# Linux Kernel Programming

## Table of Contents

# Introduction to the Linux Kernel

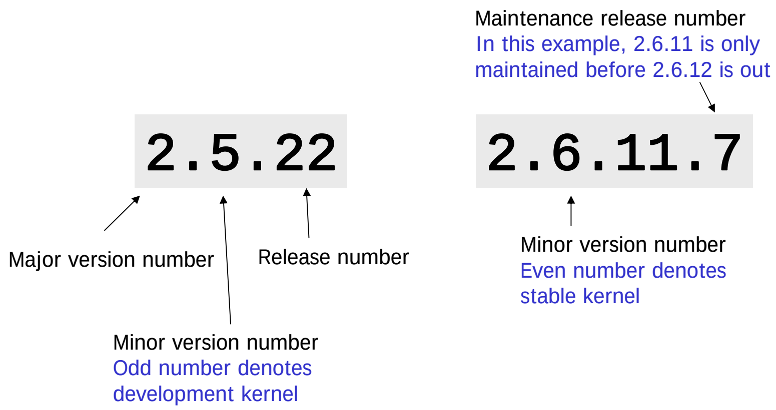
Hao-Ran Liu

# The history

- Initially developed by Linus Torvalds in 1991
- Source code is released under GNU Public License (GPL)
  - If you modify and release a program protected by GPL, you are obliged to release your source code

| Version | Features | Release Date |
|---|---|---|
| 0.01 | initial release, only on i386 | May 1991 |
| 1.0 | TCP/IP networking, swapping | March 1994 |
| 1.2 | more hardware support, DOSEMU | March 1995 |
| 2.0 | more arch. support, page cache, kernel thread | June 1996 |
| 2.2 | better firewalling, SMP performance, NTFS | January 1999 |
| 2.4 | iptable, ext3, ReiserFS, LVM | January 2001 |
| 2.6 | BIO, preemptive kernel, O(1) scheduler, I/O scheduler, objrmap, native POSIX thread library | December 2003 |

## Rules of Linux versioning




## Features of the Linux kernel

- Monolithic kernel
  - Do everything in a single large program in a single address space
    - Allow direct function invocation between components
  - Microkernel, on the other hand
    - Modular design, the kernel is broken down into separate processes
    - Use message passing interface instead of direction function call
    - Example: Mach, Windows NT/2000/XP

## Features of the Linux kernel (cont.)

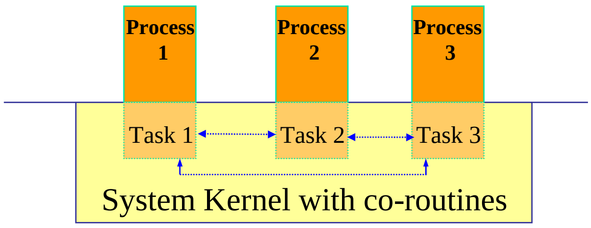
- Dynamic loading of kernel modules
  - Runtime binding of Linux kernel and modules
- Multiprocessor support
  - SMP, NUMA
- Preemptive kernel
  - Since 2.6, the kernel is capable of preempting a task even if it is running in the kernel
  - Dispatch latency of real-time tasks is greatly improved
- Threads are treated just like processes
  - The only difference is the sharing of memory resources
- Object-oriented device model, hotpluggable events, and a user-space device filesystem (sysfs)

## The concepts of processes

- Linux is a multi-user system, allowing multiple instances of programs to be executed at the same time
- Processes
  - An instance of a program in execution
  - Execution may be preempted at any time
  - Concurrency by means of context switching
  - Independency via the support of the CPU to prevent user programs from direct interacting with hardware components or accessing arbitrary memory locations
    - User mode and kernel mode (CPU ring level)
    - Memory protection (paging)

# Processes and tasks

- Processes
  - seen from outside: individual processes exist independently
- Tasks
  - seen from inside: only one operating system is running




# Process descriptor – `task_struct`

- Each process is represented by a process descriptor that includes information about the current state of the process

| Type | Name | Description |
|---|---|---|
| volatile long | state | Current state of the process |
| int | prio | Priority of the process |
| unsigned long | policy | Scheduling policy (FIFO, round robin, normal) |
| unsigned int | time_slice | Time quantum of the process, decreased at every timer interrupt. If zero, scheduler activates other process |
| struct list_head | tasks | double linked list of all process descriptors |
| pid_t | pid | the process ID of the process |
| struct thread_struct | thread | CPU-specific state (registers) of the process |

Accusys
The RAID Architects

# Context switching

- Context switching
  - Save the contents of several CPU registers into current process's process descriptor
  - Restore the contents of the CPU registers from next process's process descriptor
- Registers to be saved or restored
  - Program counter and stack pointer registers
  - General purpose registers
  - Floating point registers
  - Processor control registers (process status word)
  - Memory management registers (e.g. CR3 on x86)

Accusys
The RAID Architects

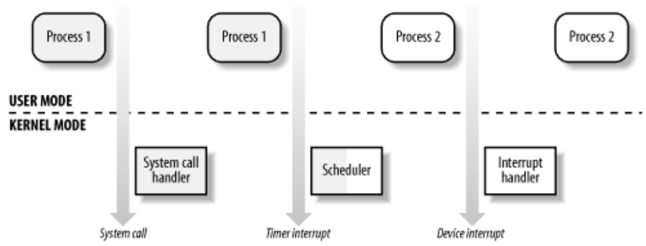
# User mode and kernel mode

- CPU runs in either user mode or kernel mode
- Programs run in user mode cannot access kernel space data structures or functions
- Programs in kernel mode can access anything
- CPU provides special instructions to switch between these modes

