first):

ansic:	12261184	(96.40%)
asm:	283674	(2.23%)
xml:	126713	(1.00%)
perl:	19930	(0.16%)
sh:	8504	(0.07%)
python:	6149	(0.05%)
cpp:	5145	(0.04%)
yacc:	4299	(0.03%)
lex:	2243	(0.02%)
awk:	746	(0.01%)
pascal:	231	(0.00%)
lisp:	218	(0.00%)
sed:	30	(0.00%)

# Coding style

## Kernel

Intro  
Compiling  
**Code**

Debugging  
Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers

Int. handlers  
Bottom halves  
Examples

- Explained in the file [Documentation/CodingStyle](#)
- Philosophy: **Make it simpler, faster and smaller**
- Maintainability and aesthetics
  - Clean implementation, not ugly hacks
- Line size: should be less than 80 characters
- Small functions with less than 10 local parameters
- No mixed case: use `get_active_tty`, not `getActiveTty`
- Avoid `typedefs`
- Avoid `extern` in C files, use headers instead
- Indentation
  - Use tabs for indentation
  - Tab size is 8 (not 8 spaces!)
  - On vim use: `set sw=8 ts=8` and `set noexpandtab`
  - Use `indent -kr -i8 -ts8 -sob -l180 -ss -bs -ps1`  
*file\_name*

# Coding style

## Kernel

Intro  
Compiling  
**Code**

Debugging  
Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

I/O

Platforms  
Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Explained in the file Documentation/CodingStyle
- Philosophy: **Make it simpler, faster and smaller**
- Maintainability and aesthetics
  - Clean implementation, not ugly hacks
- Line size: should be less than 80 characters
- Small functions with less than 10 local parameters
- No mixed case: use `get_active_tty`, not `getActiveTty`
- Avoid `typedefs`
- Avoid `extern` in C files, use headers instead
- Indentation
  - Use tabs for indentation
  - Tab size is 8 (not 8 spaces!)
  - On vim use: `set sw=8 ts=8` and `set noexpandtab`
  - Use `indent -kr -i8 -ts8 -sob -l180 -ss -bs -ps1`  
*file\_name*

# Coding style (2)

Kernel  
Intro  
Compiling  
**Code**

Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Avoid adding more `ioctls`: use `/sys`
- Don't introduce kernel drivers if the same functionality can be done reasonably in user space
- Don't include unused files
- Include files in alphabetical order
- Avoid `void*` functions
- Include Copyright and license: `MODULE_LICENSE("GPL");`
- Don't use recursion
- Minimize Macro usage (prefer `static inline` functions for type-checking)
- Use (but don't abuse) `goto`, especially for error handling
- Functions starting with double underscore (`__`) are low-level components and should be used with caution

# Coding style (2)

Kernel  
Intro  
Compiling  
**Code**

Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Avoid adding more `ioctls`: use `/sys`
- Don't introduce kernel drivers if the same functionality can be done reasonably in user space
- Don't include unused files
- Include files in alphabetical order
- Avoid `void*` functions
- Include Copyright and license: `MODULE_LICENSE("GPL");`
- Don't use recursion
- Minimize Macro usage (prefer `static inline` functions for type-checking)
- Use (but don't abuse) `goto`, especially for error handling
- Functions starting with double underscore (`__`) are low-level components and should be used with caution

# Coding style (3)

Kernel

Intro

Compiling

**Code**

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Braces:

- If statement:

```
if (pippo) {  
    func1();  
    func2();  
} else {  
    func3();  
}
```

- Functions:

```
unsigned long func(void)  
{  
    /* ... */  
}
```

- (Ugly) C99 structure initializers:

- ```
struct foo my_foo = {  
    .a = INITIAL_A,  
    .b = INITIAL_B,  
};
```

# Coding style (3)

Kernel

Intro  
Compiling

**Code**

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Braces:

- If statement:

```
if (pippo) {  
    func1();  
    func2();  
} else {  
    func3();  
}
```

- Functions:

```
unsigned long func(void)  
{  
    /*...*/  
}
```

- (Ugly) C99 structure initializers:

- `struct foo my_foo = {  
 .a = INITIAL_A,  
 .b = INITIAL_B,  
};`

# Coding style (4): Comments

Kernel

Intro

Compiling

**Code**

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Use comments but not for obvious code

- Important comments prefixed with **XXX:**

- Bugs prefixed with **FIXME:**

- C-style comments:

```
/*
 * This is
 * my comment
 */
```

- For automatic documentation (through **make htmldocs**):

```
/**
 * This is
 * my comment
 */
```

# Coding style (5): `ifdefs`

- No `ifdefs` in the source.

- Instead of writing:

```
#ifdef CONFIG_FOO
    foo();
#endif
```

use stubs in header files:

```
#ifdef CONFIG_FOO
static int foo(void)
{
    /*....*/
}
#else
static inline int foo(void) { }
#endif
```

Kernel

Intro

Compiling

**Code**

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

# Generating patches

Kernel

Intro

Compiling

**Code**

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Patches are created through the `diff` command:

```
diff -uprN linux-x.y.z/ linux-modified/ > my_patch
```

# Example of patch

Kernel

Intro

Compiling

**Code**

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
diff -Nrup linux-2.6.23.orig/Makefile linux-2.6.23.new/Makefile
--- linux-2.6.23.orig/Makefile 2008-03-17 16:04:42 +0100
+++ linux-2.6.23.new/Makefile 2008-03-17 16:04:48 +0100 } File date
@@ -1,7 +1,7 @@
VERSION = 2 } Context info: 3 lines before change
PATCHLEVEL = 6
SUBLEVEL = 11 } (useful to apply patch when line numbers change)
-EXTRAVERSION =
+EXTRAVERSION = -evidence } Removed lines (if any) Added lines (if any)
NAME = Arr Matey! A Hairy Bilge Rat!
# *DOCUMENTATION* } Context info: 3 lines after change
```

# Applying a patch

Kernel

Intro

Compiling

**Code**

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Patches are applied through the `patch` command:  
`patch -p<n> < my_patch`
  - `n`: number of directory levels to skip in the file paths
- Patch can be reverted with the `-R` option
- Patch can be tested with the `--dry-run` option

# Submitting patches

- Read

- [Documentation/SubmittingPatches](#)
- [Documentation/SubmittingDrivers](#)
- <http://kernelnewbies.org/UpstreamMerge>

- Check the patch by using `linux/scripts/checkpatch.pl`
- Send an email to the LKML [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org) and to the appropriate kernel maintainer, if any (see the `MAINTAINERS` file)
- The subject line of the email beginning with `[PATCH]`
- Put the patch as inline plain text at the bottom of the email
- Include specific and technical description of the changes
- Specify patch dependency
- The patch must apply to the current Linus' git tree

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Submitting patches

- Read

- [Documentation/SubmittingPatches](#)
- [Documentation/SubmittingDrivers](#)
- <http://kernelnewbies.org/UpstreamMerge>

- Check the patch by using `linux/scripts/checkpatch.pl`
- Send an email to the LKML [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org) and to the appropriate kernel maintainer, if any (see the `MAINTAINERS` file)
- The subject line of the email beginning with `[PATCH]`
- Put the patch as inline plain text at the bottom of the email
- Include specific and technical description of the changes
- Specify patch dependency
- The patch must apply to the current Linus' git tree

Kernel  
Intro  
Compiling  
**Code**  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Submitting patches

- Read

- [Documentation/SubmittingPatches](#)
- [Documentation/SubmittingDrivers](#)
- <http://kernelnewbies.org/UpstreamMerge>

- Check the patch by using `linux/scripts/checkpatch.pl`
- Send an email to the LKML [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org) and to the appropriate kernel maintainer, if any (see the **MAINTAINERS** file)
- The subject line of the email beginning with **[PATCH]**
- Put the patch as inline plain text at the bottom of the email
- Include specific and technical description of the changes
- Specify patch dependency
- The patch must apply to the current Linus' git tree

Kernel  
Intro  
Compiling  
**Code**  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Submitting patches (2)

Kernel

Intro

Compiling

**Code**

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Avoid huge patches
  - Don't do multiple things in one patch
  - One patch per email
  - Large patches should be split into logical pieces
- Sign your work with the Developer's Certificate of Origin (**DCO**) by adding

*Signed-off-by: Name Lastname <your@email.address>*  
(this is done automatically by `git`)
- After an extensive discussion on LKML, send the patch to appropriate maintainer

# Kernel debugging

## Kernel

Intro

Compiling

Code

### Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

There is no reason to debug the kernel: it always works... **FALSE!**

- The kernel crashes and we don't know why
- We are writing a new driver or module that don't work properly

# Kernel debugging is a hard task

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. The kernel source code is huge and complex

- More than 10 millions of lines
- Multithreaded
- Hardware-related
- Highly optimized for speed
- Compiled with `-O2`
  - Optimization complicates debugging: misalignment between debugger and source code line numbers

## 2. Kernel errors can be hard to reproduce

# Kernel debugging is a hard task

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. The kernel source code is huge and complex

- More than 10 millions of lines
- Multithreaded
- Hardware-related
- Highly optimized for speed
- Compiled with `-O2`
  - Optimization complicates debugging: misalignment between debugger and source code line numbers

## 2. Kernel errors can be hard to reproduce

# Kernel debugging is a hard task (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 3. It cannot be run by higher-level programs

- Errors can bring down the entire system, destroying much of the evidence that could be used to trace them

## 4. Virtual memory isolates user-space and kernel-space memories

## 5. Startup code is especially difficult

- Proximity to the hardware
- Limited resources available (e.g., no console, limited memory mapping)

# Kernel debugging is a hard task (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 3. It cannot be run by higher-level programs

- Errors can bring down the entire system, destroying much of the evidence that could be used to trace them

## 4. Virtual memory isolates user-space and kernel-space memories

## 5. Startup code is especially difficult

- Proximity to the hardware
- Limited resources available (e.g., no console, limited memory mapping)

# Kernel debugging is a hard task (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

3. It cannot be run by higher-level programs

- Errors can bring down the entire system, destroying much of the evidence that could be used to trace them

4. Virtual memory isolates user-space and kernel-space memories

5. Startup code is especially difficult

- Proximity to the hardware
- Limited resources available (e.g., no console, limited memory mapping)

# Kernel failures

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

### • Kernel panic:

- Fatal error; no recovery is possible
- Typical in interrupt context
- Possibility to set automatic reboot: `panic=<secs>` kernel option on U-Boot's bootargs

### • Kernel oops:

- Non-fatal error
- May lead to a subsequent panic
- Typical in process context
- Current process is killed

# Kernel failures

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## • Kernel panic:

- Fatal error; no recovery is possible
- Typical in interrupt context
- Possibility to set automatic reboot: `panic=<secs>` kernel option on U-Boot's bootargs

## • Kernel oops:

- Non-fatal error
- May lead to a subsequent panic
- Typical in process context
- Current process is killed

# Debugging support in the kernel

## Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Kernel developers have built several debugging features into the kernel itself
- These features are not enabled by default because they create extra output and slow performance
- Some options are not supported by all architectures
- Settings are written in the `.config` file
- Some features are available only for a kernel without proprietary drivers.
  - Loading a proprietary or non-GPL-compatible LKM will set a “taint” flag in the running kernel (i.e., “tainted kernel”)

# Debugging support in the kernel

## Kernel

Intro  
Compiling  
Code

### Debugging

Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Kernel developers have built several debugging features into the kernel itself
- These features are not enabled by default because they create extra output and slow performance
- Some options are not supported by all architectures
- Settings are written in the `.config` file
- Some features are available only for a kernel without proprietary drivers.
  - Loading a proprietary or non-GPL-compatible LKM will set a “taint” flag in the running kernel (i.e., “tainted kernel”)

# Debugging support in the kernel (2)

Kernel  
Intro  
Compiling  
Code  
**Debugging**  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- **CONFIG\_DEBUG\_KERNEL**
  - It just makes other debugging options available
  - It doesn't enable any feature
- **CONFIG\_DEBUG\_SLAB**
  - Detect memory overruns and missing initializations
  - Poisoning: each byte of allocated memory is set to `0xa5` before being handled and to `0x6b` when it is freed
- **CONFIG\_DEBUG\_SPINLOCK**
  - Uninitialized spinlocks and unlocking a lock twice
- **CONFIG\_DEBUG\_SPINLOCK\_SLEEP**
  - Sleep while holding a spinlock
- **CONFIG\_INIT\_DEBUG**
  - Check for code that attempts to access initialization-time data (i.e. `__init` or `__initdata`) after initialization

# Debugging support in the kernel (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **CONFIG\_DEBUG\_INFO**

- Kernel built with full debugging information

- **CONFIG\_DEBUG\_STACKOVERFLOW**

- To detect stack overflow

- **CONFIG\_KALLSYMS**

- Kernel symbol information built into the kernel
  - To have symbolic tracebacks at oops

- **CONFIG\_IKCONFIG**

- **CONFIG\_IKCONFIG\_PROC**

- Kernel `.config` file available as `/proc/config.gz`

- **CONFIG\_DEBUG\_DRIVER**

- Turns on debugging information in the driver core

- **CONFIG\_INPUT\_EVBUG**

- Turns on verbose logging of input events

- Useful for input devices

# Oops messages

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

Unable to handle kernel NULL pointer dereference at virtual address 00000000  
printing eip:

d083a064

Oops: 0000 [#1]

SMP

CPU: 0

EIP: 0060:[<d083a064>] Not tainted

EFLAGS: 00010246 (2.6.6)

EIP is at faulty\_write+0x4/0x10 [faulty]

eax: 00000000 ebx: 00000000 ecx: 00000000 edx: 00000000

esi: cf8b2460 edi: cf8b2480 ebp: 00000005 esp: c31c5f74

ds: 007b es: 007b ss: 0068

Process bash (pid: 2086, threadinfo=c31c4000 task=cf0a6c0)

Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460

fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480

00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005

Call Trace:

[<c0150558>] vfs\_write+0xb8/0x130

[<c0150682>] sys\_write+0x42/0x70

[<c0103f8f>] syscall\_call+0x7/0xb

Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0

Oops counter.

Trust only the first one, it is more reliable

# Oops messages (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000  
printing eip:
```

```
d083a064
```

```
Oops: 0000 [#1]
```

```
SMP
```

```
CPU: 0
```

```
EIP: 0060:<d083a064> Not tainted
```

```
EFFLAGS: 00010246 (2.6.6)
```

```
EIP is at faulty_write+0x4/0x10 [faulty]
```

```
eax: 00000000 ebx: 00000000 ecx: 00000000 edx: 00000000
```

```
esi: cf8b2460 edi: cf8b2480 ebp: 00000005 esp: c31c5f74
```

```
ds: 007b es: 007b ss: 0068
```

```
Process bash (pid: 2086, threadinfo=c31c4000 task=cf0a6c0)
```

```
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
```

```
fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
```

```
00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
```

```
Call Trace:
```

```
[<c0150558>] vfs_write+0xb8/0x130
```

```
[<c0150682>] sys_write+0x42/0x70
```

```
[<c0103f8f>] syscall_call+0x7/0xb
```

```
Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

CPU and registers.

# Oops messages (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
printing eip:
d083a064
Oops: 0000 [#1]
SMP
CPU: 0
Function
EIP: 0060:[<d083a064>] Not tainted
EFLAGS: 00010246 (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
Module
eax: 00000000 ebx: 00000000 ecx: 00000000 edx: 00000000
esi: cf8b2460 edi: cf8b2480 ebp: 00000005 esp: c31c5f74
ds: 007b es: 007b ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cf0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
[<c0150558>] vfs_write+0xb8/0x130
[<c0150682>] sys_write+0x42/0x70
[<c0103f8f>] syscall_call+0x7/0xb
Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

# Oops messages (4)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
printing eip:
d083a064
Oops: 0000 [#1]
SMP
CPU: 0
EIP: 0060:<d083a064> Not tainted
EFLAGS: 00010246 (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000 ebx: 00000000 ecx: 00000000 edx: 00000000
esi: cf8b2460 edi: cf8b2480 ebp: 00000005 esp: c31c5f74
ds: 007b es: 007b ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cf0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
[<c0150558>] vfs_write+0xb8/0x130
[<c0150682>] sys_write+0x42/0x70
[<c0103f8f>] syscall_call+0x7/0xb
```

Backtrace

Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0

# Oops messages (5)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Techniques to locate the C instruction which caused an oops

<http://kerneltrap.org/node/3648>

# Debugging through printing

## Kernel

Intro  
Compiling

Code  
Debugging

Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

I/O

Platforms  
Using sysfs  
Timers

Int. handlers  
Bottom halves  
Examples

- The most popular debugging technique is debugging through `printf`
- `printf()`: kernel's version of the familiar `printf()` C library function
  - The kernel needs its own function because it runs by itself, without the help of the C library
- It can be run from almost every context
- It allows to specify the priorities ("loglevels") of messages:

```
printf(KERN_DEBUG "Here I am: %s:%i\n", __FILE__,  
__LINE__);
```

# Debugging through printing (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Linux implements a circular buffer that is `__LOG_BUF_LEN` bytes long
- From 4KB to 1MB, chosen while configuring the kernel
- This way, the system can run even without a logging process
- `printf` can be invoked from anywhere, even from an interrupt handler, with no limit on how much data can be printed

# Debugging through printing (3)

## Kernel

Intro

Compiling

Code

## Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `printk()` doesn't flush until a trailing newline is provided
- In case of kernel panic, you can manually inspect the content of the `printk` circular buffer
- If you don't have a hardware debugger:
  - See the virtual address of the buffer:  
`grep __log_buf System.map`
  - Convert it to the physical address (often just subtract `KERNELBASE = 0xc0000000`)
  - Reboot the machine
  - Use the bootloader to examine the content of the buffer

# Debugging through printing (4)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- To debug very early kernel startup code set the **CONFIG\_EARLY\_PRINTK** kernel option

# Debugging through printing (5)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- There are 8 possible loglevels associated to messages
- Defined in `<linux/kernel.h>`
- Each string represents a number ranging from 0 to 7, with smaller values representing higher priorities
- The default log level is equal to the `DEFAULT_MESSAGE_LOGLEVEL` variable specified in `kernel/printk.c`

# Debugging through printing (5)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- There are 8 possible loglevels associated to messages
- Defined in `<linux/kernel.h>`
- Each string represents a number ranging from 0 to 7, with smaller values representing higher priorities
- The default log level is equal to the `DEFAULT_MESSAGE_LOGLEVEL` variable specified in `kernel/printk.c`

# Debugging through printing (6)

Kernel  
Intro  
Compiling  
Code  
**Debugging**

Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

Timers  
Int. handlers  
Bottom halves  
Examples

- KERN\_EMERG** Emergency: system is unusable
- KERN\_ALERT** Serious problem: action must be taken immediately
- KERN\_CRIT** Critical condition, usually related to hardware or software failure
- KERN\_ERR** Used for error conditions, usually related to hardware difficulties
- KERN\_WARNING** Used to warn about problematic situations that are not serious
- KERN\_NOTICE** Normal situations that require notification
- KERN\_INFO** Informational messages. Many drivers print information about the hardware found
- KERN\_DEBUG** Used only for debugging

# Debugging through printing (7)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Pros:

- Easy to use
- No additional tools needed

- Cons:

- Not interactive
- Need to rebuild kernel/module

# How can we read messages ?

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- If both `klogd` and `syslogd` are running, messages are appended to `/var/log/messages`:

```
tail -f /var/log/messages
```

- If `klogd` isn't running, use the `dmesg` shell command

- If `syslogd` isn't running, use

```
cat /proc/kmsg
```

# How can we change the default loglevel ?

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Through the `sys_syslog` system call
- Kill `klogd` and restart it with the `-c` option
  - Notice: with the `-f` option of `klogd` you can write messages into a file

# How can we change the default loglevel ?

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Through the `sys_syslog` system call
- Kill `klogd` and restart it with the `-c` option
  - Notice: with the `-f` option of `klogd` you can write messages into a file

# How can we change the default loglevel ? (2)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Use the `/proc/sys/kernel/printk` (from klogd 2.1.31)
  - Read the content through  
`cat /proc/sys/kernel/printk`
  - The file contains 4 integers:
    - Current loglevel
    - Default level for messages that lack an explicit loglevel
    - Minimum allowed loglevel
    - Boot-time default loglevel
  - We are interested in the first two values: console loglevel and default loglevel
  - `echo 5 > /proc/sys/kernel/printk` displays only messages with loglevel from 0 to 4 on console
  - `echo 8 > /proc/sys/kernel/printk` displays all messages on console

# How can we change the console loglevel ?

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Console loglevel can be changed through kernel boot parameters:
  - `debug`: sets console loglevel to 10
  - `quiet`: sets console loglevel to 4
  - `loglevel=`: sets a specific value for console loglevel

# Turning messages on and off

In our header:

```
# undef EVI_DEBUG /* undef it, just in case */
#ifndef EVI_DEBUG_ON
    # define EVI_DEBUG(fmt, args...) \
        printk(KERN_DEBUG "<module>: " fmt, ##args)
#else
    # define EVI_DEBUG(fmt, args...) /* not debugging */
#endif
```

In our Makefile:

```
# Comment/uncomment the following line
# to enable/disable debugging
DEBUG = y

# Note: "-O" is needed to expand inlines
ifeq ($(DEBUG),y)
    CFLAGS += -O -g -EVI_DEBUG_ON
else
    CFLAGS += -O2
endif
```

# Other debugging techniques

## Kernel

Intro  
Compiling

Code

## Debugging

Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Debugging by querying

- Use an entry in `/proc`
- Use an entry in `/sys`
- Use an ioctl on a char device

- Debugging by watching

- Use `strace` to see system calls issued by a user-space program
- Use ftrace to trace kernel functions

# Other debugging techniques

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Debugging by querying

- Use an entry in /proc
- Use an entry in /sys
- Use an ioctl on a char device

- Debugging by watching

- Use `strace` to see system calls issued by a user-space program
- Use ftrace to trace kernel functions

# Ftrace

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- New infrastructure that can be used for debugging or analyzing latencies and performance issues in the kernel
- Can be used to trace any kernel function!
- Developed by Steven Rostedt. Merged in 2.6.27.
- Very well documented in [Documentation/trace/ftrace.txt](#)
  - For the very first time in kernel history, it was documented even before it was included
- Negligible overhead when tracing is not enabled at run-time.
- Tracing information available through the debugfs virtual fs (i.e., `CONFIG_DEBUG_FS`)

# Ftrace: usage

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `mount -t debugfs nodev /sys/kernel/debug`
- `cd /sys/kernel/debug/tracing`
- `echo sys_nanosleep hrtimer_interrupt > set_ftrace_filter`
- `echo function > current_tracer`
- `echo 1 > tracing_on`
- `usleep 1`
- `echo 0 > tracing_on`
- `cat trace`

# Kprobes (kernel probes)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Kprobes allow to define **handlers** that are executed every time the kernel executes a specific function
- They allow to inject object code at run-time in the kernel
- They work like a breakpoint, by executing code when the kernel reaches a certain address

# Kprobes: usage

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
static int my_handler(struct kprobe *p, struct pt_regs *regs)
{
    /* ... */
}

static struct kprobe my_kp = {
    .pre_handler = my_handler,
    .symbol_name = "scheduler_timeout",
};

static int __init my_kprobe_init(void)
{
    int ret;

    ret = register_kprobe(&my_kp);
    if (ret < 0) {
        printk (KERN_ERR "%s: error %d\n", __FUNCTION__, ret);
        return ret;
    }
    return 0;
}

static void __exit my_kprobe_exit(void)
{
    unregister_kprobe(&my_kp);
}
```

# Kernel debuggers: GDB and KDB

## Kernel

Intro  
Compiling

Code  
**Debugging**

Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem

proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms  
Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Linus doesn't believe in interactive debuggers
  - He fears that they lead to poor fixes, by patching symptoms rather than real problems
- GDB
  - Only permits to read value of variables
  - Not able to modify kernel data
  - Not possible to set breakpoints or watchpoints
- KDB
  - Non-official patch
  - Only works on x86 and IA64 platforms
  - <http://oss.sgi.com/projects/kdb>

# Kernel debuggers: GDB and KDB

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Linus doesn't believe in interactive debuggers
  - He fears that they lead to poor fixes, by patching symptoms rather than real problems
- GDB
  - Only permits to read value of variables
  - Not able to modify kernel data
  - Not possible to set breakpoints or watchpoints
- KDB
  - Non-official patch
  - Only works on x86 and IA64 platforms
  - <http://oss.sgi.com/projects/kdb>

# Kernel debuggers: GDB and KDB

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Linus doesn't believe in interactive debuggers
  - He fears that they lead to poor fixes, by patching symptoms rather than real problems
- GDB
  - Only permits to read value of variables
  - Not able to modify kernel data
  - Not possible to set breakpoints or watchpoints
- KDB
  - Non-official patch
  - Only works on x86 and IA64 platforms
  - <http://oss.sgi.com/projects/kdb>

# KGDB

- Official Linux debugger
- Now integrated in the official kernel
- The debugger runs on a separate machine than the debugged kernel:
  - Host: normal gdb
  - Target:
    - Kernel built with debug information
    - Kernel built with kgdb support
    - KGDB support enabled through kernel boot parameters: `kgdboc=ttyS1,115200 kgdbwait`  
The order of these parameters is important: the serial port must be specified before we set the breakpoint.
- Host and target are connected through serial (default) or ethernet
- Some features (e.g., KGDB over Ethernet) are available only for some platforms
- Very early kernel startup code cannot be debugged with KGDB

Kernel  
Intro  
Compiling  
Code  
**Debugging**  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# KGDB: common kernel breakpoints

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Two common GDB breakpoints are:

- **b panic**: invokes the debugger when a non-recoverable failure occurs
- **b sys\_sync**: useful to trap into the debugger by typing the **sync** command on console

# Hardware-assisted debuggers: Lauterbach

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

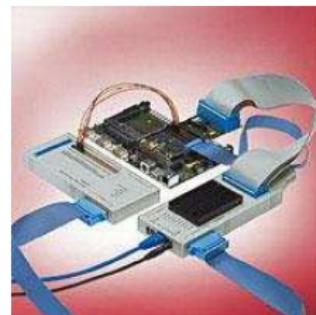
Timers

Int. handlers

Bottom halves

Examples

- Lauterbach Trace32 <http://www.lauterbach.com>



# How to install T32 USB on Ubuntu

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Insert the Lautebach T32 CD and mount it:

- `mount /mnt/cdrom`

2. Create the directory on your filesystem:

- `mkdir /home/t32 /home/t32/bin`

3. Copy the files:

- `cd /home/t32`

- `cp -r /mnt/cdrom/files/* .`

- `cp -r /mnt/cdrom/bin/pc_linux ./bin`

4. Change file permissions, extract compressed files and convert files to Linux format:

- `chmod -R u+w *`

- `/home/t32/bin/pc_linux/filecvt /home/t32`

# How to install T32 USB on Ubuntu (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 5. Copy the configuration file:

■ `cp /home/t32/bin/pc_linux/config.t32 /home/t32/`

## 6. Open `/home/t32/config.t32`

■ Uncomment the following lines:

- PBI=
- USB

■ Comment the following lines:

- ;SCREEN=
- ;FONT=DEC
- ;FONT=SMALL

## 7. Install Acrobat Reader

# How to install T32 USB on Ubuntu (3)

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

8. Open `.bashrc` and set the following variables:

- `export ACROBAT_PATH=/usr/lib/Adobe/Acrobat7.0`
- `export T32SYS=/home/t32`
- `export T32TMP=/tmp`
- `export T32ID=T32`

9. Copy the Trace32 plugin to the plugins folder of Acrobat Reader:

- `cp /home/t32/bin/pc_linux/trace32.api`  
`$ACROBAT_PATH/Reader/intellinux/plug_ins`

10. Udev configuration:

- `cd /home/t32/bin/pc_linux/udev.conf`
- `sudo cp 10-Lauterbach.rules /etc/udev/rules.d/`

# How to install T32 USB on Ubuntu (4)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

11. Create the following script:

```
#!/bin/bash
cd /home/t32/fonts
mkfontdir .
xset +fp /home/t32/fonts
xset fp rehash
cd ..
./bin/pc_linux/t32marm
```

# How to debug the kernel using T32

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Configure the kernel with `CONFIG_DEBUG_INFO`, `CONFIG_KALLSYMS` and `CONFIG_KALLSYMS_ALL`
2. Compile the kernel for normal images (i.e. no `uImage`)
3. The `vmlinux` with all symbols included is put in the Linux main directory (not in `arch/.../boot/`) and is around 20MB

# How to debug the kernel using T32 (2)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

On the Lauterbach script ...

4. Set the path of Linux sources:

```
&linux_source="/<linux_path>"
```

5. Set the path of Linux image:

```
&linux_elf="/<linux_path>/vmlinuz"
```

6. Load kernel source files:

```
y.sourcepath.setbasedir &linux_source
```

7. Open a code window:

```
Data.ListH1
```

8. Load kernel symbols:

```
Data.LOAD.Elf &linux_elf /gnu /nocode
```

# How to debug the kernel using T32 (3)

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

Still on the Lauterbach script ...

9. Initialize Linux awareness:

```
TASK.CONFIG /<t32_path>/demo/arm/kernel/linux/linux  
MENU.ReProgram /<t32_path>/demo/arm/kernel/linux/linux  
HELP.FILTER.Add rtoslinux
```

10. Finally, set:

```
TASK.sYmbol.Option MMUSCAN OFF  
sYmbol.AutoLoad.CHECKLINUX "do "+os.ppd()+"autoload "
```

# Testing tools: LTP

Kernel

Intro

Compiling

Code

**Debugging**

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## ● Linux Test Project (LTP)

- Project started by SGI and maintained by IBM to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux
- The testsuite contains a collection of tools for testing the Linux kernel and related features
- <http://ltp.sourceforge.net>

# Linked lists

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

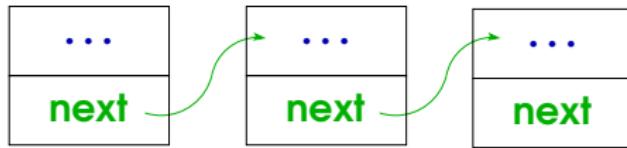
- Data structure that stores a certain amount of **nodes**
- The nodes can be dynamically created, added and removed at runtime
  - Number of nodes unknown at compile time
  - Different from array
- For this reason, the nodes are linked together
  - Each node contains at least one pointer to another element

# Singly linked lists

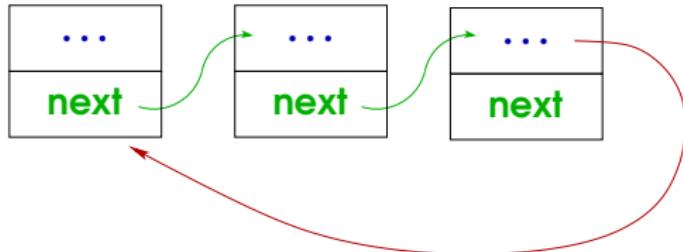
Kernel  
Intro  
Compiling  
Code  
Debugging  
**Linked lists**  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

```
struct list_element {  
    int data;  
    struct list_element *next;  
};
```

- Singly linked list:



- Circular singly linked list:

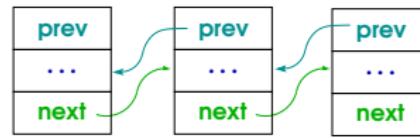


# Doubly linked lists

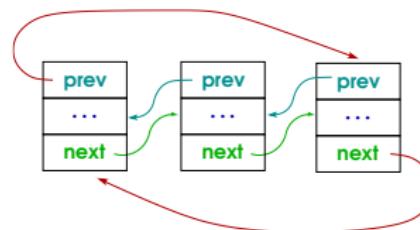
Kernel  
Intro  
Compiling  
Code  
Debugging  
**Linked lists**  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

```
struct list_element {  
    int data;  
    struct list_element *next;  
    struct list_element *prev;  
};
```

- Doubly linked list:



- Circular doubly linked list:



# Kernel's linked list implementation

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Circular doubly linked list
- No head pointer: does not matter where you start...
  - All individual nodes are called *list heads*

- Declared in `linux/list.h`
- Data structure:

```
struct list_head {  
    struct list_head* next;  
    struct list_head* prev;  
};
```

- No locking: your responsibility to implement a locking scheme

# Defining linked lists

Kernel

Intro  
Compiling

Code  
Debugging

**Linked lists**

Scheduling

Linux filesystem  
/dev filesystem

proc filesystem  
sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Include the `list.h` file:

```
#include <linux/list.h>
```

2. Embed a `list_head` inside your structure:

```
struct my_node {  
    struct list_head klist;  
    /* Data */  
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

# Defining linked lists

Kernel

Intro  
Compiling

Code  
Debugging

**Linked lists**

Scheduling

Linux filesystem  
/dev filesystem

proc filesystem  
sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Include the `list.h` file:

```
#include <linux/list.h>
```

2. Embed a `list_head` inside your structure:

```
struct my_node {  
    struct list_head klist;  
    /* Data */  
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

# Defining linked lists

Kernel

Intro  
Compiling

Code  
Debugging

Linked lists

Scheduling

Linux filesystem  
/dev filesystem

proc filesystem  
sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Include the `list.h` file:

```
#include <linux/list.h>
```

2. Embed a `list_head` inside your structure:

```
struct my_node {  
    struct list_head klist;  
    /* Data */  
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

# Defining linked lists

Kernel

Intro  
Compiling

Code  
Debugging  
**Linked lists**

Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers

Modules

Memory

Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

Timers

Int. handlers

Bottom halves  
Examples

1. Include the `list.h` file:

```
#include <linux/list.h>
```

2. Embed a `list_head` inside your structure:

```
struct my_node {  
    struct list_head klist;  
    /* Data */  
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

# Using linked lists

Kernel

Intro  
Compiling  
Code  
Debugging  
**Linked lists**

Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));  
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;  
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;  
list_for_each (g, &my_list) {  
    /* g points to a klist field inside  
     * the next my_node structure */  
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

# Using linked lists

## Kernel

Intro  
Compiling  
Code  
Debugging  
**Linked lists**

Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));  
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;  
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;  
list_for_each (g, &my_list) {  
    /* g points to a klist field inside  
     * the next my_node structure */  
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

# Using linked lists

## Kernel

Intro  
Compiling  
Code  
Debugging  
**Linked lists**

Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem

sys filesystem  
Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));  
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;  
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;  
list_for_each (g, &my_list) {  
    /* g points to a klist field inside  
     * the next my_node structure */  
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

# Using linked lists

## Kernel

Intro  
Compiling  
Code  
Debugging  
**Linked lists**

Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem

sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));  
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;  
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;  
list_for_each (g, &my_list) {  
    /* g points to a klist field inside  
     * the next my_node structure */  
}
```

- Knowing the structure containing a **klist\*** **h**:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

# Using linked lists: Example

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

How to remove from the linked list the node having value 7:

```
struct my_node {  
    struct list_head klist;  
    int value;  
};  
  
struct list_head my_list;  
  
struct list_head *h;  
list_for_each_safe(h, &my_list)  
    if ((list_entry(h, struct my_node, klist))->value == 7)  
        list_del(h);
```

# Using linked lists (3)

Kernel  
Intro  
Compiling  
Code  
Debugging  
**Linked lists**  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Add a new node after the given list head:  
`list_add_tail();`
- Delete a node and reinitialize it: `list_del_init();`
- Move one node from one list to another: `list_move();`,  
`list_move_tail();`
- Check if a list is empty: `list_empty();`
- Join two lists: `list_splice();`
- Iterate without prefetching: `_list_for_each();`
- Iterate backward: `list_for_each_prev();`
- If your loop may delete nodes in the list:  
`list_for_each_safe();`

# Unix structure

## Kernel

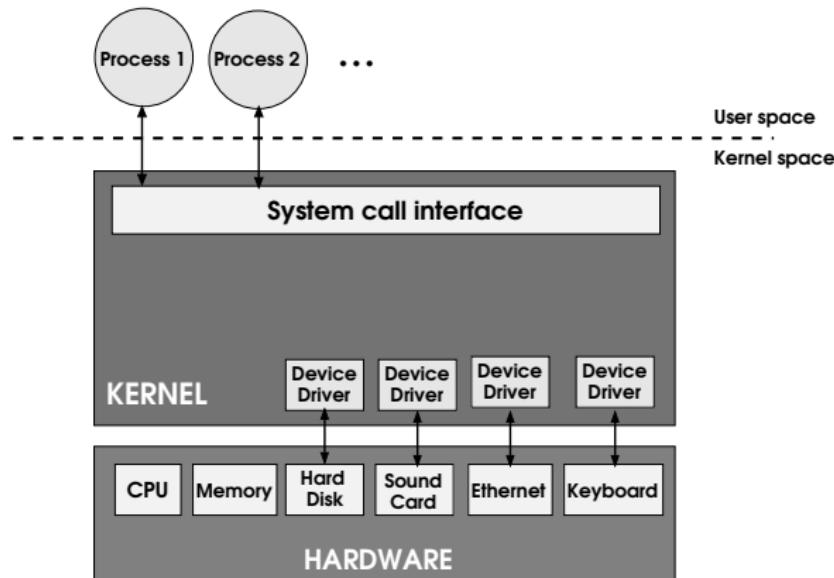
- Intro
- Compiling
- Code
- Debugging
- Linked lists

## Scheduling

- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples



# Virtualization

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The OS provides two virtualizations
  1. Virtualized processor → Process Scheduling
  2. Virtualized memory → Memory Management

# Processes

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Program:** description of an algorithm [static]

- **Process:** program in execution [dynamic]

- One of the fundamental abstractions in OSs
- It includes:

- Set of resources (e.g., open files)
- Pending signals
- Processor state
- Address space
- One or more threads of execution
- Data section containing global variables

# Processes

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Program**: description of an algorithm [static]

- **Process**: program in execution [dynamic]

- One of the fundamental abstractions in OSs
- It includes:

- Set of resources (e.g., open files)
- Pending signals
- Processor state
- Address space
- One or more threads of execution
- Data section containing global variables

# Threads

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

## Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Thread**: entity of activity within a process
- Threads of the same process share memory address space, open files and other resources
- In traditional Unix, each process consists of one thread
- To Linux, a thread is a special kind of process
  - It is a process sharing certain resources
  - It has a unique `task_struct`

# Process classification

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. I/O-bound

- Spend much time submitting and waiting on I/O requests
- Typical process waiting for human input
- Often runnable
- Run for short durations
- Example: text editor

## 2. Processor-bound

- Spend much time doing computation
- Typically, run until preempted
- Example: video encoder

# Process classification

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. I/O-bound

- Spend much time submitting and waiting on I/O requests
- Typical process waiting for human input
- Often runnable
- Run for short durations
- Example: text editor

## 2. Processor-bound

- Spend much time doing computation
- Typically, run until preempted
- Example: video encoder

# The scheduler

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Scheduler:** kernel program that selects the “best” process to run

- Important part of the kernel
- Basis of *multitasking*
- Responsible for best utilizing the system
- Gives the illusion of multiple processes running simultaneously

- **Scheduling policy:** behaviour of the scheduler that determines what runs when

- Complex algorithms to satisfy conflicting goals
  1. Fast response time (low latency)
  2. Maximal system utilization (high throughput)
  3. No starvations
- Unix variants (Linux included) tend to explicitly favor I/O-bound and interactive processes

# The scheduler

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Scheduler:** kernel program that selects the “best” process to run

- Important part of the kernel
- Basis of *multitasking*
- Responsible for best utilizing the system
- Gives the illusion of multiple processes running simultaneously

- **Scheduling policy:** behaviour of the scheduler that determines what runs when

- Complex algorithms to satisfy conflicting goals
  1. Fast response time (low latency)
  2. Maximal system utilization (high throughput)
  3. No starvations
- Unix variants (Linux included) tend to explicitly favor I/O-bound and interactive processes

# Multitasking OSs

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

## 1. Cooperative multitasking

- The process voluntarily decides to stop running
- The act of a process voluntarily suspending is called **yielding**
- A process can monopolize the processor
- Example: Mac OS 9

## 2. Preemptive multitasking

- The scheduler decides when a process is to cease running
- The act of changing the current running process is called **preemption**
- The time a process runs before it is preempted is called **timeslice**
- Examples: Unix and Linux

# Multitasking OSs

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

## 1. Cooperative multitasking

- The process voluntarily decides to stop running
- The act of a process voluntarily suspending is called **yielding**
- A process can monopolize the processor
- Example: Mac OS 9

## 2. Preemptive multitasking

- The scheduler decides when a process is to cease running
- The act of changing the current running process is called **preemption**
- The time a process runs before it is preempted is called **timeslice**
- Examples: Unix and Linux

# Preemptive multitasking

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 2.1 User preemption

- Provided by almost all modern general-purpose OSs
- Preemptible processes
- A process running in user-level can be preempted at any time (regardless of what it is doing)
- Kernel code runs until completion

## 2.2 Kernel preemption

- A process can be preempted even when it executes kernel code (i.e. exceptions or system call handlers)
- Provided by Linux

# Preemptive multitasking

## Kernel

Intro  
Compiling

Code  
Debugging

Linked lists  
Scheduling

Linux filesystem  
/dev filesystem

proc filesystem  
sys filesystem

Syscalls  
Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 2.1 User preemption

- Provided by almost all modern general-purpose OSs
- Preemptible processes
- A process running in user-level can be preempted at any time (regardless of what it is doing)
- Kernel code runs until completion

## 2.2 Kernel preemption

- A process can be preempted even when it executes kernel code (i.e. exceptions or system call handlers)
- Provided by Linux

# Scheduling in Unix

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists

## Scheduling

- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

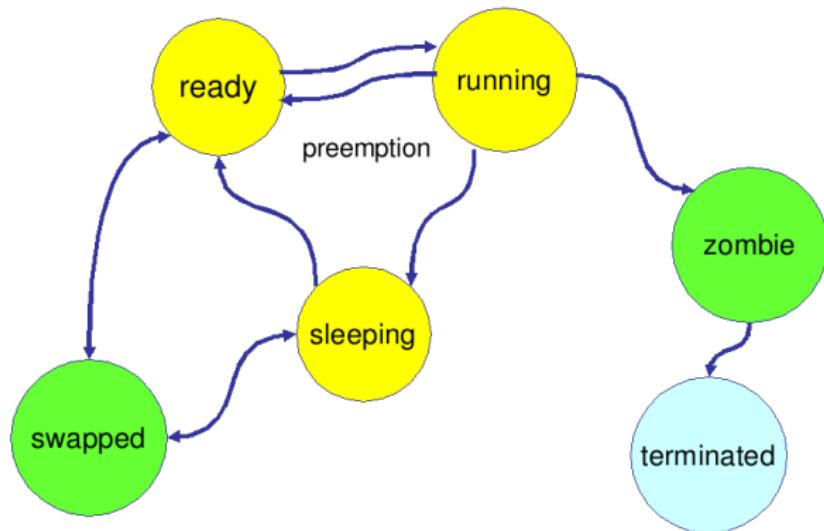
- Modules
- Memory
- Concurrency
- Char devices
- I/O

- Platforms
- Using sysfs

## Timers

- Int. handlers
- Bottom halves

- Examples



# Process creation

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Processes are created only through the `fork()` syscall
  - A process (“parent”) creates another process (“child”)
  - Hierarchical structure
  - A process can have several children
  - A process has exactly one parent
- With `fork()` the memory space of the parent is copied into the memory space of the child
  - They are not shared
  - Very expensive operation

# PIDs

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- A PID (“Process IDentificator”) is associated to each process
  - The init process is the first process created and has PID 1

# How fork() works

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
int pid = fork();
if (pid == -1) {
    /* error */
} else if (pid == 0) {
    /* Code of the child */
} else {
    /* Code of the parent */
}
```

# How fork() works: example

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main (void)
{
    int pid = fork ();
    if (pid < 0)
        fprintf(stderr, "Error in fork()\n");
    else if (pid == 0)
        printf ("I'm the child. My pid is %d Parent pid is %d\n", getpid(), getppid());
    else
        printf ("I'm the parent. My pid is %d Child pid is %d\n", getpid(), pid);
    return 0;
}
```

We cannot assume any order of execution!

# Copy-on-write

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- A child process can start executing a new program through the `exec` family of functions
  - It creates a new address space and loads a new program into it
  - If called after a `fork()`, the **Copy-On-Write** technique prevents copying of data that will be overwritten

# Process termination

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- A parent can know the termination status of the child through the `wait()` family of functions
- A child can communicate a termination code to its parent through the `exit()` syscall
  - The child is put in the `TASK_ZOMBIE` state until the parent calls a `wait()` function.
  - If the parent terminates, the process becomes child of another process in the same thread group
  - If no other process exists in the thread group, the process becomes child of the `init` process. The `init` process periodically calls `wait()` on its children.
  - Note: if the parent exists but doesn't call `wait()`, then the child process remains zombie with space in process table

# Processes on Linux

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- On Linux each process is characterized by:
  - A state
  - A scheduling policy
  - A (static or dynamic) priority

# Process states on Linux

## Kernel

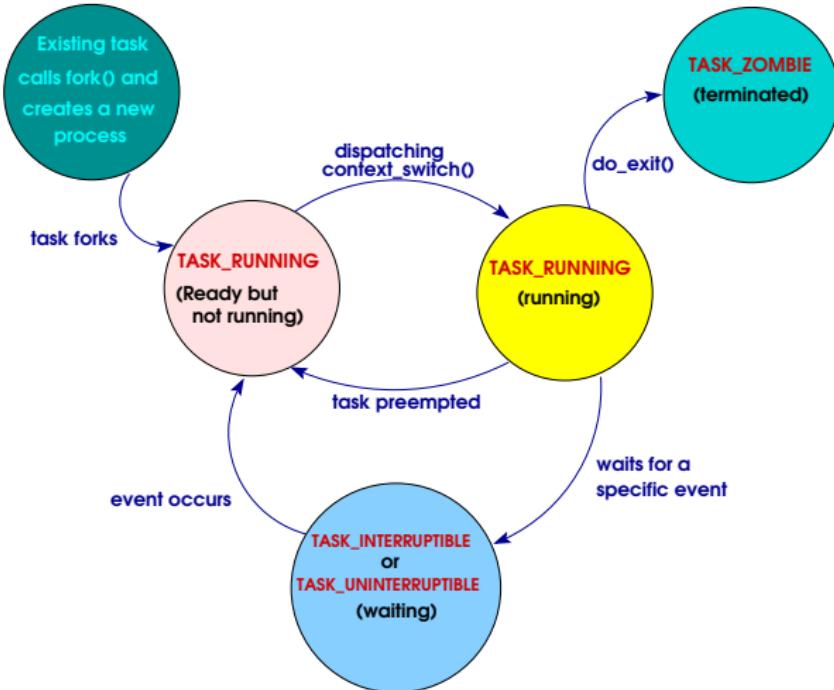
- Intro
- Compiling
- Code
- Debugging
- Linked lists

## Scheduling

- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples



# Process descriptor

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The kernel keeps information for each process in a table called **process descriptor**
- The process descriptor is a table of type **task\_struct**
  - Composed by more than 100 fields
  - Around 1.7 KB on 32-bit machines
  - See **include/linux/sched.h**
- The macro **current** points to the descriptor of the process currently in execution
  - Defined in **asm/current.h**
  - Architecture-dependent mechanism
  - Yields a pointer to **struct task\_struct**
  - Used when the kernel executes on behalf of a process

# Process Identifier

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- PID: Unique identification of the process
- Type `pid_t` (typically `int`)
- `cat /proc/sys/kernel/pid_max` to know the maximum value
  - Maximum number of processes that may exist concurrently
  - Typically, 32,768 (`short int`)

# Process states on Linux

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

The current state of each process is stored in its descriptor:

|                                                                     |                                   |
|---------------------------------------------------------------------|-----------------------------------|
| In execution on a CPU or<br>ready (runnable but not in execution)   | <code>TASK_RUNNING</code>         |
| Sleeping (blocked)<br>Waiting for an event or a signal              | <code>TASK_INTERRUPTIBLE</code>   |
| Waiting for an event                                                | <code>TASK_UNINTERRUPTIBLE</code> |
| Stopped                                                             | <code>TASK_STOPPED</code>         |
| Terminated but the parent has<br>not yet issued a <code>wait</code> | <code>TASK_ZOMBIE</code>          |

Process's state can be changed through  
`set_task_state(task, state)` and  
`set_current_state(state)`

# Scheduling policies

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

### ● Normal policies:

- **SCHED\_OTHER**: standard round-robin time-sharing policy
- **SCHED\_BATCH**: for cpu-intensive non-interactive “batch” processes
- **SCHED\_IDLE**: for very low-priority background tasks; small penalty applied.

### ● POSIX fixed-priority “real-time” policies:

- **SCHED\_FIFO**: w/out timeslice
- **SCHED\_RR**: with timeslice (use `sched_rr_get_interval()` to know the length)  
Use `SCHED_RESET_ON_FORK` in `sched_setscheduler` if this policy shouldn't be inherited by children. Example:  
`sched_setscheduler(pid,  
SCHED_FIFO|SCHED_RESET_ON_FORK, &param)`

# Scheduling policies

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Normal policies:

- **SCHED\_OTHER**: standard round-robin time-sharing policy
- **SCHED\_BATCH**: for cpu-intensive non-interactive “batch” processes
- **SCHED\_IDLE**: for very low-priority background tasks; small penalty applied.

- POSIX fixed-priority “real-time” policies:

- **SCHED\_FIFO**: w/out timeslice
- **SCHED\_RR**: with timeslice (use `sched_rr_get_interval()` to know the lenght)

Use **SCHED\_RESET\_ON\_FORK** in `sched_setscheduler` if this policy shouldn't be inherited by children. Example:

```
sched_setscheduler(pid,  
SCHED_FIFO|SCHED_RESET_ON_FORK, &param)
```

# Process priorities

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **SCHED\_FIFO** and **SCHED\_RR**:

- Static priority set by the user
- Between 1 (lowest) and 99 (highest)

- **SCHED\_OTHER** and **SCHED\_BATCH**:

- Dynamic priority
- Based on **nice** value (set by the user)
- Increased for each time quantum the process is denied execution

- **SCHED\_IDLE**: no priorities

# Process priorities

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `SCHED_FIFO` and `SCHED_RR`:

- Static priority set by the user
- Between 1 (lowest) and 99 (highest)

- `SCHED_OTHER` and `SCHED_BATCH`:

- Dynamic priority
- Based on `nice` value (set by the user)
- Increased for each time quantum the process is denied execution

- `SCHED_IDLE`: no priorities

# Process priorities

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **SCHED\_FIFO** and **SCHED\_RR**:

- Static priority set by the user
- Between 1 (lowest) and 99 (highest)

- **SCHED\_OTHER** and **SCHED\_BATCH**:

- Dynamic priority
- Based on **nice** value (set by the user)
- Increased for each time quantum the process is denied execution

- **SCHED\_IDLE**: no priorities

# Process status and priorities

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Information about running processes can be shown through:
  - `top`
  - `ps aux`

# Getting/setting scheduling params from C

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
#include <sched.h>

struct sched_param {
    ...
    int sched_priority;
    ...
};

// Set policy and priority:
int sched_setscheduler(pid_t pid,
                      int policy,
                      const struct sched_param *param);

// Get policy:
int sched_getscheduler(pid_t pid);

// Set priority:
int sched_setparam(pid_t pid,
                   const struct sched_param *param);

// Get priority:
int sched_getparam(pid_t pid, struct sched_param *param);
```

# Getting/setting scheduling affinity from C

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
#define _GNU_SOURCE
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t cpusetsize,
                      cpu_set_t *mask);

int sched_getaffinity(pid_t pid, size_t cpusetsize,
                      cpu_set_t *mask);

//Example:
cpu_set_t cs;
CPU_ZERO(&cs);
CPU_SET(1, &cs);
sched_setaffinity(pid, sizeof(cpu_set_t), &cs);
```

# Getting/setting dynamic priority from console

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The command `nice` allows to specify the dynamic priority when starting a program:  
`nice -n <priority> <command> <args>`
  - Niceness ranges from -20 (highest priority) to 19 (lowest priority)
- 
- The command `renice` allows to change the dynamic priority of a running process:  
`renice [-n] <new priority> [[-p] pid ...] [-g pgrp ...]  
[-u user ...]`

# Getting/setting dynamic priority from console

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The command `nice` allows to specify the dynamic priority when starting a program:  
`nice -n <priority> <command> <args>`
  - Niceness ranges from -20 (highest priority) to 19 (lowest priority)
- 
- The command `renice` allows to change the dynamic priority of a running process:  
`renice [-n] <new priority> [[-p] pid ...] [-g pgrp ...]  
[-u user ...]`

# Getting/setting scheduling params from console

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Get min/max allowed priorities: `chrt -m`
- Get process current params: `chrt -p <pid>`
- Set a new priority: `chrt -p <priority> <pid>`
- Set `SCHED_BATCH`: `chrt -b -p 0 <pid>`
- Set `SCHED_OTHER`: `chrt -o -p 0 <pid>`
- Set `SCHED_FIFO`: `chrt -f -p [1..99] <pid>`
- Set `SCHED_RR`: `chrt -r -p [1..99] <pid>`
- Set affinity:

```
taskset 0x00000001 <command> <args>
```

```
taskset -p <pid> 0x00000001
```

# Getting/setting scheduling params from console

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists

## Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Get min/max allowed priorities: `chrt -m`
- Get process current params: `chrt -p <pid>`
- Set a new priority: `chrt -p <priority> <pid>`
- Set `SCHED_BATCH`: `chrt -b -p 0 <pid>`
- Set `SCHED_OTHER`: `chrt -o -p 0 <pid>`
- Set `SCHED_FIFO`: `chrt -f -p [1..99] <pid>`
- Set `SCHED_RR`: `chrt -r -p [1..99] <pid>`
- Set affinity:

```
taskset 0x00000001 <command> <args>
```

```
taskset -p <pid> 0x00000001
```

# Scheduling threads

Kernel

Intro

Compiling

Code

Debugging

Linked lists

**Scheduling**

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- For threads there are similar mechanisms:

- `pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);`
- `pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);`
- `pthread_setschedprio(pthread_t thread, int prio);`
- `pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset);`
- `pthread_getaffinity_np(pthread_t thread, size_t cpusetsize, cpu_set_t *cpuset);`

# Introduction to Linux filesystem

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

**Linux filesystem**

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- In Linux there are several kinds of file:

- Regular files
- Directories
- Symbolic links
- FIFOs
- Sockets
- Char device files
- Block device files

# POSIX

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling

## Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- POSIX comprises a series of standards from IEEE
- IEEE is the Institute of Electrical and Electronics Engineers
  - Non-profit professional association
  - <http://www.ieee.org>
- POSIX goal: create a portable operating system standard
  - Roughly based on Unix
  - POSIX was created to resemble the API of earlier Unix systems
  - On Unix systems POSIX-defined calls have a strong correlation to the system calls
  - Systems far from Unix (e.g. Windows NT) offer POSIX-compliant libraries

# POSIX API

## Kernel

Intro  
Compiling

Code  
Debugging

Linked lists  
Scheduling

## Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- A common programming model is used for both regular files and device files
- This programming model is specified by the POSIX standard
- All Unix kernels (Linux included) have a software layer that handles all system calls related to a file
  - `open()`
  - `close()`
  - `read()`
  - `write()`
  - `ioctl()`
  - ...

# POSIX API (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- This model hides the differences between device files and regular files
  - When a process accesses a regular file, it is accessing data blocks on a disk partition through a filesystem
  - When a process accesses a device file, it is just driving a hardware device
- ...but the system calls used are the same!

# The `/dev` filesystem

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

`/dev` filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Device files represent I/O devices
  - Each supported I/O device has one or more corresponding device files
- Generally located in the `/dev` directory
- Created by using the `mknod` command, or dynamically by the `udev` daemon

# Class of devices

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- There are 3 classes of devices:
  - Block devices
  - Char devices
  - Network interfaces

# Class of devices (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## • Block devices:

- Allow to access disk-like devices, in which data can be accessed by block number
- In most Unix systems, a block device can transfer one or more blocks at a time
  - Usually 512 bytes or another power of two

## • Character devices:

- Allow to access almost all other devices, in which data can be read and written as byte streams
- Random accesses are usually not feasible on char devices

## • Network cards:

- Special devices that do not have a device file
- Managed by a network interface identified through a unique name (such as `eth0`)

# Class of devices (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Block devices:

- Allow to access disk-like devices, in which data can be accessed by block number
- In most Unix systems, a block device can transfer one or more blocks at a time
  - Usually 512 bytes or another power of two

- Character devices:

- Allow to access almost all other devices, in which data can be read and written as byte streams
- Random accesses are usually not feasible on char devices

- Network cards:

- Special devices that do not have a device file
- Managed by a network interface identified through a unique name (such as `eth0`)

# Class of devices (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Block devices:

- Allow to access disk-like devices, in which data can be accessed by block number
- In most Unix systems, a block device can transfer one or more blocks at a time
  - Usually 512 bytes or another power of two

- Character devices:

- Allow to access almost all other devices, in which data can be read and written as byte streams
- Random accesses are usually not feasible on char devices

- Network cards:

- Special devices that do not have a device file
- Managed by a network interface identified through a unique name (such as `eth0`)

# Major and minor numbers

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- All information needed by the filesystem to handle a file is included in a file's descriptor called **inode**
- A device file is identified by a triplet (boolean, integer, integer) stored in the file's inode
  - The boolean determines whether the file is a character device file or a block device file
  - The two integers are the **major** and **minor device numbers**

# Major and minor numbers (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Major number:

- Traditionally it identifies the **driver** associated with the device (i.e. one-major-one-driver principle)
- Modern Linux kernels allow multiple drivers to share major numbers
- The same major number is used with different meanings for char and block devices

- Minor number:

- Used by the kernel to determine exactly which **device** is referred to

# Major and minor numbers (2)

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Major number:

- Traditionally it identifies the **driver** associated with the device (i.e. one-major-one-driver principle)
- Modern Linux kernels allow multiple drivers to share major numbers
- The same major number is used with different meanings for char and block devices

- Minor number:

- Used by the kernel to determine exactly which **device** is referred to

# Major and minor numbers (3)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
**/dev filesystem**

proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

Timers

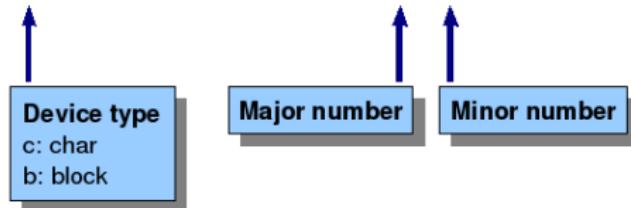
Int. handlers

Bottom halves

Examples

- With the `ls -l` command we can see the three values in the device file's inode
- On my PC with `ls -l /dev/hd* /dev/lp*` command, I get

```
brw-rw---- 1 root disk    3, 0 2006-02-09 17:50 /dev/hda
brw-rw---- 1 root disk    3, 1 2006-02-09 17:50 /dev/hda1
brw-rw---- 1 root disk    3, 2 2006-02-09 17:50 /dev/hda2
brw-rw---- 1 root cdrom   22, 0 2006-02-09 17:50 /dev/hdc
crw-rw---- 1 root lp      6, 0 2006-02-09 17:50 /dev/lp0
```



- Registered devices are shown in `/proc/devices`

# File operations

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Linux manages inodes through the `inode` structure
- One of its fields is a pointer to another important structure: the `struct file_operations`
- This structure contains pointers to low level functions that implement the hardware (or filesystem) dependent operations of each of the file's system calls:

```
struct file_operations {  
    loff_t (*llseek) (struct file *, loff_t, int);  
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);  
    int (*open) (struct inode *, struct file *);  
    int (*release) (struct inode *, struct file *);  
    /* ...and many other fields... */  
};
```

# File operations (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Each driver must implement the proper file operations for its device file and store the pointers to these functions into a `struct file_operations`
- Linux ensures that the inode of a device file includes a pointer to the `struct file_operations` filled by the device driver
- Each system call acting on that device file triggers the execution of a file operation provided by the device driver

# The proc filesystem

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling

Linux filesystem  
`/dev` filesystem

**proc filesystem**

`sys` filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Problem: some data structures are known by the kernel but may be useful for system administration at user-level

- Solution: **proc virtual filesystem**

- Pseudo-filesystem used to export some kernel data structures to user-level
- Initial purpose: provide information about running processes
- Mounted at `/proc`
- Mostly read-only, but some files allow kernel variables to be changed
- See `man proc` and [Documentation/filesystem/proc.txt](#)

- Examples:

- `/proc/cpuinfo` Information about the CPU installed
- `/proc/meminfo` Information about the amount of memory
- `/proc/version` Information about version and build

# The proc filesystem

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
**proc filesystem**  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency

Char devices  
I/O  
Platforms  
Using sysfs  
Timers

Int. handlers  
Bottom halves  
Examples

- Problem: some data structures are known by the kernel but may be useful for system administration at user-level

## ● Solution: **proc virtual filesystem**

- Pseudo-filesystem used to export some kernel data structures to user-level
- Initial purpose: provide information about running processes
- Mounted at `/proc`
- Mostly read-only, but some files allow kernel variables to be changed
- See `man proc` and `Documentation/filesystem/proc.txt`

## ● Examples:

- `/proc/cpuinfo` Information about the CPU installed
- `/proc/meminfo` Information about the amount of memory
- `/proc/version` Information about version and build

# The proc filesystem

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling

Linux filesystem  
`/dev` filesystem

**proc filesystem**

`sys` filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Problem: some data structures are known by the kernel but may be useful for system administration at user-level

- Solution: **proc virtual filesystem**

- Pseudo-filesystem used to export some kernel data structures to user-level
- Initial purpose: provide information about running processes
- Mounted at `/proc`
- Mostly read-only, but some files allow kernel variables to be changed
- See `man proc` and `Documentation/filesystem/proc.txt`

- Examples:

- `/proc/cpuinfo` Information about the CPU installed
- `/proc/meminfo` Information about the amount of memory
- `/proc/version` Information about version and build

# LAB #2: the `ps` program

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

**proc filesystem**

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

Write a bash script that acts like the `ps` program

# LAB #2: Solution

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
#!/bin/bash

PROC_DIRS='ls -d /proc/[1-9]* | sed 's/\//proc\///' | sort -g | xargs -r'

echo -e "PID \t STATE \t PPID \t NICE \t COMMAND"

for var in `echo $PROC_DIRS`; do
    if [[ -e "/proc/$var" ]]; then
        cmdline='cat /proc/$var/cmdline'
        if [[ "$cmdline" == "" ]]; then
            echo "cat /proc/$var/stat | awk '{print \$1 \"\t\" \$3 \"\t\" \$4 \"\t\" \
\$19 \"\t\" \$2 }'()"
        else
            echo -e "cat /proc/$var/stat | awk '{print \$1 \"\t\" \$3 \"\t\" \
\$4 \"\t\" \$19}' \t $cmdline"
        fi
    done
done
```

# LAB #2: Solution

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
#!/bin/bash

PROC_DIRS=`ls -d /proc/[1-9]* | sed 's/\//proc\///' | sort -g | xargs -r`  
echo -e "PID \t STATE \t PPID \t NICE \t COMMAND"  
  
for var in `echo $PROC_DIRS`; do  
    if [[ -e "/proc/$var" ]]; then  
        cmdline=`cat /proc/$var/cmdline`  
        if [[ "$cmdline" == "" ]]; then  
            echo "` cat /proc/$var/stat | awk '{print $1 "\t" $3 "\t" \\\$4 "\t" $19 "\t" $2 }'"  
        else  
            echo -e "` cat /proc/$var/stat | awk '{print $1 "\t" $3 "\t" \\\$4 "\t" $19}'` \t $cmdline"  
        fi  
    done
```