Accusys

# Switching into kernel mode

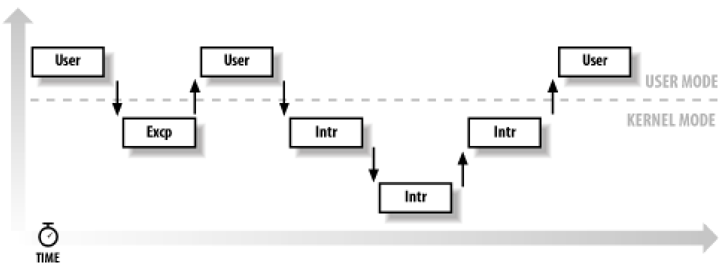
- CPU may enter kernel mode when:
  - A process invokes a system call
  - The CPU executing the process signals an exception
  - A peripheral device issues an interrupt signal to the CPU to notify it of an event




Accusys

# Reentrant kernel

- Reentrant -- several processes may be executing in kernel mode at the same time




Interleaving of kernel control paths

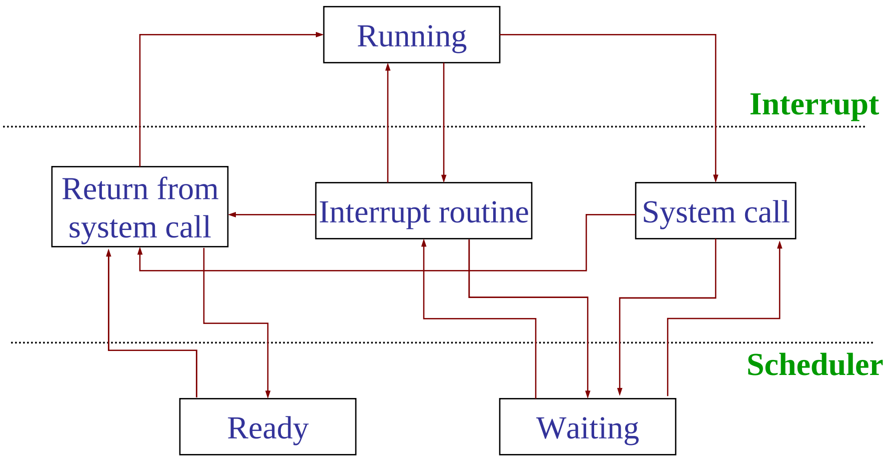
## Kernel control path

- Kernel control path – the sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt
- At any given moment, CPU may be doing one of the following things
  - In kernel space, in process context, executing on behalf of a specific process (system call or exception)
  - In kernel space, in interrupt context, not associated with a process, handling an interrupt
  - In user space, executing user code in a process

## Kernel mode stack

- In user mode, each process runs in its private address space
  - User-mode stack, data, code
- In kernel mode, each kernel control path refers to its own private kernel stack
  - A kernel mode stack per process
  - A interrupt stack for all interrupts

# Kernel control path of a process

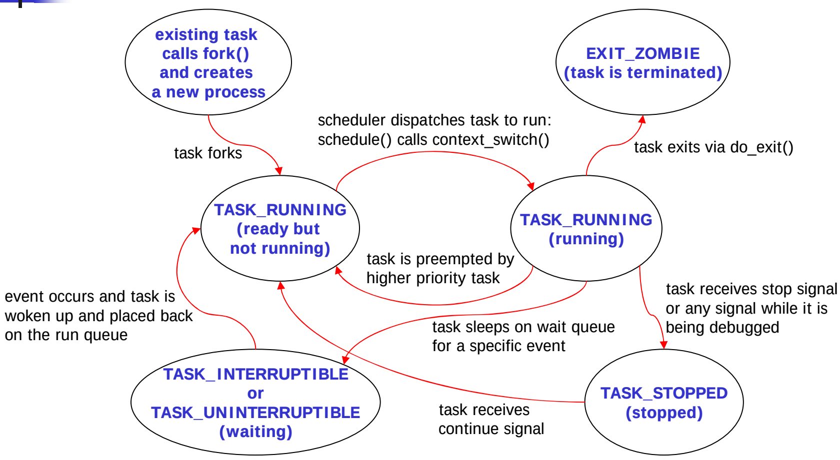



# Kernel control path of a process (cont.)

- Running
  - Task is active and running in the non-privileged user mode.
  - If an interrupt or system call occurs, the processor is switched to the privileged system mode and the appropriate interrupt routine is activated
- Interrupt routine
  - hardware signals an exception condition
  - E.g. page fault, keyboard input or clock generator signal every 1 ms
- System call
  - System calls are initiated by software interrupts
- Waiting
  - The process is waiting for an external event (e.g. I/O complete)
- Return from system call
  - When system call or interrupt is complete
  - Check if a context switch is needed and if there are signals to be processed
- Ready
  - The process is competing for the processor

Accusys

# Transition of process states




Accusys

# Interrupts

- Interrupts allows for hardware to communicate with operating system asynchronously
  - Remove the need of polling from OS
- Type of interrupts
  - Hardware generated interrupts (IRQ)
    - It is asynchronous! (the exact time of the delivery of an interrupt is unpredictable)
    - Example: interrupt from timer or network card
  - Software generated interrupts (exception or trap)
    - It is synchronous! (generated by CPU)
    - Example: Page fault, divide by zero, system call

## Designing interrupt handlers

- Limitations that must be aware of
  - Interrupt handlers may interrupt other important tasks (e.g. multimedia player) or other interrupt handlers
  - Runs with current interrupt level disabled or worst, all local interrupts are disabled
    - Delaying the interrupt processing of other devices (think about sharing interrupt lines)
  - Time critical since they deal with hardware (e.g. NIC)
  - Cannot block since they do not run in process context
- Design goal
  - Interrupt handlers should execute as quickly as possible

## Top halves and buttom halves

- Interrupt handler may need to perform a large amount of work
  - conflict with the goal of quickness
- Divide an interrupt handler into two parts
  - Top half
    - Run immediately upon receipt of the interrupt
    - Perform only the work that is time critical
  - Bottom half
    - Runs in the future at a convenient time with all interrupts enabled

## Timers and time management

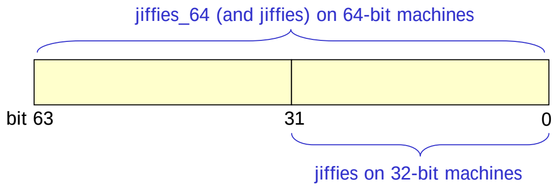
- System timer (i.e. timer interrupt)
  - Program the hardware timer to issue interrupts periodically
  - Works must be performed periodically
    - Update the system uptime and the time of day
    - Check if the current process has exhausted its timeslice and, if so, causing a reschedule
    - Run any dynamic timers that have expired
    - Update resource usage and processor time statistics
- Dynamic timer
  - schedule events that run once after a specified time has elapsed (ex. Flush an I/O request queue after some time)

## The tick rate: HZ

- HZ macro defines the frequency of the timer interrupt in Linux
  - If HZ = 100, you have 100 timer interrupts per second
  - On i386, HZ is 100 for 2.4 kernel and 250 for 2.6 kernel
- The pros and cons for a higher HZ
  - Pros: improve the accuracy of timed events and preemption of process
  - Cons: less processor time available for real work, less battery time for laptop

## `jiffies` variable

- The number of ticks that have occurred since the system booted
- `jiffies` variable is 32 bits or 64 bits in size depends on the architecture
- With `HZ` = 1000, it overflows in 49.7 days
  - Use macro provided by the kernel to compare tick counts correctly




## `xtime` variable

- The current time of day (the wall time)
  - the number of seconds that have elapsed since midnight of Jan. 1, 1970
- On boot, the kernel reads the RTC (real-time clock) and uses it to initialize `xtime`

```
struct timespec {
       time_t tv_sec;       /* seconds */
       long tv_nsec;        /* nanoseconds */
} xtime;
```

## The purposes of system calls

- The only interfaces through which user-space applications can access hardware resources
- The benefits
  - An abstracted hardware interface for user-space
    - Nearly all kinds of devices are treated as files
  - Enhancement of system security and stability
    - Properly use of CPU time, memory
  - Virtualization of hardware resources
    - Multitasking and virtual memory

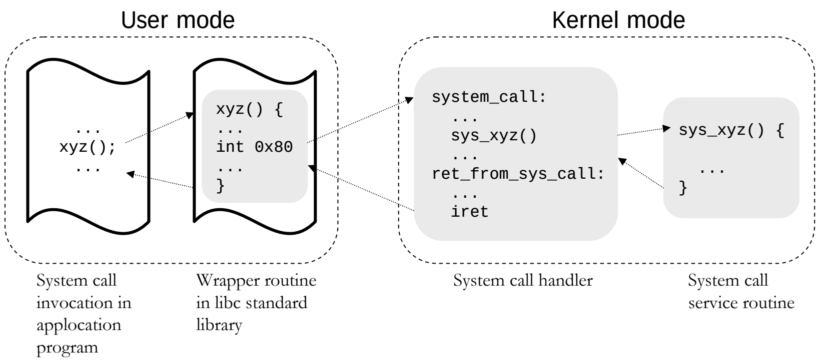
## POSIX, C library and system calls

- POSIX (Portable Operating System Interface)
  - A single set of APIs to be supported by every UNIX system to increase portability of source codes
- C library implements the majority of UNIX APIs
- A C library function can be
  - just a wrapper routine of a system call
  - implemented through several system calls
  - not related to any system calls

# *syscalls* in Linux

- Each system call is assigned a ***syscall number***, which is a unique number used to refer to a specific system call
- Kernel keeps a list of all registered system calls in the `sys_call_table`
- A special CPU instructions is used to switch into kernel mode and execute the system call in kernel-space
  - On i386, the special instructions can be `int 0x80` or `sysenter`

# Invoking a system call



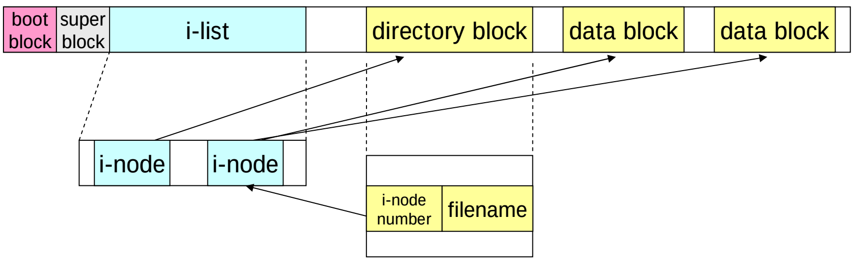
# Consideration of implementing a system call

- You need a syscall number, officially assigned to you during a developmental kernel series
- When assigned, the number and the system call interface cannot change
  - or else compiled applications will break
  - likewise, if a system call is removed, its system call number cannot be recycled
- The alternatives
  - Implement a device node and use `read()`, `write()` or `ioctl()`
  - Add the information as a file in procfs or sysfs