# The sys filesystem

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem  
/dev filesystem

proc filesystem

**sys filesystem**

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Virtual ram-based filesystem provided by the 2.6 Linux kernel
- It exports information about devices and drivers from the kernel device model to userspace, and is also used for configuration
  - **Goal:** export kernel data structures, their attributes, and the linkages between them to userspace
- Originally based on ramfs
- Previously called `ddfs` ("Device Driver Filesystem") or `driverfs`
- See Documentation/filesystem/sysfs.txt

# The sys filesystem

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem  
/dev filesystem

proc filesystem

**sys filesystem**

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Virtual ram-based filesystem provided by the 2.6 Linux kernel
- It exports information about devices and drivers from the kernel device model to userspace, and is also used for configuration
  - **Goal:** export kernel data structures, their attributes, and the linkages between them to userspace
- Originally based on ramfs
- Previously called `ddfs` ("Device Driver Filesystem") or `driverfs`
- See Documentation/filesystem/sysfs.txt

# The sys filesystem

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

**sys filesystem**

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Virtual ram-based filesystem provided by the 2.6 Linux kernel
- It exports information about devices and drivers from the kernel device model to userspace, and is also used for configuration
  - **Goal:** export kernel data structures, their attributes, and the linkages between them to userspace
- Originally based on ramfs
- Previously called **ddfs** (“*Device Driver Filesystem*”) or **driverfs**
- See Documentation/filesystem/sysfs.txt

# The sys filesystem

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

**sys filesystem**

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- Virtual ram-based filesystem provided by the 2.6 Linux kernel
- It exports information about devices and drivers from the kernel device model to userspace, and is also used for configuration
  - **Goal:** export kernel data structures, their attributes, and the linkages between them to userspace
- Originally based on ramfs
- Previously called `ddfs` (“*Device Driver Filesystem*”) or `driverfs`
- See [Documentation/filesystem/sysfs.txt](#)

# The sys filesystem (2)

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
`/dev` filesystem

`proc` filesystem

**sys filesystem**

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Compiled if `CONFIG_SYSFS=y`
- It is commonly mounted at `/sys`
- Access it by doing: `mount -t sysfs sysfs /sys`
- For each driver or device added in the driver model tree, a directory in sysfs is created:
  - `/sys/devices` reflects the parent/child relationship with subdirectories (e.g., `/sys/devices/cpu`)
    - i.e. the physical layout
  - `/sys/bus/` reflects how the devices belong to different busses (e.g., `/sys/bus/pci/`)
    - Populated with symbolic links
  - `/sys/class/` shows devices grouped according to classes (e.g., `/sys/class/sound/`)
    - Examples: network
  - `/sys/block/` contains the block devices

# The sys filesystem (2)

- Compiled if `CONFIG_SYSFS=y`
- It is commonly mounted at `/sys`
- Access it by doing: `mount -t sysfs sysfs /sys`
- For each driver or device added in the driver model tree, a directory in sysfs is created:
  - `/sys/devices` reflects the parent/child relationship with subdirectories (e.g., `/sys/devices/cpu`)
    - i.e. the physical layout
  - `/sys/bus/` reflects how the devices belong to different busses (e.g., `/sys/bus/pci/`)
    - Populated with symbolic links
  - `/sys/class/` shows devices grouped according to classes (e.g., `/sys/class/sound/`)
    - Examples: network
  - `/sys/block/` contains the block devices

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
**sys filesystem**  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# The sys filesystem (3)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem

## `sys` filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- For device drivers and devices, attributes may be created
- These are simple files that should contain only a single value and/or allow a single value to be set
  - Unlike files in procfs, which need to be heavily parsed
- These files show up in the subdirectory of the device driver respective to the device
- Attributes should be ASCII text files
  - Preferably only one value per file
  - Array of values of the same type are acceptable

# The `tmpfs` filesystem

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Temporary file storage in RAM
- Data lost on power-down
- Often used for `/tmp` directory

# Accessing peripherals

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem

sys filesystem

Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- In the end, how do we access peripherals ?
- In principle, each driver can provide its own mechanism:
  - Entry in `/dev/`
  - Entry in `/proc/`
  - Entry in `/sys/`
- The mechanism is not chosen by the bus driver (e.g., SPI, I2C) but by the device driver

# Accessing peripherals (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- In practice, mechanisms for some classes of devices have been standardized:
  - See directory `Documentation/`
  - Watchdog: `/dev/watchdog`
  - ADC: usually `/dev/*` with specific ioctl
  - GPIOs: entry in `/sys/class/gpio/`
  - Leds: entry in `/sys/class/leds/`
  - Input devices: entry in `/dev/input/` (`evdev`)  
See `include/linux/input.h`

# Accessing peripherals (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- For touchscreens 2 options available:

- TSlib:

- Library specific for touchscreens
- <https://github.com/kergoth/tslib>
- Environment variables:
  - `TSLIB_CONFFILE=/etc/ts.conf` Config file
  - `TSLIB_TSDEVICE=/dev/event0` Device in `dev/`
  - `TSLIB_CALIBFILE=/etc/pointercal` Calibration file
- Calibration through `ts_calibrate`

- Evdev:

- Generic input event interface
- Character devices in `dev/`
- In kernel config: `Device Drivers → Input Device support → Event interface`

# Accessing peripherals (3)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
**sys filesystem**  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- For touchscreens 2 options available:

- TSlib:

- Library specific for touchscreens
- <https://github.com/kergoth/tslib>
- Environment variables:
  - `TSLIB_CONFFILE=/etc/ts.conf` Config file
  - `TSLIB_TSDEVICE=/dev/event0` Device in `dev/`
  - `TSLIB_CALIBFILE=/etc/pointercal` Calibration file
- Calibration through `ts_calibrate`

- Evdev:

- Generic input event interface
- Character devices in `dev/`
- In kernel config: `Device Drivers → Input Device support → Event interface`

# Accessing peripherals (4)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

For very simple peripherals (e.g., leds, gpios) we can give applications the permission to directly write into registers:

```
int memfd = open ("/dev/mem", O_RDWR | O_SYNC);  
  
void* ptr = mmap (0, MAP_SIZE, PROT_READ|PROT_WRITE,  
                  MAP_SHARED, memfd, BASE_ADDRESS);
```

# System calls

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- System calls provide a layer between hardware and user-space processes
- Purposes
  1. Abstract hardware interface
    - During read/write operations we don't want to deal with media or filesystem type
  2. Security and robustness
    - Processes do not access devices directly
  3. Virtualization
    - The kernel manages access to shared resources

# System calls (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Also device files (i.e. `/dev/`) and `/proc/` are ultimately accessed via system calls
- Interestingly, Linux implements far fewer syscalls than most systems
  - About 250 on x86
  - Some systems (not Linux) have more than 1000
  - Incredibly clean system call layer
  - Very few regrets or deprecations
  - Slow rate of addition of new system calls
  - Linux is a relatively stable and feature-complete operating system

# System calls on Unix

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The system call interface in Linux, as with most Unix systems, is provided in part by the C library
  - The C library implements the main API, including the system call interface
- The kernel is concerned only with system calls
  - It does not deal with which library call or application make use if system calls
  - Common motto related to Unix interfaces: “*provide mechanism, not policy*”
  - Unix system calls provide specific functions in a very abstract view, regardless of the actual system call usage

# System calls on Linux

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Often called syscalls in Linux
- Can define one or more arguments (inputs)
- Can result in one or more side effects (e.g. writing data to a file)
- Return a **long** value to specify success/error
  - Long for compatibility with 64-bit architectures
  - Zero usually (but not always) means success
  - The meaning of the returned value can be understood through library functions such as **perror()**
  - System call **bar()** from user level is implemented in the kernel as **sys\_bar()**

# System calls on Linux (2)

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Each system call is assigned a *syscall number*
- Unique number used to reference a specific system call
- The kernel keeps a list of all registered system calls in the system call table, stored in `sys_call_table`
  - Architecture-dependend
  - For i386 see  
`linux-2.6.23/arch/x86/kernel/syscall_table.S`
- Linux provides a “not implemented” system call:  
`sys_ni_syscall()`
  - It just returns `-ENOSYS` (i.e. invalid system call)
  - Used to “plug the hole” in the rare case that a system call is no more available

# System call handler

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists

Scheduling  
Linux filesystem

/dev filesystem

proc filesystem  
sys filesystem

**Syscalls**

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- User-space applications cannot execute kernel code directly
- They cannot call kernel functions because the kernel exists in a protected memory space
- If applications could directly read and write kernel's address space, we wouldn't have any system security and robustness
- User-space applications must somehow "signal" the kernel that they want to execute a system call
  - Thus, the system switches to kernel mode
  - The system call is then executed in kernel-space by the kernel on behalf of the application

# System calls: user space vs kernel space

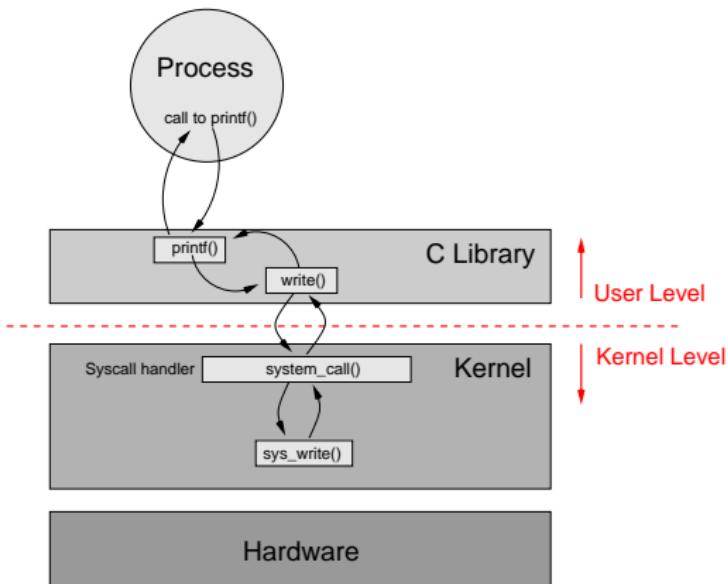
## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls**

- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples



# System call invocation

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem

## Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers

Int. handlers  
Bottom halves  
Examples

- The mechanism to signal the kernel is a software synchronous interrupt (called exception)
- The user-level code triggers an exception to enter the kernel
- The system switches to kernel mode and execute the exception handler
- The defined software interrupt on x86 is the `int $0x80` instruction
  - Modern x86 processors use a feature called `sysenter` to provide a faster way to enter the kernel than using the `int` instruction
- It triggers a switch to kernel mode and execution of exception vector 128, which is the system call handler

# System call invocation (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The system call handler is the `system_call()` function
  - Architecture dependent
  - Typically implemented in assembly in `entry.S`
- The number of the system call requested is passed into the kernel through some register
  - On x86 is the `%eax` register
  - The `system_call()` function checks the value by comparing it to `nr_syscalls`
  - If it is greater or equal to `nr_syscalls`, the function returns `-ENOSYS`
  - Otherwise, the specified system call is invoked:  
`call *sys_call_table(,%eax,4)`  
(each element in the system call table is 32 bits wide)

# System call invocation (3)

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
`/dev` filesystem

`proc` filesystem  
`sys` filesystem

## Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The return value is sent to user space via a register
  - On x86 it is the `%ebx` register
- Also parameters are stored in registers too
  - On x86 the first 5 parameters are stored in `%ebx`, `%ecx`,  
`%edx`, `%esi`, `%edi` register
  - In case of more than 5 parameters, a single register holds a pointer to user-space where all parameters are stored
- On Linux we can find two separate functions:
  - `sys_<syscall>`: it deals with arguments
  - `do_<syscall>`: it performs the actual operation

# System call design

- Define its purpose
  - Do not create syscalls that do multiple things (like `ioctl`)!
- Define arguments, return value and error codes
  - The system call should have a clean and simple interface with few parameters
- Semantics and behaviour are important and must not change
- Design for future: do not needlessly limit the function
- Be as more general as possible
- Remember that its purpose will remain constant but its uses may change
- Design for portability: do not make assumptions about architecture's word size or endianess
- Design for robustness

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# System call implementation: verifying parameters

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Carefully check all parameters to ensure they are valid and legal
  - It runs in kernel space, therefore it can break kernel security
  - E.g. I/O system calls must check file descriptor validity
  - One of the most important checks is the validity of any pointer provided by user

# System call implementation: verifying parameters (2)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

## Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The kernel provide two functions:

- `copy_to_user(dest, src, size)`
  - `dest`: destination memory in process's address space
  - `src`: source pointer in kernel space
  - `size`: size of data to copy
- `copy_from_user(dest, src, size)`
  - `dest`: destination memory in kernel space
  - `src`: source pointer in process's address space
  - `size`: size of data to copy
- Both functions return the number of bytes not copied (zero on success)
- It is standard *for the syscall* to return `-EFAULT` in case of error
- Both functions may block
  - E.g. if the page containing data is swapped (i.e. not in physical memory)
  - In this case, the page fault handler brings the page from the swap file into physical memory

# System call context

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The kernel is in process context during the execution of a system call
- The `current` pointer points to the current task (i.e. the process that issued the syscall)
- The process context is fully preemptible
- Unlike interrupt context, process context can block
  - E.g. if the system call explicitly blocks, or if `schedule()` is called
  - Pros:
    - Therefore, system calls can make use of the majority of the kernel's functionality
    - This greatly simplifies kernel programming
  - Cons:
    - The current task may be preempted by another task which may execute the same system call
    - System calls must be reentrant

# Registering a new system call

Kernel

Intro  
Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Add a new entry to the end of the system call table:

```
.long sys_foo
```

- It needs to be done for each architecture that supports the syscall
- The position in the table is the system call number
- The system call need not receive the same syscall number under each architecture

2. For each architecture supported, the syscall number needs to be defined in `<asm/unistd.h>`

```
#define __NR_foo <syscall number>
```

3. Implement the syscall

- Usually in `kernel/sys.c` or in the most relevant file inside `kernel/`

4. The syscall needs to be compiled into the kernel image
  - It cannot be compiled as a module

5. After compilation, the system call is ready!

# Example of syscall

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
diff -ur /linux-2.6.15-orig/arch/x86/kernel/syscall_table_32.S
/linux-2.6.15-modified/arch/x86/kernel/syscall_table_32.S
--- /linux-2.6.15-orig/arch/x86/kernel/syscall_table_32.S 2006-03-02 22:18:35
+++ /linux-2.6.15-modified/arch/x86/kernel/syscall_table_32.S 2007-11-05 15:1
@@ -294,3 +294,4 @@
        .long sys_inotify_init
        .long sys_inotify_add_watch
        .long sys_inotify_rm_watch
+       .long sys_mysyscall /* 294 */

diff -ur /linux-2.6.15-orig/arch/x86/include/asm/unistd_32.h
/linux-2.6.15-modified/arch/x86/include/asm/unistd_32.h
--- /linux-2.6.15-orig/arch/x86/include/asm/unistd.h 2006-03-02 22:18:41.0000
+++ /linux-2.6.15-modified/arch/x86/include/asm/unistd.h 2007-11-05 15:16:52.
@@ -299,8 +299,9 @@
#define __NR_inotify_init 291
#define __NR_inotify_add_watch 292
#define __NR_inotify_rm_watch 293
+#define __NR_mysyscall 294

-#define NR_syscalls 294
+#define NR_syscalls 295

/*
 * user-visible error numbers are in the range -1 - -128: see
```

# Example of syscall (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
diff -ur /linux-2.6.15-orig/kernel/sys.c /linux-2.6.15-modified/kernel/sys.c
--- /linux-2.6.15-orig/kernel/sys.c 2007-09-24 19:16:49.000000000 +0200
+++ /linux-2.6.15-modified/kernel/sys.c 2007-11-05 16:25:50.000000000 +0100
@@ -1851,3 +1851,10 @@
        }
        return error;
    }
+
+/* System call added by us */
+asm linkage long sys_mysyscall (void)
+{
+    printk(KERN_ERR "This is my system call :)\n");
+    return 0;
+}
```

# Accessing the system-call from user-space

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Generally, the C library provides support for system call
- However, if we just wrote the system call, it is doubtful that glibc already supports it

# Accessing the system-call from user-space (2)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

Before Linux 2.6.18:

- Linux provides a set of macros for wrapping access to system calls
- These macros set up the register content and use the trap instructions
- These macros are named `_syscalln()`
  - Syntax:

```
#define __NR_<syscall name> <syscall number>
_syscalln(<return type>, <syscall name>,
parameters)
```
  - n: the number of parameters of the system call (between zero and six)
  - Used to know how many parameters push into registers
  - The macro expands into a C function with inline assembly

# Accessing the system-call from user-space (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Example:

```
#include <sys/syscall.h>
#include <errno.h>

#define __NR_mysyscall 294
_syscall0 (long, mysyscall)
```

```
int main (void)
{
    mysyscall();
    return 0;
}
```

# Accessing the system-call from user-space (4)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

After Linux 2.6.18:

- We use the `syscall` system call
- We use the syscall number as argument:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>

int main (void)
{
    /* ... */
    syscall(324);
    /* ... */
}
```

# LAB #3: Implementing a system call

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Implement the function `mysyscall()` inside the `kernel/sys.c` file, which just prints "This is my system call" with priority `KERN_ERR`
2. Register a new system call that calls the function
3. Implement a simple C program that invokes the system call by number
4. Compile and run the program
5. Type `dmesg | tail` to check that the system call has been actually called

# Why not to implement a new system call

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Sysealls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Implementation of new system calls on Linux is discouraged
- Pros:
  - Simple to implement
  - Easy to use
  - Fast mechanism
- Cons:
  - Official syscall number needed
  - The syscall interface cannot change
  - Must be registered/implemented on all supported architectures
  - Not easily used from scripts
- Alternatives:
  - Implement a device node and use `read` and `write` to it
    - Use `ioctl` to manipulate specific settings
  - Implement a file in the `sysfs`

# Why not to implement a new system call

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Sysealls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Implementation of new system calls on Linux is discouraged
- Pros:
  - Simple to implement
  - Easy to use
  - Fast mechanism
- Cons:
  - Official syscall number needed
  - The syscall interface cannot change
  - Must be registered/implemented on all supported architectures
  - Not easily used from scripts
- Alternatives:
  - Implement a device node and use `read` and `write` to it
    - Use `ioctl` to manipulate specific settings
  - Implement a file in the `sysfs`

# Why not to implement a new system call

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syccalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Implementation of new system calls on Linux is discouraged
- Pros:
  - Simple to implement
  - Easy to use
  - Fast mechanism
- Cons:
  - Official syscall number needed
  - The syscall interface cannot change
  - Must be registered/implemented on all supported architectures
  - Not easily used from scripts
- Alternatives:
  - Implement a device node and use `read` and `write` to it
    - Use `ioctl` to manipulate specific settings
  - Implement a file in the `sysfs`

# LAB #4: Address space with fork

## 1. Create the `test-fork-address-space.c` file:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

void child(int *m)
{
    sleep(20);
    printf("20 - Child process\n");
    sleep(20);
    m[2] = 5;
    printf("40 - Child: m[2] = 5 set\n");
    printf("40 - Child process %d %d %d\n", m[0], m[1], m[2]);
    sleep(20);
    free (m);
    printf("60 - Child: Free done\n");
}

void parent(int *m)
{
    sleep(10);
    printf("10 - Parent process\n");
    sleep(20);
    m[1] = 3;
    printf("30 - Parent: m[1] = 3 set\n");
    printf("30 - Parent process %d %d %d\n", m[0], m[1], m[2]);
    sleep(20);
    free (m);
    printf("50 - Parent: Free done\n");
    sleep(20);
}
```

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# LAB #4: Address space with fork (2)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

```
int main (void)
{
    int pid;
    int * m;
    m = (int*) malloc (1024);
    m[0] = 1;
    pid = fork();
    if (pid == 0)
        child(m);
    else
        parent(m);
    return 0;
}
```

2. Compile it: `gcc -O0 -static -Wall`

`test-fork-address-space.c -o test-fork-address-space`

3. Run it: `./test-fork-address-space`:

```
10 - Parent process
20 - Child process
30 - Parent: m[1] = 3 set
30 - Parent process 1 3 0
40 - Child: m[2] = 5 set
40 - Child process 1 0 5    <-- Parent did not overwrite child's space
50 - Parent: Free done
60 - Child: Free done
```

# LAB #5: Address space with vfork

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

## 1. Create the `test-vfork-address-space.c` file:

```
int main (void)
{
    int pid;
    int * m;
    m = (int*) malloc (1024);
    m[0] = 1;
    pid = vfork();
    if (pid == 0)
        child(m);
    else
        parent(m);
    return 0;
}
```

## 2. Compile it: `gcc -O0 -static -Wall`

```
test-vfork-address-space.c -o test-vfork-address-space
```

## 3. Run it: `./test-vfork-address-space`:

```
20 - Child process          <-- Child starts before parent
40 - Child: m[2] = 5 set    <-- Child writes into its address space
40 - Child process 1 0 5
60 - Child: Free done
10 - Parent process         <-- Parent starts after child
Segmentation fault (core dumped) <-- Child shouldn't write in its address space
```

# LAB #5: Address space with pthread\_create

## 1. Create the `test-pthread-address-space.c` file:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void* child(void *mm)
{
    int * m = (int *) mm;
    sleep(20);
    printf("20 - Child process\n");
    sleep(20);
    m[2] = 5;
    printf("40 - Child: m[2] = 5 set\n");
    printf("40 - Child process %d %d %d\n", m[0], m[1], m[2]);
    sleep(20);
    free (m);
    printf("60 - Child: Free done\n");
    exit(1);
    return NULL;
}

void parent(int *m)
{
    sleep(10);
    printf("10 - Parent process\n");
    sleep(20);
    m[1] = 3;
    printf("30 - Parent: m[1] = 3 set\n");
    printf("30 - Parent process %d %d %d\n", m[0], m[1], m[2]);
    sleep(20);
    free (m);
    printf("50 - Parent: Free done\n");
    sleep(20);
}
```

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# LAB #6: Address space with pthread\_create (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
int main (void)
{
    pthread_t tid;
    int * m;
    m = (int*) malloc (1024);
    m[0] = 1;
    pthread_create(&tid, NULL, child, (void*) m);
    parent(m);
    return 0;
}
```

2. Compile it: `gcc -O0 -static -Wall`

```
test-pthread-address-space.c -o
```

```
test-pthread-address-space -lpthread
```

3. Run it: `./test-pthread-address-space:`

```
10 - Parent process
```

```
20 - Child process
```

```
30 - Parent: m[1] = 3 set
```

```
30 - Parent process 1 3 0
```

```
40 - Child: m[2] = 5 set
```

```
40 - Child process 1 3 5    <- Parent did overwrite child's space
```

```
50 - Parent: Free done
```

```
*** glibc detected *** ./test-pthread-malloc: double free or corruption
```

# LAB #7: Tracing fork

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Create the `test-fork.c` file:

```
#include <unistd.h>

int main (void)
{
    fork();
    return 0;
}
```

2. Compile it: `gcc -O0 -static -Wall test-fork.c -o test-fork`

3. Trace it: `strace ./test-fork`:

```
fork()
```

4. Disassemble it: `objdump -D -S test-fork > test-fork.dis`

# LAB #7: Tracing fork (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

5. Open the `test-fork.dis` file

6. Search `fork`:

```
        fork();  
80481f4: e8 63 55 00 00          call    804d75c <__libc_fork>  
                    return 0;  
80481f9: b8 00 00 00 00          mov     $0x0,%eax  
}  
80481fe: c9                      leave  
80481ff: c3                      ret
```

# LAB #7: Tracing fork (3)

Kernel

Intro  
Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 7. Then search `__libc_fork`:

```
0804d75c <__libc_fork>:
 804d75c: 55          push    %ebp
 804d75d: b8 00 00 00 00  mov    $0x0,%eax
 804d762: 89 e5        mov    %esp,%ebp
 804d764: 53          push    %ebx
 804d765: 83 ec 04    sub    $0x4,%esp
 804d768: 85 c0        test   %eax,%eax
 804d76a: 74 12        je     804d77e <__libc_fork+0x22>
 804d76c: c7 04 24 80 e0 0a 08  movl   $0x80ae080 ,(%esp)
 804d773: e8 88 28 fb f7  call   0 <_init-0x80480d4>
 804d778: 83 c4 04    add    $0x4,%esp
 804d77b: 5b          pop    %ebx
 804d77c: 5d          pop    %ebp
 804d77d: c3          ret
 804d77e: b8 02 00 00 00  mov    $0x2,%eax
 804d783: cd 80        int    $0x80
 804d785: 3d 00 f0 ff ff  cmp    $0xfffff000 ,%eax
 804d78a: 89 c3        mov    %eax,%ebx
 804d78c: 77 08        ja    804d796 <__libc_fork+0x3a>
 804d78e: 89 d8        mov    %ebx,%eax
 804d790: 83 c4 04    add    $0x4,%esp
 804d793: 5b          pop    %ebx
 804d794: 5d          pop    %ebp
 804d795: c3          ret
```

## 8. Notice that `fork` is associated with syscall number 2

# LAB #7: Tracing fork: the kernel

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

9. Download a copy of the Linux kernel:

```
wget
```

```
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.tar.bz2
```

10. Open the

```
linux-2.6.23/arch/x86/kernel/syscall_table_32.S file
```

11. Notice that the syscall number 2 is associated to

```
sys_fork:
```

```
.long sys_fork /* 2 */
```

# LAB #7: Tracing fork: the kernel (2)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

12. Open the `linux-2.6.23/arch/x86/kernel/process.c` file
13. Search `sys_fork`:

```
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs, 0, NULL, NULL);
}
```
14. Notice that `do_fork()` is called only with the `SIGCHLD` parameter
15. Open the `linux-2.6.23/kernel/fork.c` file
16. Here `do_fork` is!
17. `do_fork` then calls `copy_process()`

# LAB #7: Tracing fork: the kernel (3)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

```
/*
 * Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
long do_fork(unsigned long clone_flags,
              unsigned long stack_start,
              struct pt_regs *regs,
              unsigned long stack_size,
              int __user *parent_tidptr,
              int __user *child_tidptr)
{
    struct task_struct *p;
    int trace = 0;
    struct pid *pid = alloc_pid();
    long nr;

    if (!pid)
        return -EAGAIN;
    nr = pid->nr;
    if (unlikely(current->ptrace)) {
        trace = fork_traceflag (clone_flags);
        if (trace)
            clone_flags |= CLONE_PTRACE;
    }

    p = copy_process(clone_flags, stack_start, regs, stack_size,
                     parent_tidptr, child_tidptr, pid);
```

# LAB #7: Tracing fork: the kernel (4)

```
/*
 * Do this prior waking up the new thread - the thread
 * pointer might get invalid after that point,
 * if the thread exits quickly.
 */
if (!IS_ERR(p)) {
    struct completion vfork;

    if (clone_flags & CLONE_VFORK) {
        p->vfork_done = &vfork;
        init_completion(&vfork);
    }

    if (((p->ptrace & PT_PTRACED) || \
         (clone_flags & CLONE_STOPPED)) {
        /*
         * We'll start up with an immediate SIGSTOP.
         */
        sigaddset(&p->pending.signal, SIGSTOP);
        set_tsk_thread_flag(p, TIF_SIGPENDING);
    }

    if (!(clone_flags & CLONE_STOPPED))
        wake_up_new_task(p, clone_flags);
    else
        p->state = TASK_STOPPED;

    if (unlikely (trace)) {
        current->ptrace_message = nr;
        ptrace_notify ((trace << 8) | SIGTRAP);
    }
}
```