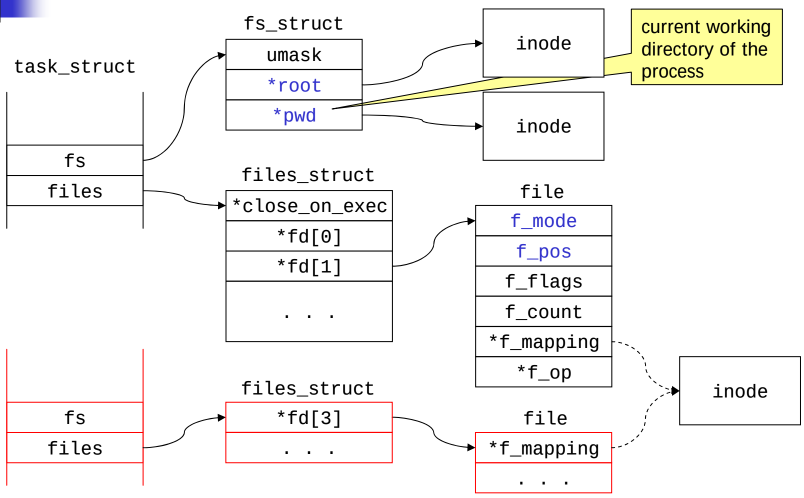
# Files and inodes

- Inode has a number of meanings
  - The inode structure in the kernel memory
  - The inode structure stored on the hard disk
  - Both describe files from their own viewpoint
- File structures is the view of a process on files represented by inodes
  - File is opened for: read, write or read+write
  - Current I/O position

## The structure of a traditional UNIX file system




## Files and inodes (cont.) -- two processes open the same file



## Linux kernel programming -- a different world

- No access to the C library
- The kernel code uses a lot of ISO C99 and GNU C extensions
  - Inline assembly
  - Inline functions
  - Branch optimization with macros: `likely()` and `unlikely()`
- No memory protection
- No (easy) use of floating point
- Small, fixed size stack
- Kernel is susceptible to race conditions because of
  - Multi-tasking support, Multiprocessing support, Interrupts and preemptive kernel

## Kernel books

- Linux Kernel Development 2nd Edition, Robert Love, Novell Press, 2005
- Understanding the Linux Kernel 3rd Edition, Bovet & Cesati, O'REILLY, 2005
- Linux Device Drivers 3rd Edition, Corbet, Rubini & Kroah-Hartman, O'REILLY, 2005

# Useful sites about Linux kernel

- Linux Weekly News, http://lwn.net
  - A great news site with an excellent commentary on the week's kernel happenings
- KernelTrap, http://www.kerneltrap.org
  - This site has many kernel-related development news, especially about the Linux kernel
- Kernel.org, http://www.kernel.org
  - The official repository of the kernel source
- Linux Kernel Mailing List, http://vger.kernel.org
  - The main forum for Linux kernel hackers

# Introduction to Linux start-up

Hao-Ran Liu

# Boot process overview

Hardware startup → BIOS → LILO or bootsect.s → Linux kernel → Mount root file system → /sbin/init

LILO is a versatile boot loader,
but it is functional-equivalent to bootsect.s in Linux (version 2.4 or before)

# Boot process

- BIOS
  - reads the first sector of the boot disk (floppy, hard disk, …, according to the BIOS parameter setting)
  - the boot sector (512 bytes) will contain program code for loading the operating system kernel (e.g., Linux Loader, LILO)
  - boot sector ends with 0xAA55
- Boot disk
  - Floppy: the first sector
  - Hard disk: the first sector is the master boot record (MBR)

# Boot sector and MBR

Boot Sector (Floppy)

| | |
|---|---|
| 0x000 | JMP xx |
| 0x003 | Disk parameters |
| 0x03E | Program code loading the OS kernel |
| 0x1FE | 0xAA55 |

MBR and extended partition table (Hard disk)

| | | |
|---|---|---|
| 0x000 | 0x1BE | Code for loading the boot sector of the active partition |
| 0x1BE | 0x010 | Partition 1 |
| 0x1CE | 0x010 | Partition 2 |
| 0x1DE | 0x010 | Partition 3 |
| 0x1EE | 0x010 | Partition 4 |
| 0x1FE | 0x002 | 0xAA55 |

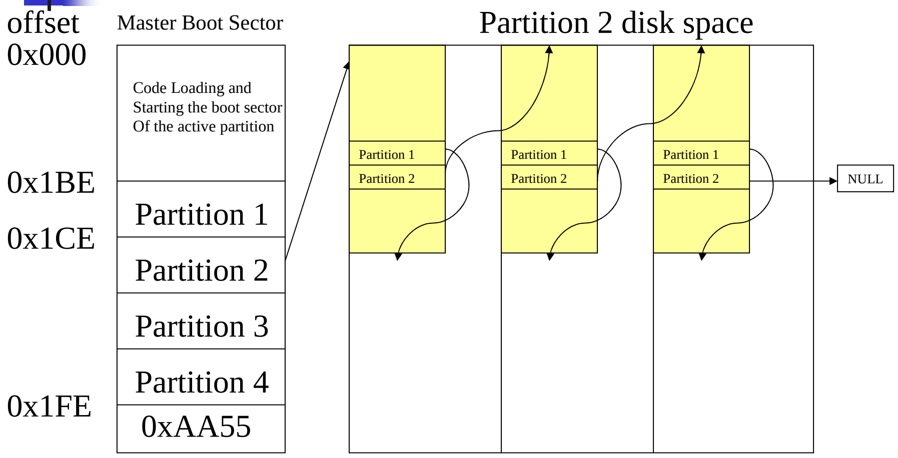
## MBR (*Master Boot Record*)

- Four primary partitions
  - only 4 partition entries
  - Each entry is 16 bytes
- Extended partition
  - If more than 4 partitions are needed
  - The first sector of extended partition is same as MBR
  - The first partition entry is for the first logical drive
  - The second partition entry points to the next logical drive (MBR)
- The first sector of each primary or extended partition contains a boot sector

## Structure of a Partition Entry

| | | | |
|---|---|---|---|
| 1 | Boot | | Boot flag: 0=not active, 0x80 active |
| 1 | HD | | Begin: head number |
| 2 | SEC | CYL | Begin: sector and cylinder number of boot sector |
| 1 | SYS | | System code: 0x83 Linux, 0x82: swap |
| 1 | HD | | End: head number |
| 2 | SEC | CYL | End: sector and cylinder number of last sector |
| 4 | low byte | high byte | Relative sector number of start sector |
| 4 | low byte | high byte | Number of sectors in the partition |

# Extended partition table



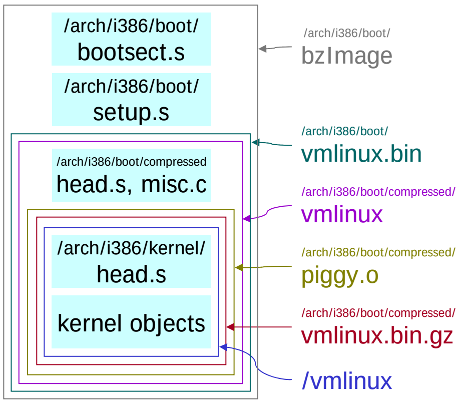

The structure of MBR and Extended Partition are the same.

# Active Partition

- Booting is carried out from the active partition which is determined by the boot flag
- Operations of MBR
  - determine active partition
  - load the boot sector of the active partition
  - jump into the boot sector at offset 0

Accusys

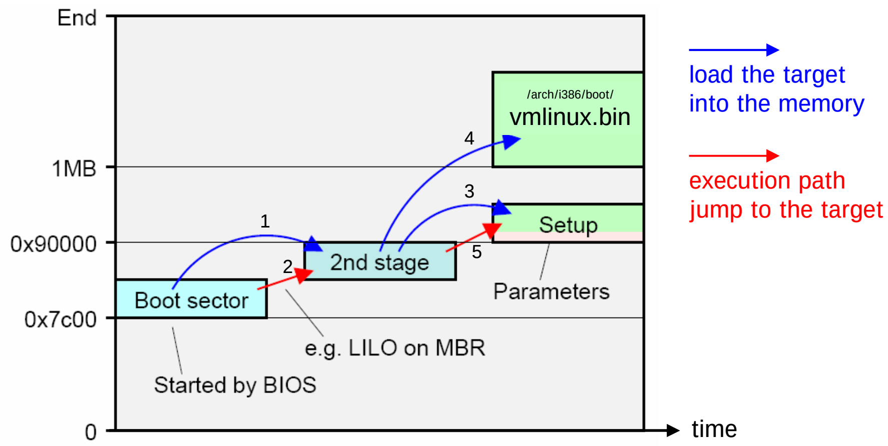
# Linux 2.6 kernel image



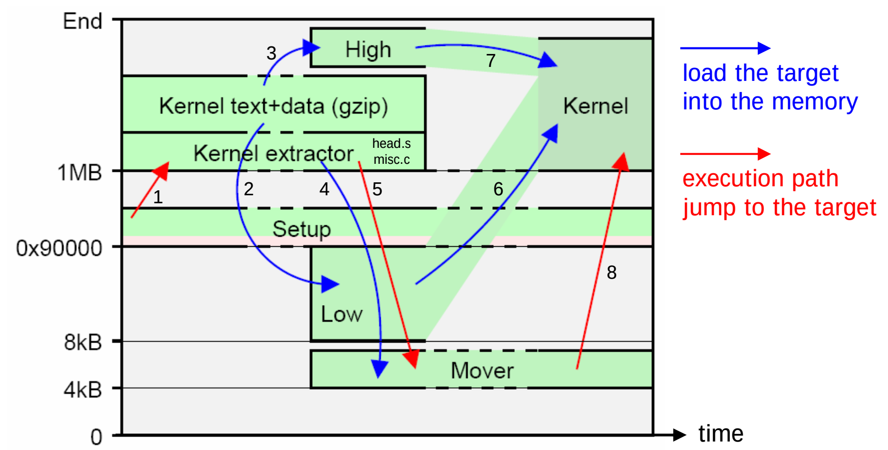

- bootsect.s: Linux floppy boot loader (only in version <= 2.4)
- setup.s: hardware initialization, switch to protected mode
- First head.s, misc.c: decompress kernel
- Second head.s: enable paging, setup GDT, jump to C routine start_kernel()

Accusys

# Loading a bzImage



## Starting a bzImage




## zImage and bzImage

- Both zImage and bzImage are compressed with gzip
- The only difference
  - zImage is loaded low and has a size limit of 512KB

| | zImage | bzImage |
|---|---|---|
| Boot loader (bootsect.s) places /arch/i386/boot/ vmlinux.bin at | between 0x10000~0x90000 | above 0x100000 |
| Decompressed kernel ( /arch/i386/boot/compressed/ vmlinux.bin )'s final address | 0x100000 | 0x100000 |