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# LAB #7: Tracing fork: the kernel (5)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
        if (clone_flags & CLONE_VFORK) {
                freezer_do_not_count();
                wait_for_completion(&vfork);
                freezer_count();
                if (unlikely (current->ptrace & \
                             PT_TRACE_VFORK_DONE)) {
                        current->ptrace_message = nr;
                        ptrace_notify \
                            ((PTRACE_EVENT_VFORK_DONE << 8) | \
                             SIGTRAP);
                }
        } else {
                free_pid(pid);
                nr = PTR_ERR(p);
        }
        return nr;
    }
}
```

# Operations done by do\_fork

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**

Real-Time  
Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

The bulk of the work in forking is handled by `do_fork()`, defined in `kernel/fork.c`

Operations performed by `do_fork()`:

1. It allocates a new PID for the child by calling `alloc_pid()`
2. It checks the `ptrace` field of the parent (i.e. `current->ptrace`)
  - If it is not zero, the parent process is being traced by another process
3. It calls `copy_process()`, which sets up the process descriptor and any other kernel data structure required for child's execution
  - 3.1 Its parameters are the same as `do_fork()` plus the PID of the child

# Operations done by do\_fork

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

The bulk of the work in forking is handled by `do_fork()`, defined in `kernel/fork.c`

Operations performed by `do_fork()`:

1. It allocates a new PID for the child by calling `alloc_pid()`
2. It checks the `ptrace` field of the parent (i.e. `current->ptrace`)
  - If it is not zero, the parent process is being traced by another process
3. It calls `copy_process()`, which sets up the process descriptor and any other kernel data structure required for child's execution
  - 3.1 Its parameters are the same as `do_fork()` plus the PID of the child

# Operations done by do\_fork

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

The bulk of the work in forking is handled by `do_fork()`, defined in `kernel/fork.c`

Operations performed by `do_fork()`:

1. It allocates a new PID for the child by calling `alloc_pid()`
2. It checks the `ptrace` field of the parent (i.e. `current->ptrace`)
  - If it is not zero, the parent process is being traced by another process
3. It calls `copy_process()`, which sets up the process descriptor and any other kernel data structure required for child's execution
  - 3.1 Its parameters are the same as `do_fork()` plus the PID of the child

# Operations done by do\_fork (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

`copy_process()` ...

- 3.2 It checks if the flags passed in the `clone_flags` parameter are compatible
- 3.3 It performs additional security checks by invoking `security_task_create()` and `security_task_alloc()`
- 3.4 It calls `dup_task_struct()` which creates new kernel stack, `thread_info` and `task_struct` structures for the new process.
  - The new values are identical to those of current task
  - At this point child and parent process descriptors are identical

# Operations done by do\_fork (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

`copy_process()` ...

- 3.2 It checks if the flags passed in the `clone_flags` parameter are compatible
- 3.3 It performs additional security checks by invoking `security_task_create()` and `security_task_alloc()`
- 3.4 It calls `dup_task_struct()` which creates new kernel stack, `thread_info` and `task_struct` structures for the new process.
  - The new values are identical to those of current task
  - At this point child and parent process descriptors are identical

# Operations done by do\_fork (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

`copy_process()` ...

- 3.2 It checks if the flags passed in the `clone_flags` parameter are compatible
- 3.3 It performs additional security checks by invoking `security_task_create()` and `security_task_alloc()`
- 3.4 It calls `dup_task_struct()` which creates new kernel stack, `thread_info` and `task_struct` structures for the new process.
  - The new values are identical to those of current task
  - At this point child and parent process descriptors are identical

# Operations done by do\_fork (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

`dup_task_struct()` ...

- 3.4.1 It executes the `alloc_task_struct()` macro to get a `task_struct` structure for the new process, and stores its address in the `tsk` local variable.
- 3.4.2 It executes the `alloc_thread_info` macro to get a free memory area to store the `thread_info` structure and the Kernel Mode stack of the new process, and saves its address in the `ti` local variable
- 3.4.3 It copies the contents of the current's process descriptor into the `task_struct` structure pointed to by `tsk`, then it sets `tsk->thread_info` to `ti`

# Operations done by do\_fork (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

`dup_task_struct()` ...

- 3.4.1 It executes the `alloc_task_struct()` macro to get a `task_struct` structure for the new process, and stores its address in the `tsk` local variable.
- 3.4.2 It executes the `alloc_thread_info` macro to get a free memory area to store the `thread_info` structure and the Kernel Mode stack of the new process, and saves its address in the `ti` local variable
- 3.4.3 It copies the contents of the current's process descriptor into the `task_struct` structure pointed to by `tsk`, then it sets `tsk->thread_info` to `ti`

# Operations done by do\_fork (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

**dup\_task\_struct()** ...

- 3.4.1 It executes the `alloc_task_struct()` macro to get a `task_struct` structure for the new process, and stores its address in the `tsk` local variable.
- 3.4.2 It executes the `alloc_thread_info` macro to get a free memory area to store the `thread_info` structure and the Kernel Mode stack of the new process, and saves its address in the `ti` local variable
- 3.4.3 It copies the contents of the current's process descriptor into the `task_struct` structure pointed to by `tsk`, then it sets `tsk->thread_info` to `ti`

# Operations done by do\_fork (4)

Kernel

  Intro

  Compiling

  Code

  Debugging

  Linked lists

  Scheduling

  Linux filesystem

  /dev filesystem

  proc filesystem

  sys filesystem

**Syscalls**

  Real-Time

  Boot time

Kernel drivers

  Modules

  Memory

  Concurrency

  Char devices

  I/O

  Platforms

  Using sysfs

  Timers

  Int. handlers

  Bottom halves

  Examples

**dup\_task\_struct()** ...

- 3.4.4 It copies the contents of the current's **thread\_info** descriptor into the structure pointed to by **ti**, then it sets **ti->task** to **tsk**
- 3.4.5 It sets the usage counter of the new process descriptor (i.e. **tsk->usage**) to 2 to specify that the process descriptor is in use and that the corresponding process is alive (its state is not **EXIT\_ZOMBIE** or **EXIT\_DEAD**)
- 3.4.6 It returns the process descriptor pointer of the new process (i.e. **tsk**)

# Operations done by do\_fork (4)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

`dup_task_struct()` ...

- 3.4.4 It copies the contents of the current's `thread_info` descriptor into the structure pointed to by `ti`, then it sets `ti->task` to `tsk`
- 3.4.5 It sets the usage counter of the new process descriptor (i.e. `tsk->usage`) to 2 to specify that the process descriptor is in use and that the corresponding process is alive (its state is not `EXIT_ZOMBIE` or `EXIT_DEAD`)
- 3.4.6 It returns the process descriptor pointer of the new process (i.e. `tsk`)

# Operations done by do\_fork (4)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

**dup\_task\_struct()** ...

- 3.4.4 It copies the contents of the current's `thread_info` descriptor into the structure pointed to by `ti`, then it sets `ti->task` to `tsk`
- 3.4.5 It sets the usage counter of the new process descriptor (i.e. `tsk->usage`) to 2 to specify that the process descriptor is in use and that the corresponding process is alive (its state is not `EXIT_ZOMBIE` or `EXIT_DEAD`)
- 3.4.6 It returns the process descriptor pointer of the new process (i.e. `tsk`)

# Operations done by do\_fork (5)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

## Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

`copy_process()` ...

- 3.5 It checks if the maximum amount of processes for the current user has not been exceeded (i.e. greater than `max_threads`)
- 3.6 It differentiates the child from the parent by clearing or initializing various fields of the `task_struct`
- 3.7 It calls `copy_flags()` to update the `flags` field of the `task_struct`
  - The `PF_SUPERPRIV` (which denotes if a task used superuser privileges) and `PF_NOFREEZE` flags are cleared
  - The `PF_FORKNOEXEC` flag (which denotes if a task has not called `exec()`) is set
- 3.8 It calls `init_sigpending()` that clears the pending signals

# Operations done by do\_fork (5)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

## Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

### copy\_process () ...

- 3.5 It checks if the maximum amount of processes for the current user has not been exceeded (i.e. greater than `max_threads`)
- 3.6 It differentiates the child from the parent by clearing or initializing various fields of the `task_struct`
- 3.7 It calls `copy_flags()` to update the `flags` field of the `task_struct`
  - The `PF_SUPERPRIV` (which denotes if a task used superuser privileges) and `PF_NOFREEZE` flags are cleared
  - The `PF_FORKNOEXEC` flag (which denotes if a task has not called `exec()`) is set
- 3.8 It calls `init_sigpending()` that clears the pending signals

# Operations done by do\_fork (5)

Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem

proc filesystem  
sys filesystem

Syscalls

Real-Time  
Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

`copy_process()` ...

- 3.5 It checks if the maximum amount of processes for the current user has not been exceeded (i.e. greater than `max_threads`)
- 3.6 It differentiates the child from the parent by clearing or initializing various fields of the `task_struct`
- 3.7 It calls `copy_flags()` to update the `flags` field of the `task_struct`
  - The `PF_SUPERPRIV` (which denotes if a task used superuser privileges) and `PF_NOFREEZE` flags are cleared
  - The `PF_FORKNOEXEC` flag (which denotes if a task has not called `exec()`) is set
- 3.8 It calls `init_sigpending()` that clears the pending signals

# Operations done by do\_fork (5)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

## Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

### copy\_process () ...

- 3.5 It checks if the maximum amount of processes for the current user has not been exceeded (i.e. greater than `max_threads`)
- 3.6 It differentiates the child from the parent by clearing or initializing various fields of the `task_struct`
- 3.7 It calls `copy_flags()` to update the `flags` field of the `task_struct`
  - The `PF_SUPERPRIV` (which denotes if a task used superuser privileges) and `PF_NOFREEZE` flags are cleared
  - The `PF_FORKNOEXEC` flag (which denotes if a task has not called `exec()`) is set
- 3.8 It calls `init_sigpending()` that clears the pending signals

# Operations done by do\_fork (6)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

`copy_process()` ...

3.9 Depending on the parameters passed to `do_fork()`,  
`copy_process()` then duplicates or shares resources

- Open files
- Filesystem information
- Signal handlers
- Address space

3.10 It calls `sched_fork()` which splits the remaining timeslice between parent and child

3.11 Finally, it returns a pointer to the new child

# Operations done by do\_fork (6)

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

**copy\_process()** ...

- 3.9 Depending on the parameters passed to **do\_fork()**, **copy\_process()** then duplicates or shares resources

- Open files
- Filesystem information
- Signal handlers
- Address space

- 3.10 It calls **sched\_fork()** which splits the remaining timeslice between parent and child

- 3.11 Finally, it returns a pointer to the new child

# Operations done by do\_fork (6)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

**copy\_process()** ...

3.9 Depending on the parameters passed to **do\_fork()**,  
**copy\_process()** then duplicates or shares resources

- Open files
- Filesystem information
- Signal handlers
- Address space

3.10 It calls **sched\_fork()** which splits the remaining  
timeslice between parent and child

3.11 Finally, it returns a pointer to the new child

# Operations done by do\_fork (7)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

do\_fork() ...

4. If the `CLONE_STOPPED` flag is set or the child process must be traced (i.e. the `PT_PTRACED` flag is set in `p->ptrace`), it adds a pending `SIGSTOP` signal to it

# Operations done by do\_fork (8)

5. If the `CLONE_STOPPED` flag is not set, it invokes the `wake_up_new_task()` function, which performs the following operations:
  - 5.1 It adjusts the scheduling parameters of both the parent and the child
  - 5.2 If the child will run on the same CPU as the parent and parent and child do not share the same set of page tables (i.e. `CLONE_VM` flag cleared), it then forces the child to run before the parent by inserting it into the parent's runqueue right before the parent. This simple step yields better performance if the child flushes its address space and executes a new program right after the forking. If we let the parent run first, the Copy On Write mechanism would give rise to a series of unnecessary page duplications.
  - 5.3 Otherwise, if the child will not be run on the same CPU as the parent, or if parent and child share the same set of page tables (i.e. `CLONE_VM` flag set), it inserts the child in the last position of the parent's runqueue

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Operations done by do\_fork (8)

5. If the `CLONE_STOPPED` flag is not set, it invokes the `wake_up_new_task()` function, which performs the following operations:
  - 5.1 It adjusts the scheduling parameters of both the parent and the child
  - 5.2 If the child will run on the same CPU as the parent and parent and child do not share the same set of page tables (i.e. `CLONE_VM` flag cleared), it then forces the child to run before the parent by inserting it into the parent's runqueue right before the parent. This simple step yields better performance if the child flushes its address space and executes a new program right after the forking. If we let the parent run first, the Copy On Write mechanism would give rise to a series of unnecessary page duplications.
  - 5.3 Otherwise, if the child will not be run on the same CPU as the parent, or if parent and child share the same set of page tables (i.e. `CLONE_VM` flag set), it inserts the child in the last position of the parent's runqueue

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Operations done by do\_fork (8)

5. If the `CLONE_STOPPED` flag is not set, it invokes the `wake_up_new_task()` function, which performs the following operations:
  - 5.1 It adjusts the scheduling parameters of both the parent and the child
  - 5.2 If the child will run on the same CPU as the parent and parent and child do not share the same set of page tables (i.e. `CLONE_VM` flag cleared), it then forces the child to run before the parent by inserting it into the parent's runqueue right before the parent. This simple step yields better performance if the child flushes its address space and executes a new program right after the forking. If we let the parent run first, the Copy On Write mechanism would give rise to a series of unnecessary page duplications.
  - 5.3 Otherwise, if the child will not be run on the same CPU as the parent, or if parent and child share the same set of page tables (i.e. `CLONE_VM` flag set), it inserts the child in the last position of the parent's runqueue

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem  
**Syscalls**  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Operations done by do\_fork (9)

Kernel

Intro  
Compiling

Code  
Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

6. Otherwise, if the `CLONE_STOPPED` flag is set, it puts the child in the `TASK_STOPPED` state
7. If the parent process is being traced, it stores the PID of the child in the `ptrace_message` field of `current` and invokes `ptrace_notify()`, which essentially stops the current process and sends a `SIGCHLD` signal to its parent. The “grandparent” of the child is the debugger that is tracing the parent; the `SIGCHLD` signal notifies the debugger that `current` has forked a child, whose PID can be retrieved by looking into the `current->ptrace_message` field.

# Operations done by do\_fork (9)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

6. Otherwise, if the `CLONE_STOPPED` flag is set, it puts the child in the `TASK_STOPPED` state
7. If the parent process is being traced, it stores the PID of the child in the `ptrace_message` field of `current` and invokes `ptrace_notify()`, which essentially stops the current process and sends a `SIGCHLD` signal to its parent. The “grandparent” of the child is the debugger that is tracing the parent; the `SIGCHLD` signal notifies the debugger that `current` has forked a child, whose PID can be retrieved by looking into the `current->ptrace_message` field.

# Operations done by do\_fork (10)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

8. If the **CLONE\_VFORK** flag is specified, it inserts the parent process in a wait queue and suspends it until the child releases its memory address space (that is, until the child either terminates or executes a new program)
9. It terminates by returning the PID of the child.

# Operations done by do\_fork (10)

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

**Syscalls**

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

8. If the `CLONE_VFORK` flag is specified, it inserts the parent process in a wait queue and suspends it until the child releases its memory address space (that is, until the child either terminates or executes a new program)
9. It terminates by returning the PID of the child.

# vfork

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The `vfork()` system call is similar to `fork()`
- The page table entries of the parent are not replicated
  - **The child executes in parent's address space**
  - The child is not allowed to write to the address space
  - The stack is shared too
  - Signal handlers are inherited, but not shared
- The parent is blocked until the child calls `exec()` or exits
  - In the `mm_release()` function, which is used when a task exits a memory address space, `vfork_done` is checked. If it is not NULL, the parent is signaled
  - Signals to the parent arrive after the child releases the parent

# vfork (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The behaviour is undefined in the following circumstances:
  - If the child process modifies any data other than the variable of type `pid_t` used to store the return value from `vfork()`
  - If the child process returns from the function in which `vfork()` was called
  - If the child process calls any other function than `_exit()` or one of the `exec()` functions
- This system call is deprecated!
  - Introduced by BSD 3.0 for efficiency when `fork` didn't have the Copy-On-Write mechanism
  - The behaviour is system-dependent and unreliable

# LAB #8: Tracing vfork

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. Create the `test-vfork.c` file:

```
#include <unistd.h>

int main (void)
{
    vfork ();
    _exit (-1);
    return 0;
}
```

## 2. Compile it: `gcc -O0 -static -Wall test-vfork.c -o test-vfork`

## 3. Trace it: `strace ./test-vfork:`

```
vfork ()
```

## 4. Disassemble it: `objdump -D -S test-vfork > test-vfork.dis`

# LAB #8: Tracing vfork (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 5. Open the `test-vfork.dis` file

## 6. Search `vfork`:

```
080481d8 <main>:
 80481d8: 55          push    %ebp
 80481d9: 89 e5        mov     %esp,%ebp
 80481db: 83 ec 08      sub    $0x8,%esp
 80481de: 83 e4 f0      and    $0xfffffffff0,%esp
 80481e1: b8 00 00 00 00  mov    $0x0,%eax
 80481e6: 83 c0 0f      add    $0xf,%eax
 80481e9: 83 c0 0f      add    $0xf,%eax
 80481ec: c1 e8 04      shr    $0x4,%eax
 80481ef: c1 e0 04      shl    $0x4,%eax
 80481f2: 29 c4        sub    %eax,%esp
 80481f4: e8 67 55 00 00  call   804d760 <__vfork>
 80481f9: b8 00 00 00 00  mov    $0x0,%eax
 80481fe: c9          leave
 80481ff: c3          ret
```

# LAB #8: Tracing vfork (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 7. Then search `__vfork`:

```
0804d760 <__vfork>:  
 804d760: b8 00 00 00 00      mov    $0x0,%eax  
 804d765: 85 c0              test   %eax,%eax  
 804d767: 0f 85 cb d1 01 00  jne    806a938 <__libc_fork>  
 804d76d: 59                 pop    %ecx  
 804d76e: b8 be 00 00 00      mov    $0xbe,%eax  
 804d773: cd 80              int    $0x80
```

## 8. Notice that `vfork` is associated with syscall number 0xBE (i.e. 190)

# LAB #8: Tracing vfork: the kernel

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

9. Download a copy of the Linux kernel:

```
wget
```

```
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.tar.bz2
```

10. Open the

```
linux-2.6.23/arch/x86/kernel/syscall_table_32.S file
```

11. Notice that the syscall number 190 is associated to

```
sys_vfork:
```

```
.long sys_vfork /* 190 */
```

# LAB #8: Tracing vfork: the kernel (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

12. Open the `linux-2.6.23/arch/x86/kernel/process.c` file
13. Search `sys_vfork`:

```
asmlinkage int sys_vfork(struct pt_regs *regs)
{
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD,
                   regs.esp, &regs, 0, NULL, NULL);
}
```

14. Notice that this time `do_fork()` is called also with the `CLONE_VFORK` and `CLONE_VM` parameters
15. Open the `include/linux/sched.h` file:

```
/* set if VM shared between processes: */
#define CLONE_VM          0x00000100
```

```
/* set if the parent wants the child
 * to wake it up on mm_release: */
#define CLONE_VFORK        0x00004000
```

# Meaning of CLONE\_VM

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem

## Syscalls

- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

If `CLONE_VM` is set, the calling process and the child processes run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. Moreover, any memory mapping or unmapping performed with `mmap(2)` or `munmap(2)` by the child or calling process also affects the other process.

If `CLONE_VM` is not set, the child process runs in a separate copy of the memory space of the calling process at the time of `clone()`. Memory writes or file mappings/unmappings performed by one of the processes do not affect the other, as with `fork(2)`.

# Meaning of CLONE\_VFORK

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

If **CLONE\_VFORK** is set, the execution of the calling process is suspended until the child releases its virtual memory resources via a call to `execve(2)` or `_exit(2)` (as with `vfork(2)`).

If **CLONE\_VFORK** is not set then both the calling process and the child are schedulable after the call, and an application should not rely on execution occurring in any particular order.

# fork vs vfork in the kernel

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `copy_process()` calls `copy_mm()`
- If `CLONE_VM` is set
  - `copy_mm()` doesn't call `dup_mm()` which is the function encharged of duplicate process's address space
- In `copy_process()` the `task_struct` member `vfork_done` is seto to `NULL`
- If `CLONE_VFORK` is set
  - In `do_fork()`:
    - The `vfork_done` field of child's `task_struct` is pointed at a specific address
    - `wait_for_completion()` blocks the parent process
  - In the `mm_release()` function, which is used when a task exits a memory address space:
    - `vfork_done` is checked to see if it is `NULL`
    - If not so, the parent is signaled and waked up through `complete()`

# fork vs vfork in the kernel

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `copy_process()` calls `copy_mm()`
- If `CLONE_VM` is set
  - `copy_mm()` doesn't call `dup_mm()` which is the function encharged of duplicate process's address space
- In `copy_process()` the `task_struct` member `vfork_done` is set to `NULL`
- If `CLONE_VFORK` is set
  - In `do_fork()`:
    - The `vfork_done` field of child's `task_struct` is pointed at a specific address
    - `wait_for_completion()` blocks the parent process
  - In the `mm_release()` function, which is used when a task exits a memory address space:
    - `vfork_done` is checked to see if it is `NULL`
    - If not so, the parent is signaled and waked up through `complete()`

# LAB #9: Tracing pthread\_create

## 1. Create the `test-pthread-create.c` file:

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

void * start_routine(void* a)
{

int main (void)
{
    pthread_t tid;
    pthread_create(&tid, NULL, start_routine, NULL);

    return 0;
}
```

## 2. Compile it: `gcc -O0 -static -Wall`

```
test-pthread-create.c -o test-pthread-create -lpthread
```

## 3. Trace it: `strace ./test-pthread-create`:

```
clone(child_stack=0x80fe9e4, flags=CLONE_VM|CLONE_FS| \
CLONE_FILES|CLONE_SIGHAND)
```

## 4. Disassemble it: `objdump -D -S test-pthread-create > test-pthread-create.dis`

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Sysealls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# LAB #9: Tracing pthread\_create (2)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

## 5. Open the `test-pthread-create.dis` file

### 6. Search `__clone`:

```
0805a700 <__clone>:  
0805a700: b8 ea ff ff ff      mov    $0xffffffff, %eax  
0805a705: 8b 4c 24 04        mov    0x4(%esp), %ecx  
0805a709: 85 c9              test   %ecx, %ecx  
0805a70b: 0f 84 7f 3d ff ff  je     804e490 <__syscall_error  
0805a711: 8b 4c 24 08        mov    0x8(%esp), %ecx  
0805a715: 85 c9              test   %ecx, %ecx  
0805a717: 0f 84 73 3d ff ff  je     804e490 <__syscall_error  
0805a71d: 83 e1 f0          and    $0xfffffff0, %ecx  
0805a720: 83 e9 1c          sub    $0x1c, %ecx  
0805a723: 8b 44 24 10        mov    0x10(%esp), %eax  
0805a727: 89 41 0c          mov    %eax, 0xc(%ecx)  
0805a72a: 8b 44 24 04        mov    0x4(%esp), %eax  
0805a72e: 89 41 08          mov    %eax, 0x8(%ecx)  
0805a731: c7 41 04 00 00 00 00  movl   $0x0, 0x4(%ecx)  
0805a738: c7 01 00 00 00 00 00  movl   $0x0, (%ecx)  
0805a73e: 53                push   %ebx  
0805a73f: 56                push   %esi  
0805a740: 57                push   %edi  
0805a741: 8b 74 24 24        mov    0x24(%esp), %esi  
0805a745: 8b 54 24 20        mov    0x20(%esp), %edx  
0805a749: 8b 5c 24 18        mov    0x18(%esp), %ebx  
0805a74d: 8b 7c 24 28        mov    0x28(%esp), %edi  
0805a751: b8 78 00 00 00      mov    $0x78, %eax  
0805a756: cd 80              int    $0x80
```

## 8. Notice that `clone` is associated with syscall number 0x78 (i.e. 120)

# LAB #9: Tracing pthread\_create: the kernel

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

9. Download a copy of the Linux kernel:

```
wget
```

```
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.tar.bz2
```

10. Open the

```
linux-2.6.23/arch/x86/kernel/syscall_table_32.S file
```

11. Notice that the syscall number 120 is associated to

```
sys_clone:
```

```
.long sys_clone /* 120 */
```

# LAB #9: Tracing pthread\_create: the kernel (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

12. Open the `linux-2.6.23/arch/x86/kernel/process.c` file

13. Search `sys_clone`:

```
asmlinkage int sys_clone(struct pt_regs *regs)
{
    /* ... */
    return do_fork(clone_flags, newsp, &regs, \
                   0, parent_tidptr, child_tidptr);
}
```

14. Therefore this time `do_fork()` is called with the `CLONE_VM`, `CLONE_FS`, `CLONE_FILES`, `CLONE_SIGHAND` parameters

15. Open the `include/linux/sched.h` file:

```
/* set if VM shared between processes: */
#define CLONE_VM          0x00000100

/* set if fs info shared between processes: */
#define CLONE_FS          0x00000200

/* set if open files shared between processes: */
#define CLONE_FILES        0x00000400

/* set if signal handlers and blocked signals shared: */
#define CLONE_SIGHAND      0x00000800
```

# Meaning of CLONE\_FS

## Kernel

- [Intro](#)
- [Compiling](#)
- [Code](#)
- [Debugging](#)
- [Linked lists](#)
- [Scheduling](#)
- [Linux filesystem](#)
- [/dev filesystem](#)
- [proc filesystem](#)
- [sys filesystem](#)

## Syscalls

- [Real-Time](#)
- [Boot time](#)

## Kernel drivers

- [Modules](#)
- [Memory](#)
- [Concurrency](#)
- [Char devices](#)
- [I/O](#)
- [Platforms](#)
- [Using sysfs](#)
- [Timers](#)
- [Int. handlers](#)
- [Bottom halves](#)
- [Examples](#)

If `CLONE_FS` is set, the caller and the child processes share the same file system information. This includes the root of the file system, the current working directory, and the umask. Any call to `chroot(2)`, `chdir(2)`, or `umask(2)` performed by the calling process or the child process also affects the other process.

If `CLONE_FS` is not set, the child process works on a copy of the file system information of the calling process at the time of the `clone()` call. Calls to `chroot(2)`, `chdir(2)`, `umask(2)` performed later by one of the processes do not affect the other process.

# Meaning of CLONE\_FILES

Kernel

Intro  
Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

If `CLONE_FILES` is set, the calling process and the child processes share the same file descriptor table. Any file descriptor created by the calling process or by the child process is also valid in the other process. Similarly, if one of the processes closes a file descriptor, or changes its associated flags (using the `fcntl(2)` `F_SETFD` operation), the other process is also affected.

If `CLONE_FILES` is not set, the child process inherits a copy of all file descriptors opened in the calling process at the time of `clone()`. (The duplicated file descriptors in the child refer to the same open file descriptions (see `open(2)`) as the corresponding file descriptors in the calling process.) Subsequent operations that open or close file descriptors, or change file descriptor flags, performed by either the calling process or the child process do not affect the other process.

# Meaning of CLONE\_SIGHAND

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

If **CLONE\_SIGHAND** is set, the calling process and the child processes share the same table of signal handlers. If the calling process or child process calls `sigaction(2)` to change the behavior associated with a signal, the behavior is changed in the other process as well. However, the calling process and child processes still have distinct signal masks and sets of pending signals. So, one of them may block or unblock some signals using `sigprocmask(2)` without affecting the other process.

If **CLONE\_SIGHAND** is not set, the child process inherits a copy of the signal handlers of the calling process at the time `clone()` is called. Calls to `sigaction(2)` performed later by one of the processes have no effect on the other process.

# fork vs pthread\_create in the kernel

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- `copy_process()` calls `copy_mm()`, `copy_fs()`,  
`copy_files()`, `copy_sighand()`
- If `CLONE_VM` is set
  - `copy_mm()` doesn't call `dup_mm()` which is the function  
encharged of duplicating process's address space
  - i.e. `mm` field of `task_struct`
- If `CLONE_FS` is set
  - `copy_fs()` doesn't call `_copy_fs_struct()`
  - i.e. `fs` field of `task_struct`
- If `CLONE_FILES` is set
  - `copy_files()` doesn't call `dup_fd()`, which is the  
function encharged of allocating a new files structure
  - i.e. `files` field of `task_struct`
- If `CLONE_SIGHAND` is set
  - `copy_sighand()` doesn't call `memcpy()`
  - i.e. `sighand` field of `task_struct`

# fork vs pthread\_create in the kernel

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `copy_process()` calls `copy_mm()`, `copy_fs()`,  
`copy_files()`, `copy_sighand()`
- If `CLONE_VM` is set
  - `copy_mm()` doesn't call `dup_mm()` which is the function  
encharged of duplicating process's address space
  - i.e. `mm` field of `task_struct`
- If `CLONE_FS` is set
  - `copy_fs()` doesn't call `_copy_fs_struct()`
  - i.e. `fs` field of `task_struct`
- If `CLONE_FILES` is set
  - `copy_files()` doesn't call `dup_fd()`, which is the  
function encharged of allocating a new files structure
  - i.e. `files` field of `task_struct`
- If `CLONE_SIGHAND` is set
  - `copy_sighand()` doesn't call `memcpy()`
  - i.e. `sighand` field of `task_struct`

# fork vs pthread\_create in the kernel

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- `copy_process()` calls `copy_mm()`, `copy_fs()`,  
`copy_files()`, `copy_sighand()`
- If `CLONE_VM` is set
  - `copy_mm()` doesn't call `dup_mm()` which is the function  
encharged of duplicating process's address space
  - i.e. `mm` field of `task_struct`
- If `CLONE_FS` is set
  - `copy_fs()` doesn't call `_copy_fs_struct()`
  - i.e. `fs` field of `task_struct`
- If `CLONE_FILES` is set
  - `copy_files()` doesn't call `dup_fd()`, which is the  
function encharged of allocating a new files structure
  - i.e. `files` field of `task_struct`
- If `CLONE_SIGHAND` is set
  - `copy_sighand()` doesn't call `memcpy()`
  - i.e. `sighand` field of `task_struct`

# fork vs pthread\_create in the kernel

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
**Syscalls**  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- `copy_process()` calls `copy_mm()`, `copy_fs()`,  
`copy_files()`, `copy_sighand()`
- If `CLONE_VM` is set
  - `copy_mm()` doesn't call `dup_mm()` which is the function  
encharged of duplicating process's address space
  - i.e. `mm` field of `task_struct`
- If `CLONE_FS` is set
  - `copy_fs()` doesn't call `_copy_fs_struct()`
  - i.e. `fs` field of `task_struct`
- If `CLONE_FILES` is set
  - `copy_files()` doesn't call `dup_fd()`, which is the  
function encharged of allocating a new files structure
  - i.e. `files` field of `task_struct`
- If `CLONE_SIGHAND` is set
  - `copy_sighand()` doesn't call `memcpy()`
  - i.e. `sighand` field of `task_struct`

# Summing up...

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

|                  | Memory space<br>(data, heap,<br>stack) | Text<br>Segment |
|------------------|----------------------------------------|-----------------|
| fork()           | Duplicated<br>(Copy-On-Write)          | Shared          |
| vfork()          | Shared                                 | Shared          |
| pthread_create() | Shared                                 | Shared          |
| execve           | Overwritten                            | Overwritten     |

# Summing up... (2)

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

|                  | File descriptor table | Filesystem Information |
|------------------|-----------------------|------------------------|
| fork()           | Inherited             | Inherited              |
| vfork()          | Inherited             | Inherited              |
| pthread_create() | Shared                | Shared                 |
| execve           | Inherited             | Unchanged              |

# Summing up... (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

**Syscalls**

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

|                  | Pending signals | Signal Handlers table |
|------------------|-----------------|-----------------------|
| fork()           | Resetted        | Inherited             |
| vfork()          | Resetted        | Inherited             |
| pthread_create() | Resetted        | Shared                |
| execve           | Cleared         | Re-initialized (*)    |

# Real-Time Systems

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Real-time system: computing system in which computational activities must be performed within predefined timing constraints
- Real-time operating system (RTOS): operating system capable of guaranteeing the timing requirements of the tasks under its control through some hypothesis about their behaviour and a model of the external environment
- Real-time does not mean fast computing!
  - Main goal: predictability
  - Fastness remains a desired feature

# Real-Time: approaches

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

**Real-Time**

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Interrupt Abstraction, to create a hard RTOS using Linux
2. Real-time scheduling algorithms
3. Reducing system latency

# Real-Time: Interrupt abstraction

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

**Real-Time**

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

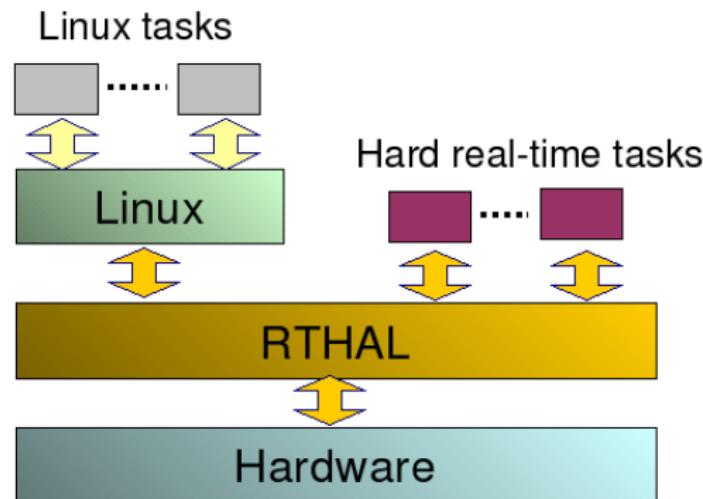
Bottom halves

Examples

- Real-Time Hardware Abstraction Layer (RTHAL) that takes full control of interrupts and system timers
- Provides Hard Real-Time
- Examples:
  - RTLinux
  - RTAI
  - Xenomai

# Real-Time: Interrupt abstraction (2)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
**Real-Time**  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples



# Real-Time: Interrupt abstraction (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- RTLinux:

- First real-time extension for Linux, created by Victor Yodaiken
- The author patented the addition of real-time support to general operating systems as implemented in RTLinux
- The project was forked into a GPL project and a commercial project
- This patent drew many developers away and frightened users. Community projects like RTAI and Xenomai now attract most developers and users.
- February, 2007: commercial version sold to Wind River (then acquired by Intel). Now supported by Wind River as “Real-Time Core for Wind River Linux.”
- Free version still available (see <http://www.rtlinuxfree.com>) but no longer a community project

# Real-Time: Interrupt abstraction (4)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- RTAI:

- Created in 1999, by Prof. Paolo Montegazza (long time contributor to RTLinux), Dipartimento di Ingegneria Aerospaziale Politecnico di Milano (DIAPM)
- Community project. Significant user base.
- Attracted contributors frustrated by the RTLinux legal issues.
- However, only actively maintained on x86.
- Url: <http://www.rtai.org>

# Real-Time: Interrupt abstraction (5)

## Kernel

Intro  
Compiling

Code  
Debugging

Linked lists

Scheduling  
Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## • Xenomai:

- Started in 2001 as a project aiming at emulating traditional RTOS.
- Initial goals: facilitate the porting of programs to GNU / Linux.
- Later included in the RTAI project
- Skins implementing various APIs of traditional RTOS such as VxWorks, as well as the POSIX API, and a “native” API.
- Support for hard real-time in user-space
- Support independent from the Linux kernel version as much as possible
- Now an independent project and an alternative to RTAI.
- Aims at working both as a co-kernel and on top of **PREEMPT\_RT** in the upcoming 3.0 branch.