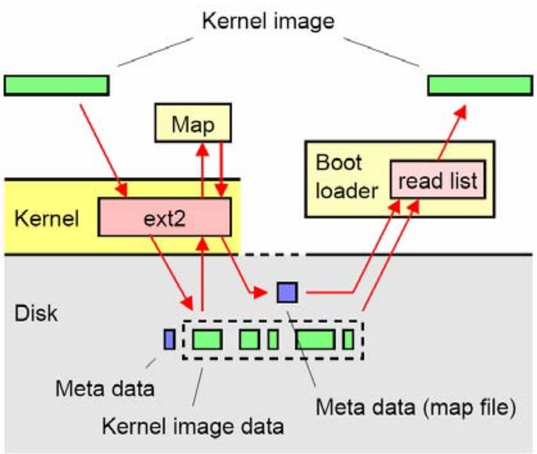
# The Linux/i386 boot protocol

```
        |                        |
0A0000  +------------------------+
        |  Reserved for BIOS     |   Do not use.  Reserved for BIOS EBDA.
09A000  +------------------------+
        |  Stack/heap/cmdline    |   For use by the kernel real-mode code.
098000  +------------------------+
        |  Kernel setup          |   The kernel real-mode code.
090200  +------------------------+
        |  Kernel boot sector    |   The kernel legacy boot sector.
090000  +------------------------+
        |  Protected-mode kernel |   The bulk of the kernel image.
010000  +------------------------+
        |  Boot loader           |   <- Boot sector entry point 0000:7C00
001000  +------------------------+
        |  Reserved for MBR/BIOS |
000800  +------------------------+
        |  Typically used by MBR |
000600  +------------------------+
        |  BIOS use only         |
000000  +------------------------+
```

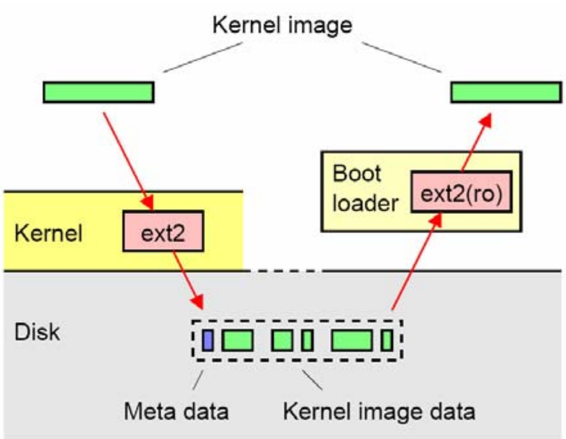
# LILO vs. GRUB

- LILO
  - Boot any file system
  - Need regeneration of a map file if kernel changes
  - Friendly to file system developers
- GRUB
  - Boot only known file system
  - No map file!
  - Friendly to normal users

## LILO – file system unaware




## GRUB – file system aware



# Starting the kernel

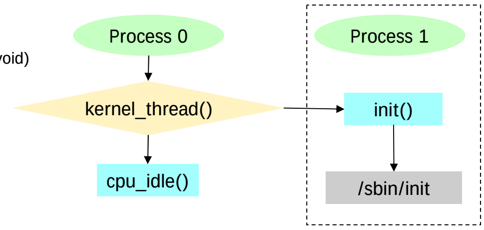
- After **head.s** and **misc.c** decompress kernel, it calls the first C routine **start_kernel()**
- Many hardware-independent parts of the kernel are initialized here.

```
asmlinkage void __init start_kernel(void)
{
  char * command_line;
  printk(linux_banner);
  setup_arch(&command_line);
  parse_options(command_line);
  trap_init();
  init_IRQ();
  sched_init();
  time_init();
  softirq_init();
  console_init();
...
```

# Spawn first process - init

- The original process now running is process 0
  - It generates a kernel thread (process 1) executing **init()** function
  - Subsequently, it is only concerned with using up unused computing time - **cpu_idle()**

```
asmlinkage void __init start_kernel(void)
{
...
  kernel_thread(init, NULL, ...)
  cpu_idle();
}
```



Accusys

# init() function

- Carry out the remaining initialization and open file descriptors 0, 1, 2 for the first user program being exec'ed later
- Try to execute a boot program specified by the user or one of the programs /sbin/init, /etc/init, or /bin/init
- If none of these programs exists, try to start a shell so that the superuser can repair the system. If this is not possible too, the system is stopped

```
static int init() {
{
 do_basic_setup();
 …
 if (open "/dev/console", O_RDWR, 0) < 0)
   printk("Warning: unable to open an initial console.\n");
 (void) dup(0);
 (void) dup(0);

 if (execute_command)
   execve(execute_command, argv_init, envp_init);
 execve("/sbin/init", argv_init, envp_init);
 execve("/etc/init", argv_init, envp_init);
 execve("/bin/init", argv_init, envp_init);
 execve("/bin/sh", argv_init, envp_init);
 panic("No init found...");
}
```

Accusys

# /sbin/init – parent of all processes

- Init configures the system and create processes according to /etc/inittab
- A Runlevel is:
  - A software configuration of what services to be started
  - A state that the system can be in
- /etc/inittab defines several runlevels

Accusys
The RAID Architects

# Description of runlevels

| Runlevel | Description |
|---|---|
| 0 | Halt - used to halt the system |
| 1 | Single-user text mode |
| 2 | Not used |
| 3 | Full multi-user text mode |
| 4 | Not used |
| 5 | Full multi-user graphic mode (with an X-based login screen) |
| 6 | Reboot – used to reboot the system |
| S or s | Used internally by scripts that run in runlevel 1 |
| a,b,c | On-demand run levels - typically not used. |

Accusys
The RAID Architects

# inittab syntax

- An entry in the inittab has this format:
  - **id : runlevels : action : process**
  - **id:** an unique id to identify the entry
  - **runlevels:** a list of runlevels for which the specified action should be taken
  - **action:** describes which action should be taken
  - **process:** specifies the program to be executed

Please refer to man page inittab(5) for details of the action field

## inittab example

```
# default runlevel is 3
id:3:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

# the start script of each runlevel
l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

# Run xdm in runlevel 5
x:5:respawn:/etc/X11/prefdm -nodaemon
```

## References


- Werner Almesberger, Booting Linux: the history and the future, Ottawa Linux Conference, 2000

# Using GNU Compiler and Binutils by Example

Hao-Ran Liu

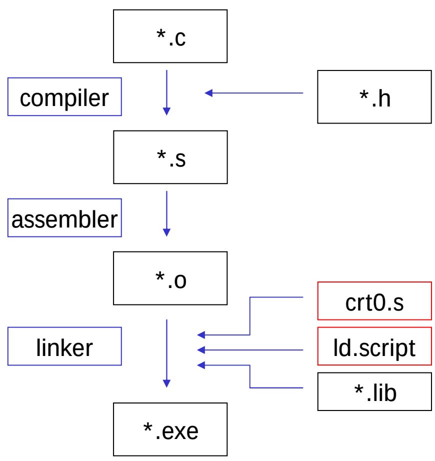
## Goals of this tutorial

- To familiar with the building process of the image of Linux kernel
- To learn the know-how of building an embedded platform

# GNU toolchains

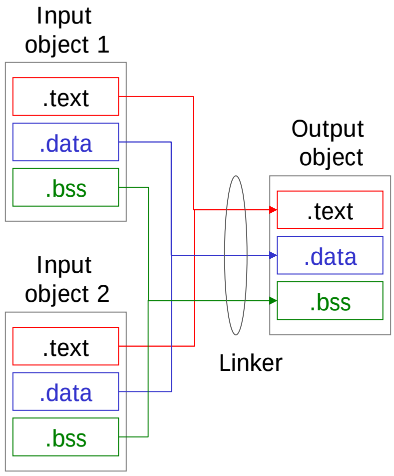
- GNU toolchain includes:
  - GNU Compiler Collection (GCC)
  - GNU Debugger (GDB)
  - GNU C Library (GLIBC)
    - Newlib for embedded systems
  - GNU Binary Utilities (BINUTILS)
    - Includes LD, AS, OBJCOPY, OBJDUMP, GPROF, STRIP, READELF, NM, SIZE…

# Compiling procedure



## Linker overview

- Linker combines input objects into a single output object
- Each input object has two table and a list of sections.
- Linker use the two table to:
  - Symbol table: resolved the address of undefined symbol in a object
  - Relocation table:
    - Translate 'relative addresses' to 'absolute address'

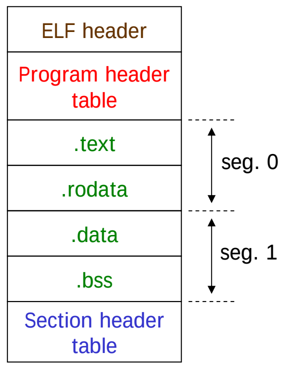



## The roles of crt0.s and ld.script in embedded development environment

- crt0.s
  - The real entry point: _start()
  - Initialize .bss sections
  - Initialize stack pointer
  - Call main()
- Linker script
  - Control memory layout of a output object
  - How input objects are mapped into a output object
  - Defaule linker script: run ld --verbose

# ELF format

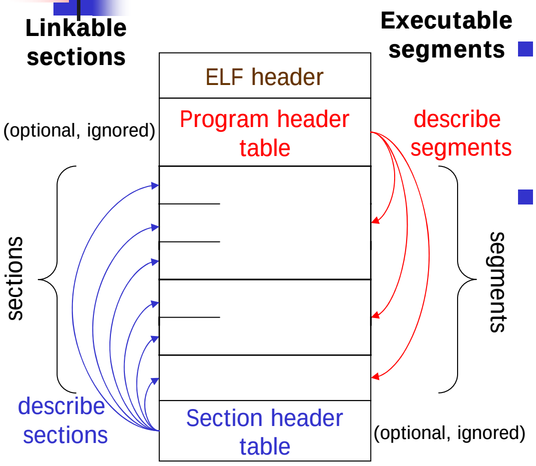
- What we load is partially defined by ELF
- **E**xecutable and **L**inkable **F**ormat
- Four major parts in an ELF file
  - ELF header – roadmap
  - Program headers describe segments directly related to program loading
  - Section headers describe contents of the file
  - The data itself




# 3 types of ELF

- relocatable(*.o) for linker
- Executable(*.exe) for loader
- shared object for both(*.so) (dynamic linking)

# Two views of an ELF file




- Program header is for ELF loader in Linux kernel
- Section header is for linker

| | Program header | Section header |
|---|---|---|
| relocatable | | V |
| executable | V | V |
| Shared object | V | V |