# Real-Time: Interrupt abstraction (6)

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time**
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

## • Xenomai...

- Threads can be executed in two modes:
  - Primary mode: the thread is handled by Xenomai scheduler
  - Secondary mode: thread handled by the Linux scheduler
- Migrations:
  - A thread migrates from secondary mode to primary mode when a Xenomai system call is issued
  - A thread migrates from primary mode to secondary mode when a Linux system call is issued, or to handle gracefully exceptional events such as exceptions or Linux signals
- Supported platforms:
  - Arm (several platforms supported)
  - Blackfin (BF533 and BF537 boards)
  - Nios2
  - PPC
  - x86

# Real-Time: Interrupt abstraction (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

System calls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Limitations:

- No memory protection (on RTLinux/GPL)
- Use of Linux device drivers not always possible
- Linux executed as a background task (i.e., risk of deadlock)

# Real-Time: scheduling algorithms

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Several algorithms proposed
- Example: SCHED\_DEADLINE
  - Implemented by Evidence in the context of the ACTORS European project with Ericsson research
  - The project adds a further scheduling algorithm based on Earliest Deadline First (EDF)
  - Platform-independent
  - Distributed as git tree
  - Url: [http://www.gitorious.org/sched\\_deadline/pages/Home](http://www.gitorious.org/sched_deadline/pages/Home)

# Real-Time: Kernel preemption

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
**Real-Time**  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

Making the system more predictable: PREEMPT\_RT project

- Main contributors: Ingo Molnar, Steven Rostedt, Thomas Gleixner
- Url: <http://rt.wiki.kernel.org>
- Available for the most recent Linux kernel releases
- Supports all the major embedded architectures
- Makes all but the most critical kernel code involuntarily preemptible
- Preemptible spinlocks:
  - Replaces most spinlocks with preemptible priority-inheritance mutexes
  - Benefit: reduce maximum latency

# Real-Time: Kernel preemption (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- PREEMPT\_RT project ...

- Threaded interrupts:

- Converting interrupt handlers into preemptible kernel threads
    - Reduces latency by running soft IRQs and selected or all hard interrupts in a dedicated kernel thread (managed in process context by the regular Linux scheduler).
    - Thanks to this, real-time tasks can have priority over some or all IRQs. Previously, even soft IRQs were run before any task managed by the scheduler.

- Priority Inheritance

- Merged inside vanilla kernel since Linux 2.6.18

# Myths about PREEMPT\_RT patches

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- It will improve throughput and overall performance
  - Wrong: it will degrade overall performance.
- It will reduce average latency
  - Often wrong. The maximum latency will be reduced.
- The primary goal is to make the system predictable and deterministic!

# How to inspect boot time

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

**Boot time**

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. Enable kernel symbol `CONFIG_PRINTK_TIME`

- It adds timing information to every kernel message (`dmesg`)

## 2. Enable kernel symbol `CONFIG_INITCALL_DEBUG`

- It enables a printk message every time an initcall is run
- It can be enabled also using the kernel option `initcall_debug` on U-Boot's bootargs

## 3. Print on dmesg any additional user-level output:

- `echo hello > /dev/kmsg`

# How to inspect boot time

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

**Boot time**

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. Enable kernel symbol `CONFIG_PRINTK_TIME`

- It adds timing information to every kernel message (`dmesg`)

## 2. Enable kernel symbol `CONFIG_INITCALL_DEBUG`

- It enables a printk message every time an initcall is run
- It can be enabled also using the kernel option `initcall_debug` on U-Boot's bootargs

## 3. Print on dmesg any additional user-level output:

- `echo hello > /dev/kmsg`

# How to inspect boot time

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

**Boot time**

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. Enable kernel symbol `CONFIG_PRINTK_TIME`

- It adds timing information to every kernel message (`dmesg`)

## 2. Enable kernel symbol `CONFIG_INITCALL_DEBUG`

- It enables a printk message every time an initcall is run
- It can be enabled also using the kernel option `initcall_debug` on U-Boot's bootargs

## 3. Print on dmesg any additional user-level output:

- `echo hello > /dev/kmsg`

# How to reduce boot time: u-boot

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

**Boot time**

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1 Disable unused features

# How to reduce boot time: kernel

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time**

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

## 2 Disable computation of loops\_per\_jiffy

- Use kernel option `lpj=xxx` on U-Boot's bootargs

## 3 Disable console output:

- Remove `console=xxx` on bootargs
- Add `quiet` on bootargs

## 4 Use LZO compression

- Less compress ration but faster
- Kernel symbol: `CONFIG_KERNEL_LZO`

## 5 Reduce kernel size to reduce number of read blocks

- Unused features disabled or compiled as modules

# How to reduce boot time: kernel

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time**

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

## 2 Disable computation of loops\_per\_jiffy

- Use kernel option `lpj=xxx` on U-Boot's bootargs

## 3 Disable console output:

- Remove `console=xxx` on bootargs
- Add `quiet` on bootargs

## 4 Use LZO compression

- Less compress ration but faster
- Kernel symbol: `CONFIG_KERNEL_LZO`

## 5 Reduce kernel size to reduce number of read blocks

- Unused features disabled or compiled as modules

# How to reduce boot time: kernel

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time**

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

## 2 Disable computation of loops\_per\_jiffy

- Use kernel option `lpj=xxx` on U-Boot's bootargs

## 3 Disable console output:

- Remove `console=xxx` on bootargs
- Add `quiet` on bootargs

## 4 Use LZO compression

- Less compress ration but faster
- Kernel symbol: `CONFIG_KERNEL_LZO`

## 5 Reduce kernel size to reduce number of read blocks

- Unused features disabled or compiled as modules

# How to reduce boot time: kernel

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
**Boot time**

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

### 2 Disable computation of loops\_per\_jiffy

- Use kernel option `lpj=xxx` on U-Boot's bootargs

### 3 Disable console output:

- Remove `console=xxx` on bootargs
- Add `quiet` on bootargs

### 4 Use LZO compression

- Less compress ration but faster
- Kernel symbol: `CONFIG_KERNEL_LZO`

### 5 Reduce kernel size to reduce number of read blocks

- Unused features disabled or compiled as modules

# How to reduce boot time: filesystem

Kernel

  Intro

  Compiling

  Code

  Debugging

  Linked lists

  Scheduling

  Linux filesystem

  /dev filesystem

  proc filesystem

  sys filesystem

  Syscalls

  Real-Time

  Boot time

Kernel drivers

  Modules

  Memory

  Concurrency

  Char devices

  I/O

  Platforms

  Using sysfs

  Timers

  Int. handlers

  Bottom halves

  Examples

## 6 Use read-only filesystem

- For the read-only part: use ext2 or squashfs (with lzo compression)
- For the read-write part: use tmpfs on RAM

## 7 Disable unused init services

- On Debian use `sysv-rc-conf`

# How to reduce boot time: filesystem

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 6 Use read-only filesystem

- For the read-only part: use ext2 or squashfs (with lzo compression)
- For the read-write part: use tmpfs on RAM

## 7 Disable unused init services

- On Debian use `sysv-rc-conf`

# Outline

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

### 1 Kernel

### 2 Kernel drivers

# Loadable Kernel Modules

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Linux provides the ability of inserting (and removing) services provided by the kernel at runtime
- Every piece of code that can be dynamically loaded (and unloaded) is called **Kernel Module**

# Loadable Kernel Modules (2)

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- A kernel module provides a new service (or services) available to users
- Event-driven programming:
  - Once inserted, a module just registers itself in order to serve future requests
  - The initialization function terminates immediately
- Once a module is loaded and the new service registered
  - The service can be used by all the processes, as long as the module is in memory
  - The module can access all the kernel's public symbols
- After unloading a module, the service is no longer available
- In the 2.6 series, modules have extensions `.ko`

# Loadable Kernel Modules (3)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The kernel core must be self-contained. Everything else can be written as a kernel module
- A kernel module is desirable for:
  - Device drivers
  - Filesystems
  - Network protocols
- Modules can only use **exported** functions (a collection of functions available to kernel developers). The function must already be part of the kernel at the time it is invoked.
- A module can export symbols through the following macros:
  - `EXPORT_SYMBOL(name);`
  - `EXPORT_SYMBOL_GPL(name);`  
makes the symbol available only to GPL-licensed modules

# Loadable Kernel Modules (3)

## Kernel

Intro  
Compiling  
Code

Debugging

Linked lists

Scheduling

Linux filesystem  
`/dev` filesystem

`proc` filesystem  
`sys` filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

### Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The kernel core must be self-contained. Everything else can be written as a kernel module
- A kernel module is desirable for:
  - Device drivers
  - Filesystems
  - Network protocols
- Modules can only use **exported** functions (a collection of functions available to kernel developers). The function must already be part of the kernel at the time it is invoked.
- A module can export symbols through the following macros:
  - **EXPORT\_SYMBOL(name);**
  - **EXPORT\_SYMBOL\_GPL(name);**  
makes the symbol available only to GPL-licensed modules

# Why using a kernel module ?

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

### Modules

Memory  
Concurrency  
Char devices

I/O

Platforms  
Using sysfs  
Timers

Int. handlers

Bottom halves

Examples

- Save memory

- Not all kernel services or features are required every time into the kernel
- A module can be loaded only when it is necessary

- Easier development:

- During a debugging session, a module can be loaded and unloaded several times
- It allows code testing and debugging without rebooting the machine

# How to write a kernel module

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules**
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

## Ways to write a kernel module:

1. Insert the code into the Linux kernel main source tree
  - Modify the **Kconfig** and the main **Makefile**
  - Create a patch for each new kernel version
2. Write the code in a separate directory, without modifying any file in the main source tree
  - More flexible
  - In the 2.6 series, the modules are linked against object files in the main source tree:
    - ⇒ The kernel must be already configured and compiled

# How to write a kernel module

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

## Ways to write a kernel module:

1. Insert the code into the Linux kernel main source tree
  - Modify the `Kconfig` and the main `Makefile`
  - Create a patch for each new kernel version
2. Write the code in a separate directory, without modifying any file in the main source tree
  - More flexible
  - In the 2.6 series, the modules are linked against object files in the main source tree:
    - ⇒ The kernel must be already configured and compiled

# Loading/unloading a module

## Kernel

Intro  
Compiling

Code  
Debugging

Linked lists  
Scheduling

Linux filesystem  
`/dev` filesystem

`proc` filesystem  
`sys` filesystem

Syscalls  
Real-Time

Boot time

## Kernel drivers

### Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using `sysfs`

Timers

Int. handlers

Bottom halves

Examples

- Only the superuser can load and unload modules
- `insmod` inserts a module and its data into the kernel.
- The kernel function `sys_init_module`:
  1. Allocates (through `vmalloc`) memory to hold the module
  2. Copies the module into that memory region
  3. Resolves kernel references in the module via the kernel symbol table (works like the linker `ld`)
  4. Calls the module's initialization function
- `rmmod` removes a loaded module and all its services
- `lsmod` lists modules currently loaded in the kernel
  - Works through `/proc/modules`

# Loading/unloading a module (2)

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- **modprobe** works as **insmod** and **rmmmod**, but it also handle module dependencies.
  - It can only load a module contained in the **/lib/modules/** directory
  - Configuration is stored in **/etc/modprobe.conf**
  - Dependencies are created by **depmod** which generates the **/lib/modules/<kernel version>/modules.dep** file

# The Makefile

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The **Makefile** uses the extended GNU *make* syntax

- Structure of the **Makefile**:

```
## Name of the module:
```

```
obj-m = mymodule.o
```

- Command line:

```
make -C kernel_dir M=$(PWD) modules
```

- See [linux/Documentation/kbuild/modules.txt](https://www.kernel.org/doc/Documentation/kbuild/modules.txt)

# Example 1: the include part

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- We now see how to write a simple module that writes “Hello World” at module insertion/removal
- For a simple module we need to include at least the following

```
#include <linux/module.h>
```

```
#include <linux/kernel.h>
```

```
#include <linux/init.h>
```

that define some essential macros and function prototypes.

# Example 1: the init function

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
**Modules**  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Function called when the module is inserted:

```
static int __init hello_init(void)
{
    printk(KERN_ALERT "Hello world!\n");
    return 0;
}

module_init(hello_init);
```
- The function is defined **static** because it shouldn't be visible outside of the file
- The **\_\_init** token tells the kernel that the memory space occupied by the function can be reclaimed after the module is loaded
  - Similar tag for data: **\_\_initdata**
- The **module\_init** macro specifies which function must be called when the module is inserted

# Example 1: the cleanup function

- The unregister function must remove all the resources allocated by the init function so that the module can be safely unloaded

```
static void __exit hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world!\n");
}

module_exit(hello_exit);
```

- The `__exit` token tells the compiler that the function will be called only during the unloading stage (the compiler puts this function in a special section of the ELF file)
- The `module_exit` macro specifies which function must be called when the module is removed
- It **must** release any resource and undo everything the `init` function built up
- If it is not defined, the kernel does not allow module unloading

# Other information

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Some other information should be specified:
  - `MODULE_AUTHOR("Claudio Scordino");`
  - `MODULE_DESCRIPTION("Kernel Development Example");`
  - `MODULE_VERSION("1.0");`
- License:
  - `MODULE_LICENSE("GPL");`
  - The kernel accepts also "GPL v2", "GPL and additional rights", "Dual BSD/GPL", "Dual MPL/GPL" and "Proprietary"
  - When debugging, use `GPL` to have full features
- Convention: put all information at the end of the file
- This information can be seen through the `modinfo` command

# Other information

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Some other information should be specified:
  - `MODULE_AUTHOR("Claudio Scordino");`
  - `MODULE_DESCRIPTION("Kernel Development Example");`
  - `MODULE_VERSION("1.0");`
- License:
  - `MODULE_LICENSE("GPL");`
  - The kernel accepts also "GPL v2", "GPL and additional rights", "Dual BSD/GPL", "Dual MPL/GPL" and "Proprietary"
  - When debugging, use `GPL` to have full features
- Convention: put all information at the end of the file
- This information can be seen through the `modinfo` command

# Other information

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Some other information should be specified:
  - `MODULE_AUTHOR("Claudio Scordino");`
  - `MODULE_DESCRIPTION("Kernel Development Example");`
  - `MODULE_VERSION("1.0");`
- License:
  - `MODULE_LICENSE("GPL");`
  - The kernel accepts also "GPL v2", "GPL and additional rights", "Dual BSD/GPL", "Dual MPL/GPL" and "Proprietary"
  - When debugging, use `GPL` to have full features
- Convention: put all information at the end of the file
  - This information can be seen through the `modinfo` command

# Other information

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Some other information should be specified:
  - `MODULE_AUTHOR("Claudio Scordino");`
  - `MODULE_DESCRIPTION("Kernel Development Example");`
  - `MODULE_VERSION("1.0");`
- License:
  - `MODULE_LICENSE("GPL");`
  - The kernel accepts also "GPL v2", "GPL and additional rights", "Dual BSD/GPL", "Dual MPL/GPL" and "Proprietary"
  - When debugging, use `GPL` to have full features
- Convention: put all information at the end of the file
- This information can be seen through the `modinfo` command

# Module parameters

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Both `insmod` and `modprobe` accept parameters given at loading time
- Require `#include <linux/moduleparam.h>`
- A module parameter is defined through a macro:

```
static int myvar = 13;  
module_param(myvar, int, SIRUGO);
```

- All parameters should be given a default value
- The last argument is a permission bit-mask (see `linux/stat.h`) for the corresponding file in sysfs
- The macro should be placed outside of any function

# Module parameters (2)

Kernel

  Intro

  Compiling

  Code

  Debugging

  Linked lists

  Scheduling

  Linux filesystem

  /dev filesystem

  proc filesystem

  sys filesystem

  Syscalls

  Real-Time

  Boot time

Kernel drivers

  Modules

  Memory

  Concurrency

  Char devices

  I/O

  Platforms

  Using sysfs

  Timers

  Int. handlers

  Bottom halves

  Examples

- Supported types: `bool`, `charp` (i.e. strings), `int`, `long`,  
`short`, `uint`, `ulong`, `ushort`
- The module can be loaded assigning a value to the parameter myvar:  
`insmod <module_name> myvar=27`
- Another macro allows to accept array parameters:  
`module_param_array(name, type, num, permission);`
  - The module loader refuses to accept more values than will fit in the array

# Memory allocation

- Memory can be allocated through:  
`void *kmalloc (size_t size, int flags);`
- Defined in `<linux/slab.h>`
- Most efficient way of allocate large areas of memory
- Quick (unless it's blocked waiting for memory to be freed)
- It doesn't initialize the allocated area
- The allocated area is contiguous **in physical RAM**
- Linux can allocate only certain predefined sizes
  - The smallest allocation is 32 or 64 bytes (depending on the page size)
  - For completely portable code, don't allocate anything larger than 128 KB
- Allocated memory must be freed using `kfree`
  - It is legal to pass a `NULL` pointer to `kfree`

# Memory allocation

- Memory can be allocated through:  
`void *kmalloc (size_t size, int flags);`
- Defined in `<linux/slab.h>`
- Most efficient way of allocate large areas of memory
- Quick (unless it's blocked waiting for memory to be freed)
- It doesn't initialize the allocated area
- The allocated area is contiguous **in physical RAM**
- Linux can allocate only certain predefined sizes
  - The smallest allocation is 32 or 64 bytes (depending on the page size)
  - For completely portable code, don't allocate anything larger than 128 KB
- Allocated memory must be freed using `kfree`
  - It is legal to pass a `NULL` pointer to `kfree`

# Memory allocation

- Memory can be allocated through:  
`void *kmalloc (size_t size, int flags);`
- Defined in `<linux/slab.h>`
- Most efficient way of allocate large areas of memory
- Quick (unless it's blocked waiting for memory to be freed)
- It doesn't initialize the allocated area
- The allocated area is contiguous **in physical RAM**
- Linux can allocate only certain predefined sizes
  - The smallest allocation is 32 or 64 bytes (depending on the page size)
  - For completely portable code, don't allocate anything larger than 128 KB
- Allocated memory must be freed using **kfree**
  - It is legal to pass a **NULL** pointer to **kfree**

# kmalloc flags

Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
`/dev` filesystem  
`proc` filesystem

`sys` filesystem  
Syscalls

Real-Time  
Boot time

Kernel drivers

Modules

**Memory**

Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers

Int. handlers  
Bottom halves

Examples

- Flags:

## `GFP_ATOMIC`

- Used to allocate memory from interrupt context
- Never blocks

## `GFP_KERNEL`

- Standard allocation: fine for most needs
- Used to allocate memory from process context
- May block
- The function that allocates memory using `GFP_KERNEL` must be reentrant

...

- These flags can be ORed with any of the following flags:

`_GFP_DMA`: DMA-capable memory zone

`_GFP_HIGH`: High-priority request

...

# kmalloc flags

Kernel

Intro  
Compiling

Code  
Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers

Modules

**Memory**

Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers

Int. handlers  
Bottom halves

Examples

- Flags:

## GFP\_ATOMIC

- Used to allocate memory from interrupt context
- Never blocks

## GFP\_KERNEL

- Standard allocation: fine for most needs
- Used to allocate memory from process context
- May block
- The function that allocates memory using GFP\_KERNEL must be reentrant

...

- These flags can be ORed with any of the following flags:

**\_GFP\_DMA**: DMA-capable memory zone

**\_GFP\_HIGH**: High-priority request

...

# Other allocation functions

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

**Memory**

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `void *kzalloc(size_t size, int flags);`
- It zeroes the allocated memory

- `void *vmalloc (unsigned long size);`
- `void vfree (void * addr);`
- Used to obtain contiguous memory zones in **virtual address space**
- Cannot be used with DMA

# Other allocation functions

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `void *kzalloc(size_t size, int flags);`
- It zeroes the allocated memory

- `void *vmalloc (unsigned long size);`
- `void vfree (void * addr);`
- Used to obtain contiguous memory zones in **virtual address space**
- Cannot be used with DMA

# Sources of concurrency

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

The same resources can be accessed by several kernel processes in parallel, causing potential concurrency issues.

Sources of concurrency:

1. Different processes using the same driver at the same time
2. Interrupt handlers invoked when the driver is doing something else
3. Asynchronous mechanisms for delayed code execution: workqueues, tasklets and timers
4. Symmetric MultiProcessors (SMPs): kernel code runs simultaneously on different CPUs
5. Preemptible kernel: kernel code can be preempted by a process that runs the same code

# Race conditions

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Concurrency-related bugs are some of the easiest to create and some of the hardest to find!
- **Race condition:** result of uncontrolled access to shared data

# Example of race condition

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
  /dev filesystem  
  proc filesystem  
  sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
  Examples

What happens if two processes call the following function at the same time ?

```
void * p;  
  
void my_function(void)  
{  
    if (!p)  
        p = kmalloc (1024, GFP_KERNEL);  
    /* ... */  
}
```

# Avoiding race conditions

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Keep concurrency in mind!**

- Kernel and drivers code must be **reentrant**

- Multiple instances can run at the same time in different contexts

- How do we create reentrant code ?

- Avoid shared resources (e.g. global variables) whenever possible
- In other situations, use kernel facilities to set up **critical sections**: code that can be executed by only one thread at a given time

# Avoiding race conditions

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Keep concurrency in mind!

- Kernel and drivers code must be **reentrant**
  - Multiple instances can run at the same time in different contexts

- How do we create reentrant code ?

- Avoid shared resources (e.g. global variables) whenever possible
- In other situations, use kernel facilities to set up **critical sections**: code that can be executed by only one thread at a given time

# Kernel facilities

## Kernel

Intro  
Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The Linux kernel offers a large number of synchronization primitives
  - Some are used only for efficiency reasons: the kernel must reduce to a minimum the time spent waiting for a resource
  - Thus, most synchronization primitives have been introduced to allow some kernel core components to scale well in large Enterprise systems
  
- Device drivers developers can just use
  1. **semaphores** or **mutexes**
  2. **spinlocks** (optionally coupled with interrupt disabling)
  
- Remember to initialize semaphores, mutexes and spinlockes before the protected data are made available to processes!

# Semaphores vs mutexes

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
**Concurrency**  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Binary semaphores can be used for mutual exclusion (i.e. one process at a time)
- There are a set of macros and functions to simplify the use of binary semaphores
- These macros and functions contain the name “mutex”
- Starting from Linux 2.6.16, Ingo Molnar decided to introduce a real mutex data structure for performance and efficiency reasons
- This became the main locking primitive, replacing binary semaphores
- The functions and structures related to this (different) structures contain the name “mutex” as well

# Semaphores vs mutexes

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
**Concurrency**  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Binary semaphores can be used for mutual exclusion (i.e. one process at a time)
- There are a set of macros and functions to simplify the use of binary semaphores
- These macros and functions contain the name “mutex”
- Starting from Linux 2.6.16, Ingo Molnar decided to introduce a real mutex data structure for performance and efficiency reasons
- This became the main locking primitive, replacing binary semaphores
- The functions and structures related to this (different) structures contain the name “mutex” as well

# Semaphores

## Kernel

Intro  
Compiling

Code  
Debugging

Linked lists

Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem

sys filesystem  
Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Semaphores are blocking locks used to protect the resources shared among the processes in the system
- While a process is waiting on a busy semaphore, it is blocked (put in state `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`) and replaced by another runnable process

Since the process is blocked, semaphores can be used only in process context (they cannot be used in interrupt context!)

# Using a semaphore as a mutex

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Include `<asm/semaphore.h>`

2. Allocation + initialization:

```
DECLARE_MUTEX(name);
```

or

```
DECLARE_MUTEX_LOCKED(name);
```

3. Initialization of a semaphore already allocated:

```
struct semaphore sem;
```

```
init_MUTEX(&sem);
```

or

```
init_MUTEX_LOCKED(&sem);
```

# Using a semaphore as a mutex

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

## 4. To acquire the semaphore:

- `void down(struct semaphore *sem);`
- `int down_interruptible(struct semaphore *sem);`
  - Allows the user-space process to be interrupted by the user
  - If interrupted, returns a non-zero value
  - Usage:

```
if(down_interruptible(sem)){  
    /* Undo things */  
    return -ERESTARTSYS;  
}
```
- `int down_trylock(struct semaphore *sem);`
  - Never sleeps
  - If the semaphore is not available returns a nonzero value

## 5. To release the semaphore:

- `void up(struct semaphore *sem);`

# Using a semaphore as a mutex

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
**Concurrency**  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

## 4. To acquire the semaphore:

- `void down(struct semaphore *sem);`
- `int down_interruptible(struct semaphore *sem);`
  - Allows the user-space process to be interrupted by the user
  - If interrupted, returns a non-zero value
  - Usage:

```
if(down_interruptible(sem)){  
    /* Undo things */  
    return -ERESTARTSYS;  
}
```
- `int down_trylock(struct semaphore *sem);`
  - Never sleeps
  - If the semaphore is not available returns a nonzero value

## 5. To release the semaphore:

- `void up(struct semaphore *sem);`

# Mutexes

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Include `<linux/mutex.h>`

2. Allocation + initialization:

```
DEFINE_MUTEX(name);
```

3. Initialization of a mutex already allocated:

```
struct mutex lock;
```

```
mutex_init(&lock);
```

# Using mutexes

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
  - I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

## 4. To lock the mutex:

- `void mutex_lock(struct mutex *lock);`
  - Tries to lock the mutex, sleeps otherwise
- `int mutex_lock_interruptible(struct mutex *lock);`
  - It can be interrupted
  - If interrupted, returns a non-zero value and doesn't hold the lock
- `int mutex_trylock(struct mutex *lock);`
  - Never blocks
- `int mutex_is_locked(struct mutex *lock);`
  - Just tells if the mutex is locked

## 5. To release the mutex:

- `void mutex_unlock(struct mutex *lock);`

# Using mutexes

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
**Concurrency**  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

## 4. To lock the mutex:

- `void mutex_lock(struct mutex *lock);`
  - Tries to lock the mutex, sleeps otherwise
- `int mutex_lock_interruptible(struct mutex *lock);`
  - It can be interrupted
  - If interrupted, returns a non-zero value and doesn't hold the lock
- `int mutex_trylock(struct mutex *lock);`
  - Never blocks
- `int mutex_is_locked(struct mutex *lock);`
  - Just tells if the mutex is locked

## 5. To release the mutex:

- `void mutex_unlock(struct mutex *lock);`

# Spinlocks

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Spinlocks** are used to protect data structures that can be possibly accessed in interrupt context
- A spinlock is a mutex implemented by an atomic variable that can have only two possible values: *locked* and *unlocked*
- When the CPU must acquire a spinlock, it reads the value of the atomic variable and sets it to *locked*. If the variable was already locked before the test-and-set operation, the whole step is repeated (“spinning”)
- The actual implementation is architecture-dependent

# Spinlocks

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

- **Spinlocks** are used to protect data structures that can be possibly accessed in interrupt context
- A spinlock is a mutex implemented by an atomic variable that can have only two possible values: *locked* and *unlocked*
- When the CPU must acquire a spinlock, it reads the value of the atomic variable and sets it to *locked*. If the variable was already locked before the test-and-set operation, the whole step is repeated (“spinning”)
- The actual implementation is architecture-dependent

# Spinlock constraints

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- When using spinlocks it's easy to cause **deadlocks**:
  - Example: if the process holding the spinlock blocks

**Spinlocks can be called from both process and interrupt contexts, but... when holding the spinlock, the code must be atomic (i.e. cannot sleep)**

General rules:

- Spinlocks must be held for the minimum time possible
- Pay attention to not call any blocking function when holding a spinlock!**
- The kernel automatically disables preemption when acquiring the spinlock
- If the data structure protected by the spinlock is accessed in interrupt context, we must disable the interrupts before acquiring the spinlock

# Using spinlocks

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

1. Include `<linux/spinlock.h>`

2. Allocation + initialization:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

3. Initialization of a spinlock already allocated:

```
spin_lock_t my_lock;
```

```
void spin_lock_init(&my_lock); [unlocked]
```

# Using spinlocks (2)

## 4. To lock the spinlock:

- `void spin_lock(spinlock_t *lock);`
- `void spin_trylock(spinlock_t *lock);`
  - Non-blocking function
  - Returns nonzero on success
- `void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);`
  - It disables interrupts on the local CPU
- `void spin_lock_bh(spinlock_t *lock);`
  - It disables software interrupts (hardware interrupts are left enabled)
- `void spin_trylock_bh(spinlock_t *lock);`

## 5. To unlock the spinlock:

- `void spin_unlock(spinlock_t *lock);`
- `void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);`
  - The `flags` argument must be the same passed to `spin_lock_irqsave`
- `void spin_unlock_bh(spinlock_t *lock);`

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
    /dev filesystem  
    proc filesystem  
    sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
Examples

# Using spinlocks (2)

## 4. To lock the spinlock:

- `void spin_lock(spinlock_t *lock);`
- `void spin_trylock(spinlock_t *lock);`
  - Non-blocking function
  - Returns nonzero on success
- `void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);`
  - It disables interrupts on the local CPU
- `void spin_lock_bh(spinlock_t *lock);`
  - It disables software interrupts (hardware interrupts are left enabled)
- `void spin_trylock_bh(spinlock_t *lock);`

## 5. To unlock the spinlock:

- `void spin_unlock(spinlock_t *lock);`
- `void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);`
  - The `flags` argument must be the same passed to `spin_lock_irqsave`
- `void spin_unlock_bh(spinlock_t *lock);`

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
    /dev filesystem  
    proc filesystem  
    sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
  Examples

# General rules to avoid deadlocks

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
**Concurrency**  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

1. Neither semaphores nor spinlocks allow a lock holder to acquire the lock a second time
2. Document assumptions about how calling a function (e.g. this function must be called holding that lock)
3. When multiple locks must be acquired, they should always be acquired in the same order
  - First, take the lock local to your code, then the lock belonging to a more general part of the kernel
4. If you have a combination of semaphores and spinlocks, you must obtain the semaphores first
  - `down` can sleep, so it cannot be called while holding a spinlock

# General rules to avoid deadlocks

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

1. Neither semaphores nor spinlocks allow a lock holder to acquire the lock a second time
2. Document assumptions about how calling a function (e.g. this function must be called holding that lock)
3. When multiple locks must be acquired, they should always be acquired in the same order
  - First, take the lock local to your code, then the lock belonging to a more general part of the kernel
4. If you have a combination of semaphores and spinlocks, you must obtain the semaphores first
  - `down` can sleep, so it cannot be called while holding a spinlock

# General rules to avoid deadlocks

## Kernel

[Intro](#)

[Compiling](#)

[Code](#)

[Debugging](#)

[Linked lists](#)

[Scheduling](#)

[Linux filesystem](#)

[/dev filesystem](#)

[proc filesystem](#)

[sys filesystem](#)

[Syscalls](#)

[Real-Time](#)

[Boot time](#)

## Kernel drivers

[Modules](#)

[Memory](#)

[Concurrency](#)

[Char devices](#)

[I/O](#)

[Platforms](#)

[Using sysfs](#)

[Timers](#)

[Int. handlers](#)

[Bottom halves](#)

[Examples](#)

1. Neither semaphores nor spinlocks allow a lock holder to acquire the lock a second time
2. Document assumptions about how calling a function (e.g. this function must be called holding that lock)
3. When multiple locks must be acquired, they should always be acquired in the same order
  - First, take the lock local to your code, then the lock belonging to a more general part of the kernel
4. If you have a combination of semaphores and spinlocks, you must obtain the semaphores first
  - `down` can sleep, so it cannot be called while holding a spinlock

# General rules to avoid deadlocks

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

1. Neither semaphores nor spinlocks allow a lock holder to acquire the lock a second time
2. Document assumptions about how calling a function (e.g. this function must be called holding that lock)
3. When multiple locks must be acquired, they should always be acquired in the same order
  - First, take the lock local to your code, then the lock belonging to a more general part of the kernel
4. If you have a combination of semaphores and spinlocks, you must obtain the semaphores first
  - **down** can sleep, so it cannot be called while holding a spinlock

# Atomic variables

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Sometimes the shared resource is a simple integer variable
  - Even an instruction like `i++` is not guaranteed to be atomic on all architectures!
- In such cases, a full locking scheme seems like overhead
- The kernel provides an atomic integer called `atomic_t`
- Defined in `<asm/atomic.h>`
- It holds an `int` value, but only the first 24 bits can be trustfully used

# Atomic variables (2)

Kernel

Intro  
Compiling  
Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- These variables must be accessed only through these functions:

- `void atomic_set(atomic_t *v, int i);`
- `atomic_t v = ATOMIC_INIT(0);`
  
- `int atomic_read(atomic_t *v);`
- `void atomic_add(int i, atomic_t *v);`
- `void atomic_sub(int i, atomic_t *v);`
- `void atomic_inc(atomic_t *v);`
- `void atomic_dec(atomic_t *v);`
- `int atomic_add_return(int i, atomic_t *v);`
- `int atomic_sub_return(int i, atomic_t *v);`
- `int atomic_inc_return(atomic_t *v);`
- `int atomic_dec_return(atomic_t *v);`
- `int atomic_inc_and_test(atomic_t *v);`
- `int atomic_dec_and_test(atomic_t *v);`
- `int atomic_sub_and_test(int i, atomic_t *v);`

# Bit operations

- The `atomic_t` type doesn't work for manipulating bits in an atomic manner
- The kernel offers a set of functions that modify or test single bits atomically
  - Guaranteed to be atomic even on SMP computers
- Operations are performed using a single machine instruction
- The functions are architecture-dependent and are declared in `<asm/bitops.h>`
  - Arguments type is architecture-dependent too
- Available operations:
  - `void set_bit(nr, addr);`
  - `void clear_bit(nr, addr);`
  - `void change_bit(nr, addr);`
  - `int test_bit(nr, addr);`
  - `int test_and_set_bit(nr, addr);`
  - `int test_and_clear_bit(nr, addr);`
  - `int test_and_change_bit(nr, addr);`

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Bit operations

- The `atomic_t` type doesn't work for manipulating bits in an atomic manner
- The kernel offers a set of functions that modify or test single bits atomically
  - Guaranteed to be atomic even on SMP computers
- Operations are performed using a single machine instruction
- The functions are architecture-dependent and are declared in `<asm/bitops.h>`
  - Arguments type is architecture-dependent too
- Available operations:
  - `void set_bit(nr, addr);`
  - `void clear_bit(nr, addr);`
  - `void change_bit(nr, addr);`
  - `int test_bit(nr, addr);`
  - `int test_and_set_bit(nr, addr);`
  - `int test_and_clear_bit(nr, addr);`
  - `int test_and_change_bit(nr, addr);`

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Introduction to char devices

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
**Char devices**  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- **dev\_t type:**
  - Defined in `<linux/types.h>`
  - Holds device numbers (i.e. major + minor numbers)
  
- Never make any assumption about internal representation of `dev_t`!
- Use macros defined in `<linux/kdev_t.h>`:
  - `dev_t` → device numbers:
    - `MAJOR (dev_t dev);`
    - `MINOR (dev_t dev);`
  - Device numbers → `dev_t`:
    - `MKDEV (int major, int minor);`
  - To print device numbers:
    - `int print_dev_t(char *buffer, dev_t dev);`
    - `char *format_dev_t(char *buffer, dev_t dev);`

# Introduction to char devices

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `dev_t` type:
  - Defined in `<linux/types.h>`
  - Holds device numbers (i.e. major + minor numbers)
- Never make any assumption about internal representation of `dev_t`!
- Use macros defined in `<linux/kdev_t.h>`:
  - `dev_t` → device numbers:
    - `MAJOR (dev_t dev);`
    - `MINOR (dev_t dev);`
  - Device numbers → `dev_t`:
    - `MKDEV (int major, int minor);`
  - To print device numbers:
    - `int print_dev_t(char *buffer, dev_t dev);`
    - `char *format_dev_t(char *buffer, dev_t dev);`

# Device numbers allocation

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices**
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

- How do we allocate device numbers at kernel-level ?
- Two options:
  1. If we know device numbers ahead of time:  
`register_chrdev_region(...)`
  2. If we let the kernel choose device numbers:  
`alloc_chrdev_region(...)`
- **Important:** these functions register device numbers only at kernel level!
  - At user-level an entry in the `dev/` directory must explicitly be created using `mknod` or by `udev`

# Device numbers allocation

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- How do we allocate device numbers at kernel-level ?
- Two options:
  1. If we know device numbers ahead of time:  
`register_chrdev_region(...)`
  2. If we let the kernel choose device numbers:  
`alloc_chrdev_region(...)`
- **Important:** these functions register device numbers only at kernel level!
  - At user-level an entry in the `dev/` directory must explicitly be created using `mknod` or by `udev`

# Device numbers allocation

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- How do we allocate device numbers at kernel-level ?
- Two options:
  1. If we know device numbers ahead of time:  
`register_chrdev_region(...)`
  2. If we let the kernel choose device numbers:  
`alloc_chrdev_region(...)`
- **Important:** these functions register device numbers only at kernel level!
  - At user-level an entry in the `dev/` directory must explicitly be created using `mknod` or by `udev`

# Device numbers (manual) allocation

Kernel

Intro  
Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
int register_chrdev_region (dev_t first,  
                           unsigned int count,  
                           char * name)
```

- **first**: beginning device number of the range we want to allocate
- **count**: total number of contiguous device numbers requested
- **name**:
  - Name of device that should be associated with this range
  - It will appear in `/proc/devices` and sysfs
- **As with most kernel function: return value 0 means success**

# Device numbers (manual) allocation

Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists

Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem

sys filesystem  
Syscalls

Real-Time  
Boot time

Kernel drivers

Modules

Memory

Concurrency  
Char devices

I/O

Platforms  
Using sysfs

Timers

Int. handlers  
Bottom halves

Examples

```
int register_chrdev_region (dev_t first,  
                           unsigned int count,  
                           char * name)
```

- **first**: beginning device number of the range we want to allocate
- **count**: total number of contiguous device numbers requested
- **name**:
  - Name of device that should be associated with this range
  - It will appear in `/proc/devices` and sysfs
- **As with most kernel function: return value 0 means success**

# Device numbers de-allocation

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
int unregister_chrdev_region (dev_t first,  
                           unsigned int count)
```

- Usually put in module's cleanup function

# but... which major number for our device ?

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **Problem:** some major numbers are statically assigned to common devices
- See [Documentation/devices.txt](#)
- Registered devices are shown in [/proc/devices](#)

# Dynamic allocation of device numbers

```
int alloc_chrdev_region (dev_t * dev,  
                        unsigned int firstminor,  
                        unsigned int count,  
                        char * name)
```

- **dev**: output-only parameter that will contain the first number of the allocated range
- **firstminor**: requested first minor number (usually 0)
- Need a mechanism to export the assigned values to user-level (where `mknod` can create an entry in `/dev`)
  - We can't know the values in advance
  - Such values can be read from `/proc/devices`
  - Write a simple script at user-level that parses `/proc/devices` and creates the proper entry in `/dev` through `mknod`
  - Example: `mknod /dev/mydevice c 234 0`

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Dynamic allocation of device numbers

```
int alloc_chrdev_region (dev_t * dev,  
                        unsigned int firstminor,  
                        unsigned int count,  
                        char * name)
```

- `dev`: output-only parameter that will contain the first number of the allocated range
- `firstminor`: requested first minor number (usually 0)
- Need a mechanism to export the assigned values to user-level (where `mknod` can create an entry in `/dev`)
  - We can't know the values in advance
  - Such values can be read from `/proc/devices`
  - Write a simple script at user-level that parses `/proc/devices` and creates the proper entry in `/dev` through `mknod`
  - Example: `mknod /dev/mydevice c 234 0`

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# File operations

## Kernel

Intro  
Compiling

Code  
Debugging

Linked lists  
Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- We still have to connect driver's operations to device numbers

- struct file\_operations

- Defined in <linux/fs.h>
- Collection of function pointers
  - Each pointer points to the implementation of a syscall (e.g. `open()`, `read()`, `close()`)
  - Object-oriented programming
  - `NULL` for unsupported operations
- Convention: `file_operations` structures or their pointers called `fops`

# File operations

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- We still have to connect driver's operations to device numbers

- **struct file\_operations**

- Defined in `<linux/fs.h>`
- Collection of function pointers
  - Each pointer points to the implementation of a syscall (e.g. `open()`, `read()`, `close()`)
  - Object-oriented programming
  - `NULL` for unsupported operations
- Convention: `file_operations` structures or their pointers called `fops`

# file\_operations fields

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

- **struct module \* owner**
  - Pointer to the module that “owns” the structure
  - Initialized to **THIS\_MODULE**, a macro defined in `<linux/module.h>`
- **int (\*open) (struct inode \*, struct file \*)**
  - If `NULL`, opening always succeeds
- **int (\*release) (struct inode \*, struct file \*)**

# file\_operations fields

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
  - I/O
  - Platforms
  - Using sysfs
  - Timers
  - Int. handlers
  - Bottom halves
- Examples

- struct module \* owner
  - Pointer to the module that “owns” the structure
  - Initialized to `THIS_MODULE`, a macro defined in `<linux/module.h>`
- int (\*open) (struct inode \*, struct file \*)
  - If `NULL`, opening always succeeds
- int (\*release) (struct inode \*, struct file \*)

# file\_operations fields

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- struct module \* owner
  - Pointer to the module that “owns” the structure
  - Initialized to `THIS_MODULE`, a macro defined in `<linux/module.h>`
- int (\*open) (struct inode \*, struct file \*)
  - If `NULL`, opening always succeeds
- int (\*release) (struct inode \*, struct file \*)

## file\_operations fields (2)

- `ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)`
  - Called when user-space reads from the device
  - A non-negative return value represents the number of bytes successfully read
- `ssize_t (*write) (struct file *, char __user *, size_t, loff_t *)`
  - Called when user-space writes to the device
  - A non-negative return value represents the number of bytes successfully written
- `int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)`
  - Used to issue device-specific commands
- `__user`: annotation string to find misuse of user-space addresses

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
Linux filesystem  
  /dev filesystem  
  proc filesystem  
sys filesystem  
  Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
Char devices  
  I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

## file\_operations fields (2)

- `ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)`
  - Called when user-space reads from the device
  - A non-negative return value represents the number of bytes successfully read
- `ssize_t (*write) (struct file *, char __user *, size_t, loff_t *)`
  - Called when user-space writes to the device
  - A non-negative return value represents the number of bytes successfully written
- `int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)`
  - Used to issue device-specific commands
- `__user`: annotation string to find misuse of user-space addresses

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
  /dev filesystem  
  proc filesystem  
  sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
  Examples

## file\_operations fields (2)

- `ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)`
  - Called when user-space reads from the device
  - A non-negative return value represents the number of bytes successfully read
- `ssize_t (*write) (struct file *, char __user *, size_t, loff_t *)`
  - Called when user-space writes to the device
  - A non-negative return value represents the number of bytes successfully written
- `int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)`
  - Used to issue device-specific commands
- `__user`: annotation string to find misuse of user-space addresses

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
  /dev filesystem  
  proc filesystem  
  sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
  Examples

## file\_operations fields (2)

- `ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)`
  - Called when user-space reads from the device
  - A non-negative return value represents the number of bytes successfully read
- `ssize_t (*write) (struct file *, char __user *, size_t, loff_t *)`
  - Called when user-space writes to the device
  - A non-negative return value represents the number of bytes successfully written
- `int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)`
  - Used to issue device-specific commands
- `__user`: annotation string to find misuse of user-space addresses

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
  /dev filesystem  
  proc filesystem  
  sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
  Examples

# file\_operations initialization

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

Standard C tagged structure initialization syntax:

```
struct file_operations fops = {
    .owner =      THIS_MODULE,
    .read =       my_read_function,
    .write =      my_write_function,
    .open =       my_open_function,
    .release =    my_release_function,
};
```

# struct file

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- It is the second most important data structure used in device drivers
- Defined in `<linux/fs.h>`
- Represents an open file descriptor
  - It is not specific to device drivers: every open file in the system has an associated `struct file` in kernel space
  - Nothing to do with the `FILE` pointers of user-space
- It is created by the kernel on `open` and is passed to any function that operates on the file, until the last `close`
  - After all instances of the file are closed, the kernel releases the data structure
- Convention: `file` structures or their pointers called `filp`

# struct file

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem  
/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- It is the second most important data structure used in device drivers
- Defined in `<linux/fs.h>`
- Represents an open file descriptor
  - It is not specific to device drivers: every open file in the system has an associated `struct file` in kernel space
  - Nothing to do with the `FILE` pointers of user-space
- It is created by the kernel on `open` and is passed to any function that operates on the file, until the last `close`
  - After all instances of the file are closed, the kernel releases the data structure
- Convention: `file` structures or their pointers called `filp`

# struct file

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem  
/dev filesystem

proc filesystem  
sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- It is the second most important data structure used in device drivers
- Defined in `<linux/fs.h>`
- Represents an open file descriptor
  - It is not specific to device drivers: every open file in the system has an associated `struct file` in kernel space
  - Nothing to do with the `FILE` pointers of user-space
- It is created by the kernel on `open` and is passed to any function that operates on the file, until the last `close`
  - After all instances of the file are closed, the kernel releases the data structure
- Convention: `file` structures or their pointers called `filp`

# file fields

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
**Char devices**  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

### ● mode\_t f\_mode

- It identifies the file as either readable (`FMODE_READ`) or writable (`FMODE_WRITE`), or both
- Used to check permissions in `open` or `ioctl`
- Not used in `read` and `write`, before check is done by the kernel

### ● loff\_t f\_pos

- Current position
- 64-bit value (i.e. `long long`)
- `read` and `write` methods should update a position using the pointer they receive as last argument instead of acting on this value directly

### ● unsigned int f\_flags

- File flags, defined in `<linux/fcntl.h>`
- Examples: `O_RDONLY`, `O_NONBLOCK`, `O_SYNC`

# file fields

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- mode\_t f\_mode

- It identifies the file as either readable (`FMODE_READ`) or writable (`FMODE_WRITE`), or both
- Used to check permissions in `open` or `ioctl`
- Not used in `read` and `write`, before check is done by the kernel

- loff\_t f\_pos

- Current position
- 64-bit value (i.e. `long long`)
- `read` and `write` methods should update a position using the pointer they receive as last argument instead of acting on this value directly

- unsigned int f\_flags

- File flags, defined in `<linux/fcntl.h>`
- Examples: `O_RDONLY`, `O_NONBLOCK`, `O_SYNC`

# file fields

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
**Char devices**  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

### ● mode\_t f\_mode

- It identifies the file as either readable (`FMODE_READ`) or writable (`FMODE_WRITE`), or both
- Used to check permissions in `open` or `ioctl`
- Not used in `read` and `write`, before check is done by the kernel

### ● loff\_t f\_pos

- Current position
- 64-bit value (i.e. `long long`)
- `read` and `write` methods should update a position using the pointer they receive as last argument instead of acting on this value directly

### ● unsigned int f\_flags

- File flags, defined in `<linux/fcntl.h>`
- Examples: `O_RDONLY`, `O_NONBLOCK`, `O_SYNC`

# file fields (2)

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices**
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

- **struct file\_operations \*f\_op**

- Operations associated with the file
- Can be overwritten (i.e. “method overriding” of object-oriented programming)

- **void \*private\_data**

- Can be used to allocate driver’s specific data
- In this case, remember to free memory in the `release` method

# file fields (2)

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
  - I/O
  - Platforms
  - Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

- `struct file_operations *f_op`

- Operations associated with the file
- Can be overwritten (i.e. “method overriding” of object-oriented programming)

- `void *private_data`

- Can be used to allocate driver's specific data
- In this case, remember to free memory in the `release` method

# `struct inode`

Kernel

  Intro

  Compiling

  Code

  Debugging

  Linked lists

  Scheduling

  Linux filesystem

  /dev filesystem

  proc filesystem

  sys filesystem

  Syscalls

  Real-Time

  Boot time

Kernel drivers

  Modules

  Memory

  Concurrency

  Char devices

  I/O

  Platforms

  Using sysfs

  Timers

  Int. handlers

  Bottom halves

  Examples

- Used by the kernel internally to represent files
- Fields:
  - `dev_t i_rdev`
    - For inodes that represent device files, it contains the device number
  - `struct cdev *i_cdev`
    - Kernel's internal structure that represents char devices (see next slides)
- Macros to get minor and major numbers:
  - `unsigned int iminor (struct inode *inode);`
  - `unsigned int imajor (struct inode *inode);`

# Char devices registration

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
**Char devices**  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Kernel uses structure `cdev` to represent char devices internally
  - Defined in `<linux/cdev.h>`
- Before kernel invokes device's operations we must initialize cdevs

### 1. Allocation + initialization:

```
struct cdev * my_cdev_p = cdev_alloc();  
my_cdev_p->fops = &fops;
```

### 2. Initialization of `cdev` already allocated:

```
struct cdev my_cdev;  
cdev_init (&my_cdev, &fops);
```

# Char devices registration

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
**Char devices**  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Kernel uses structure `cdev` to represent char devices internally
  - Defined in `<linux/cdev.h>`
- Before kernel invokes device's operations we must initialize cdevs

### 1. Allocation + initialization:

```
struct cdev * my_cdev_p = cdev_alloc();  
my_cdev_p->fops = &fops;
```

### 2. Initialization of `cdev` already allocated:

```
struct cdev my_cdev;  
cdev_init (&my_cdev, &fops);
```

# Char devices registration (2)

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices**
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

- Like `file_operations`, `cdev` has an `owner` field that should be set to `THIS_MODULE`:

```
my_cdev.owner = THIS_MODULE
```

- Final step:

```
int cdev_add (struct cdev * dev,  
              dev_t num,  
              unsigned int count);
```

- It can fail: if it returns a negative value, the device has not been added!

- To remove a char device:

```
void cdev_del (struct cdev * dev);
```

# Char devices registration (2)

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
  - I/O
  - Platforms
  - Using sysfs
  - Timers
  - Int. handlers
  - Bottom halves
- Examples

- Like `file_operations`, `cdev` has an `owner` field that should be set to `THIS_MODULE`:

```
my_cdev.owner = THIS_MODULE
```

- Final step:

```
int cdev_add (struct cdev * dev,  
              dev_t num,  
              unsigned int count);
```

- It can fail: if it returns a negative value, the device has not been added!

- To remove a char device:

```
void cdev_del (struct cdev * dev);
```

# Char devices registration (2)

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices**
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

- Like `file_operations`, `cdev` has an `owner` field that should be set to `THIS_MODULE`:

```
my_cdev.owner = THIS_MODULE
```

- Final step:

```
int cdev_add (struct cdev * dev,  
              dev_t num,  
              unsigned int count);
```

- It can fail: if it returns a negative value, the device has not been added!

- To remove a char device:

```
void cdev_del (struct cdev * dev);
```

# An example

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
dev_t devno = MKDEV (EVI_MAJOR, EVI_MINOR);

struct evi_dev {
    struct semaphore sem; /* Mutual exclusion */
    struct cdev cdev;    /* Char device */
    /* ... */           /* Other fields */
};

struct file_operations fops = {
    .owner =      THIS_MODULE,
    .read =       my_read_function,
    .write =      my_write_function,
    .open =       my_open_function,
    .release =    my_release_function,
};
```

# An example (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
static void evi_cdev_setup (struct evi_dev * dev)
{
    register_chrdev_region(devno, 1, "mydriver");
    cdev_init (&dev->cdev, &fops);
    dev->cdev.owner = THIS_MODULE;
    cdev_add (&dev->cdev, devno, 1);
}

static void evi_cdev_cleanup (struct evi_dev * dev)
{
    cdev_del (&dev->cdev);
    unregister_chrdev_region(devno, 1);
}
```

# The open method

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Prototype: `int (*open)(struct inode *inode, struct file *filp);`
- In most drivers, it should perform the following operations:
  1. Identify which device has been opened
    - Through `inode->i_cdev`
    - Through `imminor(inode);`
  2. Check for device-specific errors (e.g. device not ready)
  3. Initialize the device if opened for the first time
  4. Update the `f_op` pointer, if necessary
  5. Allocate and fill any data structure to be put in `filp->private_data`

# The open method (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Normally, we don't need the `cdev` structure, but the `evi_dev` structure which contains such structure
- Linux provides a useful macro:
  - `container_of(pointer, container_type, container_field);`
  - Defined in `<linux/kernel.h>`
- Once the pointer to the `evi_dev` structure has been found, usually it is stored on the `private_data` field of `file` for easier access in future

# An example

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
static int evi_open (struct inode *inode, struct file *filp)
{
    struct evi_dev *dev;
    dev = container_of(inode->i_cdev, struct evi_dev, cdev);
    filp->private_data = dev; /* For future access */
    return 0;
}

static ssize_t evi_read (struct file *filp, char __user *buf,
                      size_t count, loff_t *f_pos)
{
    struct evi_data *dev = filp->private_data;
    size_t min;
    /*...*/
}
```

# The release method

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
**Char devices**  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The role of this method is the reverse of `open`
- Sometimes called `close` instead of `release`
- The name `release` is more correct, because not all `close` syscalls invoke this method
  - The kernel keeps a counter of how many times a `file` structure is being used
  - This method is called only when the counter reaches 0
- In most drivers, it should perform the following operations:
  1. Deallocate anything allocated by `open` in `filp->private_data`
  2. Shutdown the device on the last close

# The release method

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
**Char devices**  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The role of this method is the reverse of `open`
- Sometimes called `close` instead of `release`
- The name `release` is more correct, because not all `close` syscalls invoke this method
  - The kernel keeps a counter of how many times a `file` structure is being used
  - This method is called only when the counter reaches 0
- In most drivers, it should perform the following operations:
  1. Deallocate anything allocated by `open` in `filp->private_data`
  2. Shutdown the device on the last close

# read and write methods

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

Both perform a similar task: copying data from/to application code

Their prototypes are pretty similar:

- `ssize_t read(struct file *filp, char __user *buff, size_t count, loff_t *offp);`
- `ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *offp);`

`filp`: file pointer

`count`: size of requested data

`buff`: **user-space** pointer to the buffer

`offp`: pointer to a “long offset” type which specifies the file position

- It should be updated by both methods

# read and write methods (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Returned value: number of bytes actually read/written
- In case of error, they return a negative number
  - Examples: **-EINTR** (interrupted system call), **-ENOMEM** (no memory), **-EFAULT** (bad address)
  - User-space always sees **-1** and need to access the **errno** variable to find out what happened

# Accessing user-space memory

## Kernel

Intro  
Compiling

Code  
Debugging

Linked lists

Scheduling  
Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- In driver code, you can't just `memcpy` between an address supplied by user-space and the address of a buffer in kernel-space
  - They correspond to completely different address spaces (thanks to virtual memory)
  - Therefore, the `buff` pointer cannot be directly dereferenced by kernel code!
  
- Few reasons for this restriction:
  1. The pointer may not be valid when running in kernel mode (i.e. no mapping)
  2. The buffer may not be resident in RAM (user-space memory is paged and swapped)
  3. The pointer may be buggy or malicious

# Accessing user-space memory

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- In driver code, you can't just `memcpy` between an address supplied by user-space and the address of a buffer in kernel-space
  - They correspond to completely different address spaces (thanks to virtual memory)
  - Therefore, the `buff` pointer cannot be directly dereferenced by kernel code!
- Few reasons for this restriction:
  1. The pointer may not be valid when running in kernel mode (i.e. no mapping)
  2. The buffer may not be resident in RAM (user-space memory is paged and swapped)
  3. The pointer may be buggy or malicious

# Accessing user-space memory (2)

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The access must always be performed by special functions in order to be safe:
  - The read method must use:  
`unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);`
  - The write method must use:  
`unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);`
- These calls also check if pointer is valid
  - `__copy_to_user` and `__copy_from_user` do not perform checks
- These calls can sleep (e.g. data on swap space)
  - Cannot be called from interrupt context
  - The code must be reentrant

# Communicating with Hardware

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
**I/O**

Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Every I/O peripheral is controlled by writing and reading its registers and memories

- There are 2 ways of accessing I/O registers:

### 1. Port-mapped I/O (PMIO):

- Separate address space for memory and I/O
- Few CPU manufacturers (e.g., x86)
- Concept of **I/O ports**

### 2. Memory-mapped I/O (MMIO):

- Same address space for memory and I/O
- Most CPU manufacturers (e.g., ARM)
- Memory-mapped registers and device memory
- The difference between registers and memory is transparent to software
- Concept of **I/O memory**

# Communicating with Hardware

Kernel

Intro  
Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Every I/O peripheral is controlled by writing and reading its registers and memories

- There are 2 ways of accessing I/O registers:

## 1. Port-mapped I/O (PMIO):

- Separate address space for memory and I/O
- Few CPU manufacturers (e.g., x86)
- Concept of **I/O ports**

## 2. Memory-mapped I/O (MMIO):

- Same address space for memory and I/O
- Most CPU manufacturers (e.g., ARM)
- Memory-mapped registers and device memory
- The difference between registers and memory is transparent to software
- Concept of **I/O memory**

# Communicating with Hardware

Kernel

Intro  
Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Every I/O peripheral is controlled by writing and reading its registers and memories

- There are 2 ways of accessing I/O registers:

1. Port-mapped I/O (PMIO):

- Separate address space for memory and I/O
- Few CPU manufacturers (e.g., x86)
- Concept of **I/O ports**

2. Memory-mapped I/O (MMIO):

- Same address space for memory and I/O
- Most CPU manufacturers (e.g., ARM)
- Memory-mapped registers and device memory
- The difference between registers and memory is transparent to software
- Concept of **I/O memory**

# I/O Ports: Allocation

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- First step: register exclusive access to the I/O port(s):

```
struct resource *request_region(unsigned long first,  
                               unsigned long n, const char *name);
```

- **first**: first port requested
- **n**: number of ports requested
- **name**: name of the device

- Defined in `<linux/ioport.h>`

- Return value: non-NULL in case of success

- All current allocations shown in `/proc/ioports`

- To deallocate a port:

```
void release_region(unsigned long start, unsigned long n);
```

# I/O Ports: Allocation

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

## I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- First step: register exclusive access to the I/O port(s):  
`struct resource *request_region(unsigned long first,  
unsigned long n, const char *name);`
  - `first`: first port requested
  - `n`: number of ports requested
  - `name`: name of the device
- Defined in `<linux/ioport.h>`
- Return value: non-NULL in case of success
- All current allocations shown in `/proc/ioports`
- To deallocate a port:  
`void release_region(unsigned long start, unsigned long n);`

# I/O Ports: Manipulation

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

## I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- I/O ports are different from memory
- We have 8-bit, 16-bit and 32-bit ports
- Generally, you can't read a 16-bit value to read two consecutive 8-bit ports
- The driver must call different functions to access different size ports

# I/O Ports: Manipulation (2)

Kernel

Intro  
Compiling  
Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

`inb (port);`

`outb (byte, port);`

- Read/write a 8-bit port

`inw (port);`

`outw (word, port);`

- Read/write a 16-bit port

`inl (port);`

`outl (longword, port);`

- Read/write a 32-bit port

- The type of arguments of these functions is platform-dependent

- See `asm/io.h`

- No 64-bit operations are currently defined

- 16 and 32-bit functions swap data if the target and the device have different ordering rules

# I/O Ports: Paused I/O

- If the bus is too slow, the CPU may transfer data too quickly
- Solution: use pausing functions:

`inb_p (port);`

`outb_p (byte, port);`

- Read/write a 8-bit port

`inw_p (port);`

`outw_p (word, port);`

- Read/write a 16-bit port

`inl_p (port);`

`outl_p (longword, port);`

- Read/write a 32-bit port

- Defined for most architectures

- Sometimes they expand to the same code as nonpausing I/O

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# I/O Ports: String operations

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Some processors implement instructions to transfer a sequence of bytes, words or longs
- Called “string” instructions

`insb (port, address, count);`

`outsb (port, address, count);`

- Read/write `count` bytes

`insw (port, address, count);`

`outsw (port, address, count);`

- Read/write `count` words

`insl (port, address, count);`

`outsl (port, address, count);`

- Read/write `count` 32-bit values

- **Attention:** 16 and 32-bit functions don't swap data if the target and the device have different ordering rules

# I/O Memory: Allocation

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists

Scheduling  
Linux filesystem  
/dev filesystem

proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules

Memory  
Concurrency  
Char devices

## I/O

Platforms  
Using sysfs

Timers

Int. handlers  
Bottom halves

Examples

- First step: register exclusive access to I/O memory:

```
struct resource *request_mem_region(unsigned long start,  
                                     unsigned long len, char *name);
```

- start**: start address
- len**: number of bytes
- name**: name of the device

- Defined in `<linux/ioport.h>`

- Return value: non-NULL in case of success

- All current allocations shown in `/proc/iomem`

- To deallocate a memory region:

```
void release_mem_region(unsigned long start, unsigned long len);
```

# I/O Memory: Allocation

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- First step: register exclusive access to I/O memory:

```
struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);
```

- **start**: start address
- **len**: number of bytes
- **name**: name of the device

- Defined in `<linux/ioport.h>`

- Return value: non-NULL in case of success

- All current allocations shown in `/proc/iomem`

- To deallocate a memory region:

```
void release_mem_region(unsigned long start, unsigned long len);
```

# I/O Memory: Remapping

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Before being used, the memory region must be mapped to virtual memory using

```
void *ioremap(unsigned long phys_addr, unsigned long size);
```

- Defined in `<asm/io.h>`
- Caution: check that it returns a non-NUL address
- To release, use

```
void *iounmap(void * address);
```

# I/O Memory: Remapping

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Before being used, the memory region must be mapped to virtual memory using

```
void *ioremap(unsigned long phys_addr, unsigned long size);
```

- Defined in `<asm/io.h>`
- Caution: check that it returns a non-NUL address

- To release, use

```
void *iounmap(void * address);
```

# I/O Memory: Manipulation

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
  - Platforms
  - Using sysfs
  - Timers
  - Int. handlers
  - Bottom halves
- Examples

```
unsigned int ioread8 (void *addr);  
unsigned int ioread16 (void *addr);  
unsigned int ioread32 (void *addr);
```

- Read 8,16,32-bits regions
- **addr** is an address obtained from `ioremap()`

```
void iowrite8(u8 value, void *addr);  
void iowrite16(u16 value, void *addr);  
void iowrite32(u32 value, void *addr);
```

- Write 8,16,32-bits regions

# I/O Memory: Manipulation (2)

## Kernel

Intro  
Compiling

Code  
Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

## I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Repeating versions of read and write
- Read or write `count` values from the given `buf` to the given `addr`:

- `void ioread8_rep(void *addr, void *buf, unsigned long count);`
- `void ioread16_rep(void *addr, void *buf, unsigned long count);`
- `void ioread32_rep(void *addr, void *buf, unsigned long count);`
- `void write8_rep(void *addr, void *buf, unsigned long count);`
- `void write16_rep(void *addr, void *buf, unsigned long count);`
- `void write32_rep(void *addr, void *buf, unsigned long count);`

# I/O Memory: Manipulation (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Functions that operate on a block of I/O memory:

- `void memset_io(void *addr, u8 value, unsigned int count);`
- `void memcpy_fromio(void *dest, void *source, unsigned int count);`
- `void memcpy_toio(void *dest, void *source, unsigned int count);`

# I/O Memory: Deprecated functions

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Old set of functions:

- `unsigned readb (address);`
- `unsigned readw (address);`
- `unsigned readl (address);`
- `void writeb (unsigned value, address);`
- `void writew (unsigned value, address);`
- `void writel (unsigned value, address);`

- They still work, but their use in new code is discouraged

# The Platform Model

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Refined way to attach devices to drivers
- Device's physical address is independent of the driver
- Prevents resource conflicts
- Improves portability and startup ordering
- Integrates with `sysfs` and power management

# The Platform Model (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Platform devices are all devices that are physically located on the system board
  - This includes all legacy devices and host bridges to peripheral buses
- Devices appear as autonomous devices in the system responding to I/O requests on hardcoded ports
- Drivers for these devices perform device discovery and immediately bind to the devices

# The Platform Model (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The driver

- Knows how to control the device given its location
- Never knows where the device is until `probe()`

- The device

- Informs the driver of the device's location via "resources"
- Drivers and devices are associated using names or ids
- The probe function matches each device with its own driver through a `name`

# The Platform Model (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The driver

- Knows how to control the device given its location
- Never knows where the device is until `probe()`

- The device

- Informs the driver of the device's location via "resources"

- Drivers and devices are associated using names or ids
- The probe function matches each device with its own driver through a `name`

# The Platform Model (3)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- The driver

- Knows how to control the device given its location
- Never knows where the device is until `probe()`

- The device

- Informs the driver of the device's location via "resources"

- Drivers and devices are associated using names or ids
- The probe function matches each device with its own driver through a `name`

# The Platform Model: `platform_device`

Kernel

  Intro

  Compiling

  Code

  Debugging

  Linked lists

  Scheduling

  Linux filesystem

  /dev filesystem

  proc filesystem

  sys filesystem

  Syscalls

  Real-Time

  Boot time

Kernel drivers

  Modules

  Memory

  Concurrency

  Char devices

  I/O

**Platforms**

  Using sysfs

  Timers

  Int. handlers

  Bottom halves

  Examples

In `include/linux/platform_device.h`:

```
struct platform_device {  
    const char      * name;  
    int             id;  
    struct device   dev;  
    u32             num_resources;  
    struct resource * resource;  
};
```

Note: the `name` field must be equal to the `name` field in the `platform_driver->driver` data structure in the driver

# The Platform Model: device

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

In `include/linux/device.h`:

```
struct device {  
    void      *driver_data;      /* data private to the driver */  
    void      *platform_data;    /* Platform specific data, device */  
    u64       *dma_mask;        /* DMA mask (if dma'able device)  
                                For instance, if the device can  
                                use DMA only on 24bit addresses,  
                                then dma_mask must be equal to  
                                0x00ffff. */  
    u64       coherent_dma_mask; /* Like dma_mask, but for a coherent  
                                area that can be accessed by the  
                                CPU and the device  
                                simultaneously. */  
    /* ... */  
};
```

# The Platform Model: resource

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms**
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

In `include/linux/ioport.h`:

```
struct resource {  
    resource_size_t start;  
    resource_size_t end;  
    const char *name;  
    unsigned long flags;  
    /* ... */  
};
```

- **flags** can be:

- `IORESOURCE_MEM`: for Memory I/O
- `IORESOURCE_IO`: for I/O ports
- `IORESOURCE_IRQ`: for interrupt lines

# Using platform devices: board-specific device

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
**Platforms**  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

If the device is specific of the board, all device information goes into `arch/arm/mach-at91/board-evi32.c`:

```
static struct <platform data struct> <data name> = {  
    .<struct field> = <init value>,  
};  
  
static struct resource <resources>[] = {  
    [0] = {  
        .start = ....,  
        .end = ....,  
        .flags = ....,  
    },  
    [1] = {  
        .start = ....,  
        .end = ....,  
        .flags = ....,  
    },  
};
```

# Using platform devices: board-specific device (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

Still in `arch/arm/mach-at91/board-evi32.c`:

```
static struct platform_device <platform device name> = {  
    .name          = "<driver name>",  
    .id            = 1,  
    .dev           = {  
        .platform_data = &<data name>,  
    },  
    .resource      = <my_resources>,  
    .num_resources = ARRAY_SIZE(<my_resources>),  
},  
};  
  
static void __init evi32_board_init(void)  
{  
    /* ... */  
    platform_device_register(&<platform device name>);  
}
```

Note: the `name` field must be equal to the `name` field in the `platform_driver->driver` data structure in the driver

# Using platform devices: micro-specific device

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

**Platforms**

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

If the device is specific of the architecture, instead, only platform data goes into `arch/arm/mach-at91/board-evi32.c`:

```
static struct <platform data struct> <data name> = {
    .<struct field> = <init value>,
};

static void __init evi32_board_init(void)
{
    /* ... */
    at91_add_device_<device>(&<data name>);
}
```

# Using platform devices: micro-specific device (2)

Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

**Platforms**

Using sysfs  
Timers

Int. handlers  
Bottom halves  
Examples

The rest goes into `arch/arm/mach-at91/<micro>.devices.c`:

```
static struct <platform data struct> <data name 2>;  
  
static struct resource <resources>[] = {  
    [0] = {  
        .start = ....,  
        .end = ....,  
        .flags = ....,  
    },  
    [1] = {  
        .start = ....,  
        .end = ....,  
        .flags = ....,  
    },  
};
```

# Using platform devices: micro-specific device (3)

Kernel  
  Intro  
  Compiling  
Code  
  Debugging  
Linked lists  
Scheduling  
Linux filesystem  
  /dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
  Modules  
  Memory  
Concurrency  
Char devices  
I/O  
**Platforms**  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

Still in `arch/arm/mach-at91/<micro>_devices.c`:

```
static struct platform_device <platform device name> = {  
    .name          = "<driver name>",  
    .id            = 1,  
    .dev           = {  
        .platform_data = &<data name 2>,  
    },  
    .resource      = <my_resources>,  
    .num_resources = ARRAY_SIZE(<my_resources>),  
},  
;  
  
void __init at91_add_device_<device> \  
    (struct <platform data struct> *<data name>)  
{  
    /* ... */  
    <data name 2> = *<data name>;  
    platform_device_register(&<platform device name>);  
}
```

# Using platform devices: the driver

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

Add to `drivers/.../<driver>.c`:

```
static struct platform_driver <platform driver name> = {  
    .probe          = <probe function>,  
    .remove         = <remove function>,  
    .driver         = {  
        .name      = "<driver name>",  
        .owner     = THIS_MODULE,  
    },  
};
```

- Note: the `name` field must be equal to the `name` field in the `platform_device`
- When the names match, `probe()` for of the driver is called with the right `platform_device` as argument

# Using platform devices: the driver (2)

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms**
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

Add to `drivers/.../<driver>.c`:

```
static int __devinit <probe function> \
    (struct platform_device *pdev)
{
    struct <platform data struct> *data = \
        pdev->dev.platform_data;
    /* ... */
}
```

```
static int __devexit <remove function> \
    (struct platform_device *pdev)
{
    struct <platform data struct> *data = \
        pdev->dev.platform_data;
    /* ... */
}
```

# The Flat Device Tree

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

**Platforms**

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Problem: exponentially increasing number of Linux-powered ARM devices
- Solution adopted: use "*Flattened Device Tree*" instead of standard platform devices (which consists of data structures written in plain C)
  - See <http://lwn.net/Articles/448502/>
- Mechanism derived from the device tree format used by IBM Open Firmware (OF)
- Already used for other architectures (like PPC or MicroBlaze),
- It separates most hardware description from the kernel sources
  - Duplicated code reduced
  - One kernel image can support multiple platforms

# The Flat Device Tree

Kernel  
Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

Timers  
Int. handlers  
Bottom halves  
Examples

- Problem: exponentially increasing number of Linux-powered ARM devices
- Solution adopted: use "*Flattened Device Tree*" instead of standard platform devices (which consists of data structures written in plain C)
  - See <http://lwn.net/Articles/448502/>
- Mechanism derived from the device tree format used by IBM Open Firmware (OF)
- Already used for other architectures (like PPC or MicroBlaze),
- It separates most hardware description from the kernel sources
  - Duplicated code reduced
  - One kernel image can support multiple platforms

# The Flat Device Tree: dts & dtb

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
**Platforms**  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Based on a separate textual file written in a specialized language.
  - Description of the hardware components of the board.
  - It can have different forms:
    - **dts** (Device Tree Source): human-readable text file, usually manually edited by the kernel developers;
    - **dtb** (Device Tree Blob): machine-readable binary file generated (i.e., compiled) from a dts file; this file is passed to the kernel at boot time.
- For the ARM architecture, dts files are located in the `arch/arm/boot/dts/` directory.
- Basic inheritance:
  - **.dts**: board files
  - **.dtsi**: SoC files; included by the board files

# The Flat Device Tree: dts & dtb

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling

Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem  
Syscalls

Real-Time  
Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Based on a separate textual file written in a specialized language.
  - Description of the hardware components of the board.
  - It can have different forms:
    - **dts** (Device Tree Source): human-readable text file, usually manually edited by the kernel developers;
    - **dtb** (Device Tree Blob): machine-readable binary file generated (i.e., compiled) from a dts file; this file is passed to the kernel at boot time.
- For the ARM architecture, dts files are located in the `arch/arm/boot/dts/` directory.
- Basic inheritance:
  - **.dts**: board files
  - **.dtsi**: SoC files; included by the board files

# The Flat Device Tree dtc

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
**Platforms**  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- To go from one form to another, a Device Tree Compiler (DTC) is needed
  - It can be found as `scripts/dtc/dtc` inside the Linux kernel
  - Built if the support for flattened device tree is enabled in the kernel config (symbol `CONFIG_USE_OF` where OF stands for “Open Firmware”).

- DTS → DTB:

```
./scripts/dtc/dtc -O dtb -b 0 -o myboard.dtb myboard.dts
-b <number>: set the physical cpu used for boot
-S <bytes>: add extra space to make the blob at least <bytes> long
```

- DTB → DTS:

```
./scripts/dtc/dtc -I dtb -O dts -b 0 -o myboard.dts myboard.dtb
```

# Using the Flat Device Tree

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms**
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

- U-Boot command:

```
bootm <uImage addr> - <dtb addr>
```

# The sys filesystem: internals

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Sysfs allocates a buffer of size `PAGE_SIZE` and passes it to the method
  - On i386, this is 4096
- Sysfs will call the method exactly once for each read or write
- Attribute method implementations should operate on an identical buffer when reading and writing values

# The sys filesystem: internals

Kernel

  Intro

  Compiling

  Code

  Debugging

  Linked lists

  Scheduling

  Linux filesystem

  /dev filesystem

  proc filesystem

  sys filesystem

  Syscalls

  Real-Time

  Boot time

Kernel drivers

  Modules

  Memory

  Concurrency

  Char devices

  I/O

  Platforms

  Using sysfs

  Timers

  Int. handlers

  Bottom halves

  Examples

- `show()` methods should return the number of bytes printed into the buffer
  - They should always use `snprintf()`
- `store()` methods should return the number of bytes used from the buffer
  - This can be done using `strlen()`
- `show()` and `store()` methods can always return errors
  - If a bad value comes through, be sure to return an error

# The sys filesystem: the code

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
**Using sysfs**  
Timers  
Int. handlers  
Bottom halves  
Examples

```
static DEVICE_ATTR(<name>, 0644, show_data, store_data);
```

The file name will be <name> with a mode of 0644  
(-rw-r-r-)

```
static ssize_t store_data (struct device *dev, \
                          struct device_attribute *attr, \
                          const char *buf, size_t len)
{
    struct platform_device *pdev = to_platform_device(dev);
    unsigned int value = simple_strtoul (buf, NULL, 10);
    /* ... */
    return strnlen(buf, PAGE_SIZE);
}

static ssize_t show_data (struct device *dev, \
                        struct device_attribute *attr, \
                        char *buf)
{
    struct platform_device *pdev = to_platform_device(dev);
    unsigned int value = ...;
    return sprintf(buf, PAGE_SIZE, "%u\n", value);
}
```

# The sys filesystem: the code (2)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
**Using sysfs**  
Timers  
Int. handlers  
Bottom halves  
Examples

```
static struct platform_driver <platform driver name> = {
    .probe          = <probe function>,
    .remove         = <remove function>,
    .driver         = {
        .name      = "<driver name>",
        .owner     = THIS_MODULE,
    },
};

static int __devinit <probe function> (struct platform_device *pdev)
{
    struct <platform data struct> *data = \
        pdev->dev.platform_data;
    /* ... */
    device_create_file(&(pdev->dev), &dev_attr_<name>);
}

static int __devexit <remove function> (struct platform_device *pdev)
{
    struct <platform data struct> *data = \
        pdev->dev.platform_data;
    /* ... */
    device_remove_file(&(pdev->dev), &dev_attr_<name>);
}
```

# Time management

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

## Timers

Int. handlers

Bottom halves

Examples

- Several kernel functions are time-driven
- Periodic functions:
  - Time of day and system uptime updating
  - Runqueue balancing on SMP
  - Timeslice checking

# System timer

- Hardware timer issuing an interrupt at a programmable frequency called **tick rate**
  - On x86 architectures the primary system timer is the *Programmable Interrupt Timer* (PIT)
- The interrupt handler is called timer interrupt
- On Linux the tick rate is defined by the static preprocessor define **HZ** (see `linux/param.h`)
- The value of HZ is architecture-dependent and can be chosen by `menuconfig`
- Some internal calculations assume  $12 \leq \text{HZ} \leq 1535$  (see `linux/timex.h`)
- Another option is to completely turn-off CPU timer interrupts when the system is idle
  - **Dynamic Tick** option of `menuconfig`
  - Available since Linux 2.6.21

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
**Timers**  
Int. handlers  
Bottom halves  
Examples

# System timer

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Hardware timer issuing an interrupt at a programmable frequency called **tick rate**
  - On x86 architectures the primary system timer is the *Programmable Interrupt Timer (PIT)*
- The interrupt handler is called timer interrupt
- On Linux the tick rate is defined by the static preprocessor define **HZ** (see `linux/param.h`)
- The value of HZ is architecture-dependent and can be chosen by `menuconfig`
- Some internal calculations assume  $12 \leq \text{HZ} \leq 1535$  (see `linux/timex.h`)
- Another option is to completely turn-off CPU timer interrupts when the system is idle
  - **Dynamic Tick** option of `menuconfig`
  - Available since Linux 2.6.21

# System timer

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Hardware timer issuing an interrupt at a programmable frequency called **tick rate**
  - On x86 architectures the primary system timer is the *Programmable Interrupt Timer (PIT)*
- The interrupt handler is called timer interrupt
- On Linux the tick rate is defined by the static preprocessor define **HZ** (see `linux/param.h`)
- The value of HZ is architecture-dependent and can be chosen by `menuconfig`
- Some internal calculations assume  $12 \leq \text{HZ} \leq 1535$  (see `linux/timex.h`)
- Another option is to completely turn-off CPU timer interrupts when the system is idle
  - **Dynamic Tick** option of `menuconfig`
  - Available since Linux 2.6.21

# Typical values of HZ

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
  /dev filesystem  
  proc filesystem  
  sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  **Timers**  
  Int. handlers  
  Bottom halves  
  Examples

| Architecture  | HZ value        |
|---------------|-----------------|
| alpha         | 1024            |
| <b>arm</b>    | <b>100</b>      |
| cris          | 100             |
| h8300         | 100             |
| <b>i386</b>   | <b>250</b>      |
| ia64          | 1024            |
| m68k          | 100             |
| m68k-nommu    | 50, 100 or 1000 |
| mips          | 100             |
| mips64        | 100 or 1000     |
| parisc        | 100 or 1000     |
| ppc           | 1000            |
| <b>ppc64</b>  | <b>1000</b>     |
| s390          | 100             |
| sh            | 100 or 1000     |
| sparc         | 100             |
| sparc64       | 1000            |
| um            | 100             |
| v850          | 24,100 or 122   |
| <b>x86-64</b> | <b>1000</b>     |

# Larger HZ values: pros and cons

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
**Timers**  
Int. handlers  
Bottom halves  
Examples

- Timer interrupt runs more frequently
- Benefits:
  - Higher resolution of timed events
  - Improved accuracy of timed events
    - Average error = 5msec with HZ=100
    - Average error = 0.5msec with HZ=1000
  - Improved precision of syscalls employing a timeout
    - Examples: `poll()` and `select()`.
  - Measurements (e.g. resource usage) have finer resolution
  - Process preemption occurs more accurately
- Drawbacks:
  - The processor spends more time executing the timer interrupt handler
  - Higher overhead
  - More frequent cache trashing

# Larger HZ values: pros and cons

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
**Timers**  
Int. handlers  
Bottom halves  
Examples

- Timer interrupt runs more frequently
- Benefits:
  - Higher resolution of timed events
  - Improved accuracy of timed events
    - Average error = 5msec with HZ=100
    - Average error = 0.5msec with HZ=1000
  - Improved precision of syscalls employing a timeout
    - Examples: `poll()` and `select()`.
  - Measurements (e.g. resource usage) have finer resolution
  - Process preemption occurs more accurately
- Drawbacks:
  - The processor spends more time executing the timer interrupt handler
  - Higher overhead
  - More frequent cache trashing

# Measure of time

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. Relative times

- Most important to kernel functions and device drivers
- Example: 5 seconds from now
- Kernel facilities: `jiffies`, clock cycles and `get_cycles()`

## 2. Absolute times

- Current time of day
- Called “`wall time`”
- Most important to user-space applications
- Kernel facilities: `xtime`, `mktimes()` and `do_gettimeofday()`
- Usually best left to user-space, where the C library offers better support
- Dealing with absolute times in kernel space is often sign of bad implementation

# Measure of time

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 1. Relative times

- Most important to kernel functions and device drivers
- Example: 5 seconds from now
- Kernel facilities: `jiffies`, clock cycles and `get_cycles()`

## 2. Absolute times

- Current time of day
- Called “**wall time**”
- Most important to user-space applications
- Kernel facilities: `xtime`, `mktime()` and `do_gettimeofday()`
- Usually best left to user-space, where the C library offers better support
- Dealing with absolute times in kernel space is often sign of bad implementation

# Jiffies

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Global variable `jiffies`
- Number of ticks occurred since the system booted
- Read-only
- Incremented at any timer interrupt
- Not updated when interrupts are disabled
- The system uptime is therefore `jiffies/HZ` seconds
- Declared in `linux/jiffies.h` as

```
extern unsigned long volatile jiffies;
```
- Declared as `volatile` to tell the compiler not to optimize memory reads
- `unsigned long` (32 bits) for backward compliance

# Jiffies (2)

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- For high values of `HZ`, `jiffies` wraps around very quickly
- Four macros to handle wraparounds:
  - `time_after(unknown, known)`
  - `time_before(unknown, known)`
  - `time_after_eq(unknown, known)`
  - `time_before_eq(unknown, known)`
- The macros convert the values to `signed long` and perform a subtraction
- The `unknown` parameter is typically `jiffies`
- See `linux/jiffies.h`

# Jiffies\_64

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

**Timers**

Int. handlers

Bottom halves

Examples

- Extended variable `jiffies_64`
- Read-only
- Declared in `linux/jiffies.h` as  
`extern u64 jiffies_64;`
- `jiffies` is the lower 32 bits of the full 64-bit `jiffies_64` variable
- The access is not atomic on 32-bit architectures
- Can be read through the function `get_jiffies_64()`

# High-resolution processor-specific timing

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Many architectures provide high-resolution counter registers
- Incremented once **at each clock cycle**
- Architecture-dependent: readable from user space, writable, 32 or 64 bits, etc.
- x86 processors (from Pentium) have TimeStamp Counter (TSC)
  - 64-bit register
  - Readable from both kernel and user spaces
  - See `asm/msr.h` ("*machine-specific registers*")
  - Three macros:

```
rdtsc(low32, high32);
rdtscl(low32);
rdtscll(var64);
```

# High-resolution architecture-independent timing

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

## Timers

Int. handlers

Bottom halves

Examples

- The kernel offers an architecture-independent function
- `cycles_t get_cycles(void);`
- Defined in `asm/timex.h`
- Defined for every platform
  - Returns 0 on platforms without cycle-counter register

# Absolute times: the `xtime` variable

Kernel

Intro  
Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The `xtime` variable
- Defined in `kernel/timer.c` as `struct timespec xtime;`
- Timespec data structure:

```
struct timespec
{
    time_t    tv_sec;           /* seconds */
    long      tv_nsec;          /* nanoseconds */
};
```

- Time elapsed since January 1st 1970 ("epoch")
- Jiffies granularity
- Not atomic access
- Read through

```
struct timespec current_kernel_time(void);
```

# Absolute times: do\_gettimeofday()

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Function `do_gettimeofday()`

- Exported by `linux/time.h`

- Prototype:

```
void do_gettimeofday(struct timeval *tv);
```

- Timeval data structure:

```
struct timeval {  
    time_t          tv_sec;        /* seconds */  
    suseconds_t     tv_usec;      /* microseconds */  
};
```

- Can have resolution near to microseconds

- Interpolation: see what fraction of the current jiffy has already elapsed
- m68k and Sun3 systems cannot offer more than jiffy resolution

# Absolute times: `mktime()`

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

**Timers**

Int. handlers

Bottom halves

Examples

- Function `mktime()`

- Turns a wall-clock time into a jiffies value

- Prototype:

```
unsigned long mktime (unsigned int year, \
                      unsigned int mon, \
                      unsigned int day, \
                      unsigned int hour, \
                      unsigned int min, \
                      unsigned int sec); \
```

- See `linux/time.h`

# Delaying Execution

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
**Timers**  
Int. handlers  
Bottom halves  
Examples

- The wrong way: busy waiting

```
while (time_before(jiffies, j1))
    cpu_relax();
```

- Works because `jiffies` is declared as `volatile`
- Crash if interrupts are disabled

# Delaying Execution (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

**Timers**

Int. handlers

Bottom halves

Examples

- Release the CPU

```
while (time_before(jiffies, j1))
    schedule();
```

- Still not optimal
- There is always at least one runnable process
- The idle task never runs
- Waste of energy

# Delaying Execution (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

**Timers**

Int. handlers

Bottom halves

Examples

- The best way to implement a delay is to ask the kernel to do it!
- Facilities:
  1. `ndelay()`, `udelay()`, `mdelay()`
  2. `schedule_timeout()`
  3. Kernel timers

# Small delays

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Sometimes the kernel code requires very short and rather precise delays
- Example: synchronization with hardware devices
- The kernel provides the following functions:
  - `void ndelay (unsigned long nsecs);`
  - `void udelay (unsigned long usecs);`
  - `void mdelay (unsigned long msecs);`
- Busy looping for a certain number of cycles
- Trivial usage:  
`udelay(150);` for 150  $\mu$ secs.
- The delay is **at least** the requested value
- See `linux/delay.h`

# Small delays (2)

Kernel

  Intro

  Compiling

  Code

  Debugging

  Linked lists

  Scheduling

  Linux filesystem

  /dev filesystem

  proc filesystem

  sys filesystem

  Syscalls

  Real-Time

  Boot time

Kernel drivers

  Modules

  Memory

  Concurrency

  Char devices

  I/O

  Platforms

  Using sysfs

**Timers**

  Int. handlers

  Bottom halves

  Examples

- To avoid overflows, there is a check for constant parameters
  - Unresolved symbol `__bad_udelay`
- **Do not use for big amounts of time!**
- Architecture-dependent (see `asm/delay.h`)
- BogoMIPS:
  - How many loops the processor can complete in a second
  - Stored in the `loops_per_jiffy` variable
  - See `proc/cpuinfo`

# Small delays without busy waiting

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs

## Timers

- Int. handlers
- Bottom halves
- Examples

- Another way of achieving msec delays
- The kernel provides the following functions:
  1. `void msleep (unsigned int msecs);`
    - Uninterruptible
  2. `unsigned long msleep_interruptible (unsigned int msecs);`
    - Interruptible
    - Normally returns 0
    - Returns the number of milliseconds remaining if the process is awakened earlier
  3. `void ssleep (unsigned int seconds);`
    - Uninterruptible
- See `linux/delay.h`

# Small delays without busy waiting

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Another way of achieving msec delays
- The kernel provides the following functions:
  1. `void msleep (unsigned int msecs);`
    - Uninterruptible
  2. `unsigned long msleep_interruptible (unsigned int msecs);`
    - Interruptible
    - Normally returns 0
    - Returns the number of milliseconds remaining if the process is awakened earlier
  3. `void ssleep (unsigned int seconds);`
    - Uninterruptible
- See `linux/delay.h`

# Small delays without busy waiting

Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists

Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem

sys filesystem  
Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Another way of achieving msec delays
- The kernel provides the following functions:

1. `void msleep (unsigned int msecs);`

- Uninterruptible

2. `unsigned long msleep_interruptible (unsigned int msecs);`

- Interruptible
- Normally returns 0
- Returns the number of milliseconds remaining if the process is awakened earlier

3. `void ssleep (unsigned int seconds);`

- Uninterruptible

- See `linux/delay.h`

# schedule\_timeout()

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

I/O  
Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

- Prototype:

```
signed long schedule_timeout(signed long delay);
```

- See [linux/sched.h](#)
- Returns 0 unless the function returns before the given delay has elapsed (e.g. signal)
- Usage:  

```
set_current_state(TASK_INTERRUPTIBLE);  
schedule_timeout (delay);
```
- Use **TASK\_UNINTERRUPTIBLE** for uninterruptible delays

# Kernel timers

## Kernel

Intro  
Compiling

Code  
Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
**Timers**

Int. handlers  
Bottom halves  
Examples

- Allow to schedule an action to happen later without blocking the current process until that time arrives
- Have `HZ` resolution
- Example: shut down the floppy drive motor
- Also called “**dynamic timers**” or just “timers”
- Asynchronous execution: run in **interrupt context**
- Potential source of race conditions ⇒ protect data from concurrent access
- On SMPs the timer function is executed by the same CPU that registered it to achieve better cache locality

# Kernel timers (2)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
**Timers**  
Int. handlers  
Bottom halves  
Examples

- Can be dynamically created and destroyed
- Not cyclic
- No limit on the number of timers
- See `linux/timer.h` and `kernel/timer.c`
- Represented by the `struct timer_list` structure

```
struct timer_list {  
    struct list_head entry;  
    unsigned long expires;  
    spinlock_t lock;  
    void (*function)(unsigned long);  
    unsigned long data;  
    struct tvec_t_base_s * base;  
};
```

# Kernel timers (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

**Timers**

Int. handlers

Bottom halves

Examples

- The `expires` field represents when the timer will fire (expressed in jiffies)
- When the timer fires, it runs the function `function` with `data` as argument.

# Using kernel timers

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

1. Define a timer:

```
struct timer_list my_timer;
```

2. Define a function:

```
void my_timer_function(unsigned long data);
```

3. Initialize the timer:

```
init_timer(&my_timer);
```

4. Set an expiration time:

```
my_timer.expires = jiffies + delay;
```

5. Set the argument of the function:

```
my_timer.data = 0; or
```

```
my_timer.data = (unsigned long) &param;
```

6. Set the handler function:

```
my_timer.function = my_function;
```

7. Activate the timer:

```
add_timer(&my_timer);
```

# Using kernel timers

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

1. Define a timer:

```
struct timer_list my_timer;
```

2. Define a function:

```
void my_timer_function(unsigned long data);
```

3. Initialize the timer:

```
init_timer(&my_timer);
```

4. Set an expiration time:

```
my_timer.expires = jiffies + delay;
```

5. Set the argument of the function:

```
my_timer.data = 0; or
```

```
my_timer.data = (unsigned long) &param;
```

6. Set the handler function:

```
my_timer.function = my_function;
```

7. Activate the timer:

```
add_timer(&my_timer);
```

# Using kernel timers

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

1. Define a timer:

```
struct timer_list my_timer;
```

2. Define a function:

```
void my_timer_function(unsigned long data);
```

3. Initialize the timer:

```
init_timer(&my_timer);
```

4. Set an expiration time:

```
my_timer.expires = jiffies + delay;
```

5. Set the argument of the function:

```
my_timer.data = 0; or
```

```
my_timer.data = (unsigned long) &param;
```

6. Set the handler function:

```
my_timer.function = my_function;
```

7. Activate the timer:

```
add_timer(&my_timer);
```

# Using kernel timers

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

1. Define a timer:

```
struct timer_list my_timer;
```

2. Define a function:

```
void my_timer_function(unsigned long data);
```

3. Initialize the timer:

```
init_timer(&my_timer);
```

4. Set an expiration time:

```
my_timer.expires = jiffies + delay;
```

5. Set the argument of the function:

```
my_timer.data = 0; or
```

```
my_timer.data = (unsigned long) &param;
```

6. Set the handler function:

```
my_timer.function = my_function;
```

7. Activate the timer:

```
add_timer(&my_timer);
```

# Using kernel timers

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

1. Define a timer:

```
struct timer_list my_timer;
```

2. Define a function:

```
void my_timer_function(unsigned long data);
```

3. Initialize the timer:

```
init_timer(&my_timer);
```

4. Set an expiration time:

```
my_timer.expires = jiffies + delay;
```

5. Set the argument of the function:

```
my_timer.data = 0; or
```

```
my_timer.data = (unsigned long) &param;
```

6. Set the handler function:

```
my_timer.function = my_function;
```

7. Activate the timer:

```
add_timer(&my_timer);
```

# Using kernel timers

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

1. Define a timer:

```
struct timer_list my_timer;
```

2. Define a function:

```
void my_timer_function(unsigned long data);
```

3. Initialize the timer:

```
init_timer(&my_timer);
```

4. Set an expiration time:

```
my_timer.expires = jiffies + delay;
```

5. Set the argument of the function:

```
my_timer.data = 0; or
```

```
my_timer.data = (unsigned long) &param;
```

6. Set the handler function:

```
my_timer.function = my_function;
```

7. Activate the timer:

```
add_timer(&my_timer);
```

# Using kernel timers

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

1. Define a timer:

```
struct timer_list my_timer;
```

2. Define a function:

```
void my_timer_function(unsigned long data);
```

3. Initialize the timer:

```
init_timer(&my_timer);
```

4. Set an expiration time:

```
my_timer.expires = jiffies + delay;
```

5. Set the argument of the function:

```
my_timer.data = 0; or
```

```
my_timer.data = (unsigned long) &param;
```

6. Set the handler function:

```
my_timer.function = my_function;
```

7. Activate the timer:

```
add_timer(&my_timer);
```

# Using kernel timers (2)

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem

Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency

Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

### 8. Modify the timer:

```
mod_timer (&my_timer, jiffies + new_delay);
```

### 9. Deactivate the timer:

```
del_timer (&my_timer);
```

### 10. Deactivate the timer avoiding race conditions on SMPs :

```
del_timer_sync (&my_timer);
```

### 11. Knowing timer's state:

```
timer_pending (&my_timer);
```

# Using kernel timers (2)

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

### 8. Modify the timer:

```
mod_timer (&my_timer, jiffies + new_delay);
```

### 9. Deactivate the timer:

```
del_timer (&my_timer);
```

### 10. Deactivate the timer avoiding race conditions on SMPs :

```
del_timer_sync (&my_timer);
```

### 11. Knowing timer's state:

```
timer_pending (&my_timer);
```

# Using kernel timers (2)

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

### 8. Modify the timer:

```
mod_timer (&my_timer, jiffies + new_delay);
```

### 9. Deactivate the timer:

```
del_timer (&my_timer);
```

### 10. Deactivate the timer avoiding race conditions on SMPs :

```
del_timer_sync (&my_timer);
```

### 11. Knowing timer's state:

```
timer_pending (&my_timer);
```

# Using kernel timers (2)

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

## Timers

Int. handlers  
Bottom halves  
Examples

### 8. Modify the timer:

```
mod_timer (&my_timer, jiffies + new_delay);
```

### 9. Deactivate the timer:

```
del_timer (&my_timer);
```

### 10. Deactivate the timer avoiding race conditions on SMPs :

```
del_timer_sync (&my_timer);
```

### 11. Knowing timer's state:

```
timer_pending (&my_timer);
```

# Polling vs Interrupts

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

**Int. handlers**

Bottom halves

Examples

- One of the kernel's goals is to manage the hardware connected to the machine
- However, the processor (i.e. the kernel) is typically magnitudes faster than the hardware it talks to
- Therefore, making the kernel wait for hardware response is not efficient

# Polling vs Interrupts (2)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
**Int. handlers**  
Bottom halves  
Examples

- Two solutions:

- Polling

- Periodically the kernel checks the hardware status
    - The kernel is free to handle other tasks between two checks
    - Pros: easy to implement
    - Cons: overhead (not efficient solution)

- Interrupts

- The hardware signals the kernel when attention is needed
    - Interrupts are electrical signals received by the processor from devices
    - Generated asynchronously with respect to the processor clock
    - The number of interrupts received can be seen by `/proc/interrupts`

# Polling vs Interrupts (2)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
**Int. handlers**  
Bottom halves  
Examples

- Two solutions:

- Polling

- Periodically the kernel checks the hardware status
    - The kernel is free to handle other tasks between two checks
    - Pros: easy to implement
    - Cons: overhead (not efficient solution)

- Interrupts

- The hardware signals the kernel when attention is needed
    - Interrupts are electrical signals received by the processor from devices
    - Generated asynchronously with respect to the processor clock
    - The number of interrupts received can be seen by **/proc/interrupts**

# Interrupt requests

## Kernel

Intro  
Compiling  
Code

Debugging  
Linked lists  
Scheduling

Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Unique values are associated to different devices
  - These values are often called Interrupt ReQuest (IRQ) lines
  - Typically they are given a numeric value
  - This way, the kernel knows which device issued the interrupt and can service each interrupt with a unique handler
  - Some interrupts (e.g. PCI bus) can be dynamically assigned at run-time
  
- The kernel function that responds to a specific interrupt is called *interrupt handler* or *interrupt service routine* (ISR)
  - It is part of device's driver
  - In Linux it is a normal C function
  - Executed in interrupt context

# Interrupt requests

## Kernel

Intro  
Compiling  
Code  
Debugging

Linked lists  
Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem  
sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Unique values are associated to different devices
  - These values are often called Interrupt ReQuest (IRQ) lines
  - Typically they are given a numeric value
  - This way, the kernel knows which device issued the interrupt and can service each interrupt with a unique handler
  - Some interrupts (e.g. PCI bus) can be dynamically assigned at run-time
  
- The kernel function that responds to a specific interrupt is called *interrupt handler* or *interrupt service routine* (ISR)
  - It is part of device's driver
  - In Linux it is a normal C function
  - Executed in interrupt context

# Top halves and bottom halves

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- They must run as quickly as possible
- However, often they have to perform a large amount of work
- Therefore, the processing of interrupts has been splitted in two parts (called *halves*):
  - The interrupt handler is the *top half*
    - It is run **immediately** upon receipt of the interrupt
    - It performs only time critical work (e.g. data to or from the hardware)
    - It runs in interrupt context
    - During execution some interrupts are disabled
  - Then, there is the bottom half
    - Work that can be delayed in the future
    - Depending on the type of Bottom Half, it runs in interrupt context or in process context
    - Executed with all interrupts enabled

# Reentrancy and stack

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Interrupt handlers in Linux need not to be reentrant
- When an interrupt handler is executing, the corresponding interrupt line is masked out on all processors
- The other interrupts are enabled
- From earliest 2.6 kernels, each interrupt handler has its own 4-8KB stack
  - In the 2.4 kernel, instead, the stack was greater (i.e. 8-16KB) but shared with the interrupted process

# Interrupt handler registration

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

```
int request_irq (unsigned int irq,  
                 irqreturn_t (*handler)(int, void*, struct pt_regs*),  
                 unsigned long irqflags,  
                 const char* devname,  
                 void* dev_id);
```

- **irq**: requested IRQ
  - Interrupt number to allocate
  - For some devices it is hardcoded
  - For other devices is probed or determined dynamically
- **handler**: pointer to the handler function
  - The function is called whenever the kernel receives the interrupt

# Interrupt handler registration (2)

- **irqflags**: bitmask of zero or more of the following flags:

- **IRQF\_DISABLED**

- Historically used for “fast” interrupt handlers (assumed to execute quickly)
    - Today: the handler run with all interrupts disabled on the current cpu (instead of just on the current line)
    - For latency reasons, it should only be used when needed

- **IRQF\_SAMPLE\_RANDOM**

- The interrupts will contribute to the kernel entropy pool (i.e. used for random numbers generator)
    - Only for interrupts at nondeterministic times!

- **IRQF\_SHARED**

- Interrupt line shared among multiple handlers
    - To have more than one handler per line
    - It must be specified by each handler on the shared line
    - Run with interrupts disabled only on the current irq line

- **IRQF\_TIMER**

- Only used for timer interrupts

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Interrupt handler registration (3)

- **devname**: registered ASCII name of the device
  - Used by `/proc/irq` and `/proc/interrupts` for communication with the user
  
- **dev\_id**:
  - In case of interrupt line not shared
    - It is a pointer to some handler data
    - Can be `NULL`
    - Common practice: pass driver's `device` structure
  - In case of shared interrupt line, it is used to distinguish the handler. An interrupt line can be shared among several devices
    - The interrupt handler need to know if the interrupt on the shared line was actually generated by its device
    - This pointer is passed to the handler on each invocation
    - Interrupt handlers on shared lines are invoked sequentially
    - Each handler checks if its device generated the interrupt using some device's status register

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
**Int. handlers**  
Bottom halves  
Examples

# Interrupt handler registration (3)

- **devname**: registered ASCII name of the device
  - Used by `/proc/irq` and `/proc/interrupts` for communication with the user
- **dev\_id**:
  - In case of interrupt line not shared
    - It is a pointer to some handler data
    - Can be `NULL`
    - Common practice: pass driver's **device** structure
  - In case of shared interrupt line, it is used to distinguish the handler. An interrupt line can be shared among several devices
    - The interrupt handler need to know if the interrupt on the shared line was actually generated by its device
    - This pointer is passed to the handler on each invocation
    - Interrupt handlers on shared lines are invoked sequentially
    - Each handler checks if its device generated the interrupt using some device's status register

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
`/dev` filesystem  
`proc` filesystem  
`sys` filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

# Interrupt handler registration (4)

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Return value:

- Success: zero
- Error: !zero

- A common error is **-EBUSY** when the given interrupt line is already in use

- request\_irq()** can sleep

- `proc_mkdir() → proc_create() → kmalloc()`
- `request_irq()` cannot be called from interrupt context

- Initialize hardware and register interrupt handlers in the proper order!

- To prevent the handler from running before the device is fully initialized

# Freeing interrupt handlers

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

**Int. handlers**

Bottom halves

Examples

- To unregister the handler use:

```
void free_irq(unsigned int irq, void * dev_id);
```

- For shared interrupt lines, the line is disabled only when the last handler is removed
- It can sleep
  - It cannot be called from interrupt context

# Interrupt handler prototype

```
static irqreturn_t intr_handler(int irq, void *dev_id,  
struct pt_regs * regs);
```

- **irq**: interrupt line
  - Historic reasons: not useful today
- **dev\_id**: same pointer given to `request_irq()`
- **regs**: pointer to a structure containing the processor registers and state before servicing the interrupt
  - Used only for debugging
- Return value:
  - **IRQ\_NONE**: the handler detected an interrupt for which its device was not the originator
    - Used to share interrupt channels or to detect spurious interrupts
  - **IRQ\_HANDLED**: the interrupt was correctly recognized and handled

# Disabling and re-enabling interrupts

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- The kernel provides two functions to explicitly disable/enable interrupts:
  - `local_irq_disable()`
  - `local_irq_enable()`
- They disable **all** interrupts **only** on the current processor
- There is no way of disabling interrupts on all processors
- Can be called from interrupt context

# Disabling and re-enabling interrupts (2)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers

Int. handlers  
Bottom halves  
Examples

- Since we cannot know the interrupts state prior to these calls, it is safer the use of

- `unsigned long flags`
- `local_irq_save(flags)`
- `local_irq_restore(flags)`

Note: they must be called from within the same function!

- Since at least one supported architecture (i.e. SPARC) incorporates stack information into flags...

- `flags` cannot be passed to another function
- `local_irq_save()` and `local_irq_restore()` must occur in the same function

# Disabling and re-enabling interrupts (2)

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
Bottom halves  
Examples

- Since we cannot know the interrupts state prior to these calls, it is safer the use of

- `unsigned long flags`
- `local_irq_save(flags)`
- `local_irq_restore(flags)`

Note: they must be called from within the same function!

- Since at least one supported architecture (i.e. SPARC) incorporates stack information into flags...

- `flags` cannot be passed to another function
- `local_irq_save()` and `local_irq_restore()` must occur in the same function

# Disabling a specific interrupt line

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves
- Examples

- It is possible to disable a specific line for the entire system (i.e. all CPUs)
- This is called **masking**
- The kernel provide three functions:
  - `void disable_irq(unsigned int irq);`
  - `void enable_irq(unsigned int irq);`
    - They do not return until any currently executing handler completes
  - `void disable_irq_nosync(unsigned int irq);`
    - It does not wait for current handlers to complete
- The call to these functions can be nested
  - If `disable_irq()` is called twice, only the second call to `enable_irq()` actually reenables the line

# Knowing Interrupt System Status

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `irqs_disabled()`

- It returns nonzero if the interrupts are disabled on the local processor

- `in_interrupt()`

- It returns zero if the kernel is in process context
- It returns nonzero if the kernel is in interrupt context

- `in_irq()`

- It returns nonzero if the kernel is executing an interrupt handler

# Bottom halves

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers

Int. handlers  
**Bottom halves**  
Examples

- “Bottom half” is a generic operating system term referring to the deferred portion of interrupt processing
- Called “bottom” because it refers to the second, or bottom, half of the interrupt processing solution
- In the Linux kernel there are three bottom half mechanisms:
  1. Softirqs
    - Available since 2.3
  2. Tasklets
    - Available since 2.3
    - Built on top of softirqs
  3. Work queues
    - Available since 2.5

# Bottom halves

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
**Bottom halves**  
Examples

- “Bottom half” is a generic operating system term referring to the deferred portion of interrupt processing
- Called “bottom” because it refers to the second, or bottom, half of the interrupt processing solution
- In the Linux kernel there are three bottom half mechanisms:
  1. Softirqs
    - Available since 2.3
  2. Tasklets
    - Available since 2.3
    - Built on top of softirqs
  3. Work queues
    - Available since 2.5

# Bottom halves

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem

Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers  
Int. handlers

## Bottom halves

Examples

- “Bottom half” is a generic operating system term referring to the deferred portion of interrupt processing
- Called “bottom” because it refers to the second, or bottom, half of the interrupt processing solution
- In the Linux kernel there are three bottom half mechanisms:
  1. Softirqs
    - Available since 2.3
  2. Tasklets
    - Available since 2.3
    - Built on top of softirqs
  3. Work queues
    - Available since 2.5

# Bottom halves

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs  
Timers  
Int. handlers  
**Bottom halves**  
Examples

- “Bottom half” is a generic operating system term referring to the deferred portion of interrupt processing
- Called “bottom” because it refers to the second, or bottom, half of the interrupt processing solution
- In the Linux kernel there are three bottom half mechanisms:
  1. Softirqs
    - Available since 2.3
  2. Tasklets
    - Available since 2.3
    - Built on top of softirqs
  3. Work queues
    - Available since 2.5

# Softirqs

- Rarely used
- Used for timing critical activities
  - Currently used by only two subsystems (i.e. networking and SCSI)
- Processed by the `ksoftirqd` kernel thread
- Run in interrupt context
- Kernel timers and tasklets are built on top of softirqs
- Code: `kernel/softirq.c`

- A softirq never preempts another softirq on the same processor
- The only event that can preempt a softirq is an interrupt handler
- However, if the same softirq is raised again while it is executing, another processor can run it simultaneously
  - Need proper locking or per-processor data
  - Reason why tasklets are preferred

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time

Boot time

Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O

Platforms  
Using sysfs

Timers

Int. handlers

Bottom halves  
Examples

# Softirqs (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

**Bottom halves**

Examples

```
struct softirq_action {
    void (*action)(struct softirq_action *);
    void* data; /* data to pass to function */
};
```

- Statically allocated at compile-time

- 32-entry array:

```
static struct softirq_action softirq_vec[32];
```

- Maximum of 32 registered softirqs
  - Cannot be dynamically registered and destroyed
  - Currently, only 6 entries are used

# Softirq handler

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

**Bottom halves**

Examples

- Prototype:

```
void softirq_handler(struct softirq_action *)
```

- This function is invoked with a pointer to the corresponding `softirq_action` structure
- Softirq handlers cannot sleep

# Softirq execution

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
**Bottom halves**  
Examples

- The interrupt handler marks the softirq for execution
- This is called **raising the softirq**
- Usually the softirq is marked by the handler just before returning
- Pending softirqs are checked and executed at suitable times:
  - In the return from the interrupt handler
  - In the `ksoftirqd` kernel thread
  - In any code that explicitly checks for and executes pending interrupts (e.g. networking subsystem)
- Softirqs execution is done by `do_softirq()`
  - It loops over all pending softirqs and invokes their handlers

# Softirqs priorities

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls

Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices

I/O  
Platforms  
Using sysfs  
Timers

Int. handlers

## Bottom halves

Examples

- Sofirqs have a relative priority which specifies the order of execution
- The priority is declared statically via an `enum` in `linux/interrupt.h`:

| Index                        | Priority | Description             |
|------------------------------|----------|-------------------------|
| <code>HI_SOFTIRQ</code>      | 0        | High-priority tasklets  |
| <code>TIMER_SOFTIRQ</code>   | 1        | Timer bottom half       |
| <code>NET_TX_SOFTIRQ</code>  | 2        | Send network packets    |
| <code>NET_RX_SOFTIRQ</code>  | 3        | Receive network packets |
| <code>SCSI_SOFTIRQ</code>    | 4        | SCSI bottom half        |
| <code>TASKLET_SOFTIRQ</code> | 5        | Tasklets                |

- Softirqs with lowest numerical priority execute before those with a higher numerical priority

# Softirqs priorities

Kernel

  Intro

  Compiling

  Code

  Debugging

  Linked lists

  Scheduling

  Linux filesystem

  /dev filesystem

  proc filesystem

  sys filesystem

  Syscalls

  Real-Time

  Boot time

Kernel drivers

  Modules

  Memory

  Concurrency

  Char devices

  I/O

  Platforms

  Using sysfs

  Timers

  Int. handlers

**Bottom halves**

  Examples

- Softirq registration:

```
void open_softirq(int index, void (*action)(struct  
softirq_action*), void *data);
```

- Raising a softirq:

```
raise_softirq(int index);
```

# Tasklets

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- Simple and easy-to-use softirqs
- Simpler interface and more relaxed locking rules than softirqs
- Despite the name, tasklets have nothing to do with tasks
- Run in interrupt context
- Good tradeoff between performance and ease to use
  - Tasklets are suitable for most bottom-half processing
- Unlike softirqs...
  - Two tasklets of the same type cannot run simultaneously
  - Tasklets can be dynamically created at run-time
  - Tasklets are said “scheduled” instead of “raised”
- Two kind of tasklets:
  - `HI_SOFTIRQ` and `TASKLET_SOFTIRQ`
  - The only difference is the priority

# Tasklet struct

Kernel  
Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

Kernel drivers  
Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
**Bottom halves**  
Examples

- Tasklets are represented by the following structure:

```
struct tasklet_struct {  
  
    /* next tasklet in the list: */  
    struct tasklet_struct * next;  
  
    /* state of the tasklet: */  
    unsigned long state;  
  
    /* reference counter: */  
    atomic_t count;  
  
    /*tasklet handler function: */  
    void (*func)(unsigned long);  
  
    /* argument to the tasklet function: */  
    unsigned long data;  
};
```

# Tasklet struct (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- **func**: tasklet handler
  - Equivalent of **action** to a softirq
  - Receives **data** as argument
- **state**: one of
  - 0
  - **TASKLET\_STATE\_SCHED**: tasklet that is scheduled to run
  - **TASKLET\_STATE\_RUN**: tasklet that is running
    - Used only on multiprocessors
- **count**: if nonzero, the tasklet is disabled

# Tasklet declaration

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

**Bottom halves**

Examples

- Static declaration (i.e. direct reference):

- `DECLARE_TASKLET(name, func, data)`

- Starts enabled (i.e. `count = 0`)

- `DECLARE_TASKLET_DISABLED(name, func, data)`

- Starts disabled (i.e. `count != 0`)

- They create a `tasklet_struct` with the given name

- Dynamic declaration (i.e. pointer):

- `tasklet_init(struct tasklet_struct* t, func, data);`

# Tasklet declaration

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

**Bottom halves**

Examples

- Static declaration (i.e. direct reference):

- `DECLARE_TASKLET(name, func, data)`

- Starts enabled (i.e. `count = 0`)

- `DECLARE_TASKLET_DISABLED(name, func, data)`

- Starts disabled (i.e. `count != 0`)

- They create a `tasklet_struct` with the given name

- Dynamic declaration (i.e. pointer):

- `tasklet_init(struct tasklet_struct* t, func, data);`

# Tasklet handler

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves**
- Examples

- Tasklet handler prototype:

```
void tasklet_handler(unsigned long data)
```

- Tasklet handlers cannot sleep
- Tasklet handlers run with all interrupts enabled

# Tasklet enabling/disabling

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- To enable a tasklet:

- `tasklet_enable(&my_tasklet);`
- It must be called when the tasklet has been created with `DECLARE_TASKLET_DISABLED()`

- To disable a tasklet:

- `tasklet_disable(&my_tasklet);`
  - If the tasklet is currently running, the function does not return until it finishes execution
- `tasklet_disable_sync(&my_tasklet);`
  - It does not wait for the tasklet to complete

# Tasklets local enabling/disabling

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
**Bottom halves**  
Examples

- To disable softirqs and tasklets on the local processor:
  - `void local_bh_disable();`
- To enable softirqs and tasklets on the local processor:
  - `void local_bh_enable();`
  - It also checks for any pending softirq/tasklet and executes it
- These calls do not work with Work Queues
- These calls can be nested
  - Only the final call to `void local_bh_enable()` actually re-enables them
- They use the per-task `preempt_count` counter
  - When the counter reaches zero, bottom-half processing is possible re-enables them

# Tasklet scheduling/canceling

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- `tasklet_schedule(&my tasklet);`

- It schedules a tasklet (i.e. the tasklet is marked as pending)
- Done in the Top Half (i.e. interrupt handler)
- If the tasklet is scheduled again before running, it will run only once
- If the tasklet is scheduled again when running, it will run again
- For performance (i.e. cache), a tasklet runs on the processor that scheduled

- `tasklet_kill(&my tasklet);`

- It removes the tasklet from the pending queue

# Work Queues

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
  - Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves**
- Examples

- Different form of bottom half
  - Built on top of kernel threads
  - They have more overhead than softirqs and tasklets because of context switching
- Simple interface for deferring work to kernel threads
  - Easiest to use bottom-half mechanism
  - These kernel threads are called **worker threads**

# Work Queues (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

**Bottom halves**

Examples

- They are the only bottom-half mechanisms that run in process context
  - They can sleep
- Work queues can defer the work to a default worker or create their own threads
  - The default worker is called `events/n` where `n` is the processor number
  - Processor-intense and performance-critical work might benefit from its own thread

# Data structures representing the work

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

**Bottom halves**

Examples

The work is represented by the `work_struct` structure defined in `linux/workqueue.h`:

```
struct work_struct {  
    unsigned long pending; /* is this work pending ? */  
    struct list_head entry; /* linked list of all work */  
    void (*func)(void*); /* handler function */  
    void *data; /* argument to handler */  
    void *wq_data; /* used internally */  
    struct timer_list timer; /* timer used by delayed work queues */  
}  
};
```

# Work Queue declaration

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves**
- Examples

- Static declaration (i.e. direct reference):

- `DECLARE_WORK(name, void (*func)(void*), void* data)`
- It creates a `work_struct` named `name` with handler function `func` and argument `data`

- Dynamic declaration (i.e. pointer):

- `INIT_WORK(struct work_struct * work, void (*func)(void*), void* data);`

# Work Queue declaration

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves**
- Examples

- Static declaration (i.e. direct reference):

- `DECLARE_WORK(name, void (*func)(void*), void* data)`
- It creates a `work_struct` named `name` with handler function `func` and argument `data`

- Dynamic declaration (i.e. pointer):

- `INIT_WORK(struct work_struct * work, void (*func)(void*), void* data);`

# Work Queue handler

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
**Bottom halves**  
Examples

- Work Queue handler prototype:

```
void work_handler(void * data)
```

- Work Queue handlers run with all interrupts enabled
- Work Queue handlers are executed by kernel threads
  - They run in process context
    - They can sleep
  - However, they cannot access user-space memory
    - There is no associated user-space memory map for kernel threads
    - (the kernel can access user memory only when running on behalf of a user-space process)

# Default worker: scheduling work

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

**Bottom halves**

Examples

To schedule the handler using the default *events* worker:

- `schedule_work(work_struct * work);`
  - The work is scheduled immediately
  - The work is run as soon as the worker thread on the current processor wakes up
  
- `schedule_delayed_work(work_struct * work, unsigned long delay);`
  - The work is run after at least `delay` ticks

# Default worker: flushing work

## Kernel

Intro  
Compiling  
Code  
Debugging  
Linked lists  
Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time

## Kernel drivers

Modules  
Memory  
Concurrency  
Char devices  
I/O  
Platforms  
Using sysfs  
Timers  
Int. handlers  
**Bottom halves**  
Examples

- Sometimes you need to ensure that a given work has been completed before continuing
  - Especially important for modules, which should call this function before unloading
  - Useful also to check that no work is pending, to prevent race conditions
- `void flush_scheduled_work(void);`
- The function waits and does not return until all entries in the queue are executed
  - This function sleeps, therefore can be called only from process context
- It does not cancel any work!

# Default worker: canceling work

Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
- /dev filesystem
- proc filesystem
- sys filesystem
- Syscalls
- Real-Time
- Boot time

Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves**
- Examples

- To cancel delayed work use

```
int cancel_delayed_work(struct work_struct * work);
```

# Creating new workqueues

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- If the default queue is insufficient for your needs you can create a new workqueue
- This creates one worker thread per processor
- Use it if you really needs the performance of a unique set of threads!
- `struct workqueue_struct *create_workqueue(const char* name);`
- The parameter `name` is used to name the kernel threads
- This function creates all worker threads (one for each processor) and initializes them

# Non-default worker: scheduling work

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

**Bottom halves**

Examples

To schedule the handler using your own worker:

- `int queue_work(struct workqueue_struct * wq, struct work_struct * work);`
- `int queue_delayed_work(struct workqueue_struct * wq, struct work_struct * work, unsigned long delay);`

# Non-default worker: canceling work

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
- Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
- I/O
- Platforms
- Using sysfs
- Timers
- Int. handlers
- Bottom halves**
- Examples

- To flush the work on a given queue:

```
flush_workqueue(struct workqueue_struct *wq);
```

# Example: led driver: the platform device

## Kernel

Intro  
Compiling

Code  
Debugging

Linked lists  
Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

Suppose you want to write a driver for a led mounted as GPIO on pin PA29...

1. Create `arch/arm/include/asm/gpio_led.h`:

```
#ifndef __ASM_ARCH_GPIO_LED_H
#define __ASM_ARCH_GPIO_LED_H "gpio_led.h"

/* Generic GPIO Led data */
#define AT91_GPIO_LED_ON      1
#define AT91_GPIO_LED_OFF     0

struct at91_gpio_led_platdata {
    unsigned int      gpio;
    unsigned int      initial_value;
    char             *name;
};

#endif /* __ASM_ARCH_GPIO_LED_H */
```

# Example: led driver: the platform device

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
Linux filesystem  
/dev filesystem  
proc filesystem  
sys filesystem  
Syscalls  
Real-Time  
Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
Examples

Suppose you want to write a driver for a led mounted as GPIO on pin PA29...

## 1. Create `arch/arm/include/asm/gpio_led.h`:

```
#ifndef __ASM_ARCH_GPIO_LED_H
#define __ASM_ARCH_GPIO_LED_H "gpio_led.h"

/* Generic GPIO Led data */
#define AT91_GPIO_LED_ON      1
#define AT91_GPIO_LED_OFF     0

struct at91_gpio_led_platdata {
    unsigned int      gpio;
    unsigned int      initial_value;
    char             *name;
};

#endif /* __ASM_ARCH_GPIO_LED_H */
```

# Example: led driver: the platform device (2)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

## 2. Add to `arch/arm/mach-at91/board-evi32.c`:

```
#include <linux/device.h>
#include <asm/gpio_led.h>

static struct resource at91_green_led_resource = {
    .start      = AT91_PIN_PA29,
    .end        = AT91_PIN_PA29,
    .flags       = IORESOURCE_MEM,
};

static struct at91_gpio_led_platdata green_led = {
    .gpio          = AT91_PIN_PA29,
    .initial_value = AT91_GPIO_LED_ON,
    .name          = "green_led",
};
```

# Example: led driver: the platform device (3)

Kernel

Intro  
Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

and

```
static struct platform_device at91_green_led = {
    /* Name of the driver to be used: */
    .name      = "at91-gpio-led",
    .id        = -1,
    .dev = {
        .platform_data = &green_led,
    },
    .resource      = &at91_green_led_resource,
    .num_resources = 1,
};

static void __init evi32_board_init(void)
{
    /* ... */

    /* Leds */
    platform_device_register(&at91_green_led);
}

MACHINE_START(EVI32, "Evidence evi32")
    /* ... */
    .init_machine    = evi32_board_init,
MACHINE_END
```

# Example: led driver: the driver

## 3. Create drivers/leds/led-evi32.c:

```
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/delay.h>
#include <linux/string.h>
#include <linux/ctype.h>
#include <linux/leds.h>
#include <asm/hardware.h>

#include <linux/kernel.h>
#include <asm/arch/gpio.h>
#include <linux/init.h>
#include <asm/gpio_led.h>

struct at91_gpio_led {
    struct led_classdev cdev;
    struct at91_gpio_led_platdata *pdata;
};

static inline struct at91_gpio_led *pdev_to_gpio
    (struct platform_device *dev)
{
    return platform_get_drvdata(dev);
}

static inline struct at91_gpio_led *to_gpio
    (struct led_classdev *led_cdev)
{
    return container_of(led_cdev, struct at91_gpio_led, cdev);
}
```

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

# Example: a led driver: the driver (2)

Kernel  
  Intro  
  Compiling  
  Code  
  Debugging  
  Linked lists  
  Scheduling  
  Linux filesystem  
  /dev filesystem  
  proc filesystem  
  sys filesystem  
  Syscalls  
  Real-Time  
  Boot time  
  
Kernel drivers  
  Modules  
  Memory  
  Concurrency  
  Char devices  
  I/O  
  Platforms  
  Using sysfs  
  Timers  
  Int. handlers  
  Bottom halves  
Examples

and

```
static void at91_gpio_led_set
    (struct led_classdev *led_cdev,
     enum led_brightness value)
{
    struct at91_gpio_led *led = to_gpio(led_cdev);
    struct at91_gpio_led_platdata *pd = led->pdata;
    if (value == AT91_GPIO_LED_OFF)
        at91_set_gpio_value(pd->gpio, 255);
    else if (value == AT91_GPIO_LED_ON)
        at91_set_gpio_value(pd->gpio, 0);
}
```

# Example: a led driver: the driver (3) and

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

```
static int __init at91_gpio_led_probe
    (struct platform_device *dev)
{
    struct at91_gpio_led_platdata *pdata =
        dev->dev.platform_data;
    struct at91_gpio_led *led;
    int ret;
    printk (KERN_INFO "Adding device\n");

    led = kzalloc(sizeof(struct at91_gpio_led), GFP_KERNEL);
    if (led == NULL) {
        printk (KERN_ERR "ERROR: No memory\n");
        return -ENOMEM;
    }

    printk (KERN_INFO "Setting data...\n");
    platform_set_drvdata(dev, led);

    led->cdev.brightness_set = at91_gpio_led_set;
    led->cdev.name = pdata->name;

    led->pdata = pdata;

    /* register our new led device */

    printk (KERN_INFO "Registering device...\n");
    ret = led_classdev_register(&dev->dev, &led->cdev);
    if (ret < 0) {
        printk (KERN_ERR "ERROR: Registration failed!\n");
        goto exit_err1;
    }
}
```

# Example: a led driver: the driver (3)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

Still in `at91_gpio_led_probe()`...

```
    printk (KERN_INFO "Enabling gpio...\n");
    at91_set_gpio_output (pdata->gpio, 1);

    printk (KERN_INFO "Setting initial value...\n");
    at91_gpio_led_set (&led->cdev, pdata->initial_value);

    return 0;

err1:
    kfree (led);
    return ret;
}
```

# Example: a led driver: the driver (4)

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

and

```
static int at91_gpio_led_remove
          (struct platform_device *dev)
{
    struct at91_gpio_led *led = pdev_to_gpio(dev);
    printk (KERN_INFO "Removing device\n");

    led_classdev_unregister (&led->cdev);
    kfree (led);

    return 0;
}
```

# Example: a led driver: the driver (5)

and finally

```
static struct platform_driver at91_gpio_led_driver = {
    .driver = {
        // Name of the driver
        .name     = "at91-gpio-led",
        .owner    = THIS_MODULE,
    },
    .probe     = at91_gpio_led_probe,
    .remove   = at91_gpio_led_remove,
};

static int __init at91_gpio_led_init(void)
{
    printk(KERN_INFO "Registering device\n");
    return platform_driver_register(&at91_gpio_led_driver);
}

static void __exit at91_gpio_led_exit(void)
{
    printk(KERN_INFO "Unregistering device\n");
    platform_driver_unregister(&at91_gpio_led_driver);
}

module_init(at91_gpio_led_init);
module_exit(at91_gpio_led_exit);

MODULE_AUTHOR("Claudio Scordino <claudio@evidence.eu.com>");
MODULE_DESCRIPTION("AT91 GPIO led driver");
MODULE_LICENSE("GPL");
```

Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

# A led driver: final changes

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
  - Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
  - I/O
  - Platforms
  - Using sysfs
  - Timers
- Int. handlers
- Bottom halves

## Examples

### 3. Add to `drivers/leds/Makefile`:

```
obj-$(CONFIG_LED_EVI32) += led-evi32.o
```

### 4. Add to `drivers/leds/Kconfig`:

```
config LED_EVI32
    tristate "LED Support for Evidence EVI32"
    depends on LEDS_CLASS && MACH_EVI32
    help
        This option enables support for the led on
        Evidence EVI32 board.
```

### 5. Inside `.config` add:

```
CONFIG_LED_EVI32=y or run make menuconfig
```

# A led driver: final changes

Kernel

Intro  
Compiling

Code  
Debugging

Linked lists

Scheduling  
Linux filesystem

/dev filesystem  
proc filesystem

sys filesystem  
Syscalls

Real-Time

Boot time

Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

3. Add to `drivers/leds/Makefile`:

```
obj-$(CONFIG_LED_EVI32) += led-evi32.o
```

4. Add to `drivers/leds/Kconfig`:

```
config LED_EVI32
```

```
    tristate "LED Support for Evidence EVI32"
```

```
    depends on LEDS_CLASS && MACH_EVI32
```

```
    help
```

```
        This option enables support for the led on  
        Evidence EVI32 board.
```

5. Inside `.config` add:

```
CONFIG_LED_EVI32=y or run make menuconfig
```

# A led driver: final changes

## Kernel

- Intro
- Compiling
- Code
- Debugging
- Linked lists
- Scheduling
- Linux filesystem
  - /dev filesystem
  - proc filesystem
  - sys filesystem
  - Syscalls
- Real-Time
- Boot time

## Kernel drivers

- Modules
- Memory
- Concurrency
- Char devices
  - I/O
  - Platforms
  - Using sysfs
  - Timers
  - Int. handlers
  - Bottom halves
- Examples

### 3. Add to `drivers/leds/Makefile`:

```
obj-$(CONFIG_LED_EVI32) += led-evi32.o
```

### 4. Add to `drivers/leds/Kconfig`:

```
config LED_EVI32
    tristate "LED Support for Evidence EVI32"
    depends on LEDS_CLASS && MACH_EVI32
    help
```

This option enables support for the led on  
Evidence EVI32 board.

### 5. Inside `.config` add:

```
CONFIG_LED_EVI32=y or run make menuconfig
```

# A led driver: usage

## Kernel

Intro

Compiling

Code

Debugging

Linked lists

Scheduling

Linux filesystem

/dev filesystem

proc filesystem

sys filesystem

Syscalls

Real-Time

Boot time

## Kernel drivers

Modules

Memory

Concurrency

Char devices

I/O

Platforms

Using sysfs

Timers

Int. handlers

Bottom halves

Examples

- To turn on the led just type:

```
echo 1 >  
/sys/devices/platform/evi32-led/leds:evi32:led/brightness
```

- To turn off the led just type:

```
echo 0 >  
/sys/devices/platform/evi32-led/leds:evi32:led/brightness
```

- You can also write a C program and use the typical POSIX functions for files (i.e. `open`, `read`, `write`)

- Attention: do not use integers: the sysfs accepts and returns strings!