# ELF header

```
typedef struct {
    char magic[4] = "\177ELF"; // magic number
    char class;      // address size, 1 = 32 bit, 2 = 64 bit
    char byteorder; // 1 = little-endian, 2 = big-endian
    char hversion;  // header version, always 1
    char pad[9];
    short filetype; // file type: 1 = relocatable, 2 = executable,
                    // 3 = shared object, 4 = core image
    short archtype; // 2 = SPARC, 3 = x86, 4 = 68K, etc.
    int fversion;   // file version, always 1
    int entry;      // entry point if executable
    int phdrpos;    // file position of program header or 0
    int shdrpos;    // file position of section header or 0
    int flags;      // architecture specific flags, usually 0
    short hdrsize;  // size of this ELF header
    short phdrent;  // size of an entry in program header
    short phdrcnt;  // number of entries in program header or 0
    short shdrent;  // size of an entry in section header
    short shdrcnt;  // number of entries in section header or 0
    short strsec;   // section number that contains section name strings
} Elf32_Ehdr;
```

## ELF section header

```
typedef struct {
    int sh_name;    // name, index into the string table
    int sh_type;    // section type (PROGBITS, NOBITS, SYMTAB, …)
    int sh_flags;   // flag bits (ALLOC, WRITE, EXECINSTR)
    int sh_addr;    // base memory address(VMA), if loadable, or zero
    int sh_offset;  // file position of beginning of section
    int sh_size;    // size in bytes
    int sh_link;    // section number with related info or zero
    int sh_info;    // more section-specific info
    int sh_align;   // alignment granularity if section is moved
    int sh_entsize; // size of entries if section is an array
} Elf32_Shdr;
```

## ELF program header

```
typedef struct {
   int type;      // loadable code or data, dynamic linking info, etc.
   int offset;    // file offset of segment
   int virtaddr;  // virtual address to map segment (VMA)
   int physaddr;  // physical address (LMA)
   int filesize;  // size of segment in file
   int memsize;   // size of segment in memory (bigger if contains bss)
   int flags;     // Read, Write, Execute bits
   int align;     // required alignment, invariably hardware page size
} Elf32_Phdr;
```

Accusys
The RAID Architects

# Section header of a executable

```
$ objdump -h vmkernel

vmkernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000130  00200000  00200000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000049  00200140  00200140  00001140  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000044  0020018c  0020018c  0000118c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00002020  002001e0  002001e0  000011e0  2**5
                  ALLOC
  4 .comment      00000033  00000000  00000000  000011e0  2**0
                  CONTENTS, READONLY
```

Accusys
The RAID Architects

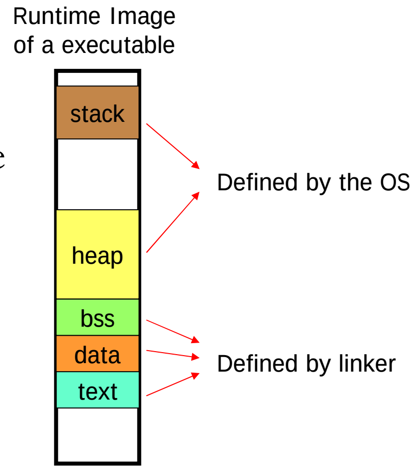
# Program header of a executable

```
$ objdump -p vmkernel

vmkernel:     file format elf32-i386

Program Header:
  LOAD off    0x00001000 vaddr 0x00200000 paddr 0x00200000 align 2**12
       filesz 0x000001d0 memsz 0x00002200 flags rwx
```

- Note: memsz > filesz
  - The difference (.bss section) is zero-inited by the operating system

## Where do we load?

- A program's address space is defined by linker and operating system together.




## Where do variables go?

| | | | .text | .rodata | .data | .bss | stack |
|---|---|---|---|---|---|---|---|
| global | static | initialized | | | v | | |
| | | non-initialized | | | | v | |
| | non-static | initialized | | | v | | |
| | | non-initialized | | | | v | |
| | const | | | v | | | |
| local | static | initialized | | | v | | |
| | | non-initialized | | | | v | |
| | non-static | initialized | | | | | v |
| | | non-initialized | | | | | v |
| | const | | v | | | | |
| Immediate value | | | v | | | | |

# Basic linker script concepts

- Linker scripts are often used to serve the following goals:
  - Change the way input sections are mapped to the output sections
  - Change LMA or VMA address of a section
    - LMA (load memory address): the address at which a section will be loaded
    - VMA (virtual memory address): the runtime address of a section
    - In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up
  - Define additional symbols for use in your code

# An example of linker script

- SECTIONS command defines a list of output sections
- '.' is the VMA location counter, which always refer to the current location in a output object
- '*' is a wildcard to match the filenames of all input objects
- '_etext' is a symbol of the value of current location counter
- AT command change the LMA of .data section
  - Only .data section has different addresses for LMA and VMA

Output section

input section

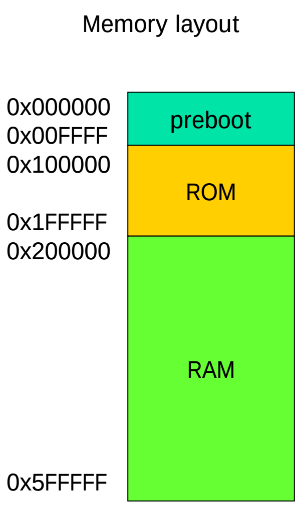
```
SECTIONS
{
   . = 0x10000;
   .text : { *(.text) }
   _etext = .;
   . = 0x80000000;
   .data : AT(_etext) { *(.data) }
   .bss: { *(.bss) }

}
```

# A mini-example demonstrating development of embedded system

- Runs in Linux user level for 3 reasons:
  - Learn most of the essential concepts without real hardware
  - Verify runtime memory contents with the help of GNU debugger
  - Avoid writing machine dependent code (switch into 32-bit protected mode on x86); besides, GCC cannot generate 16-bit code

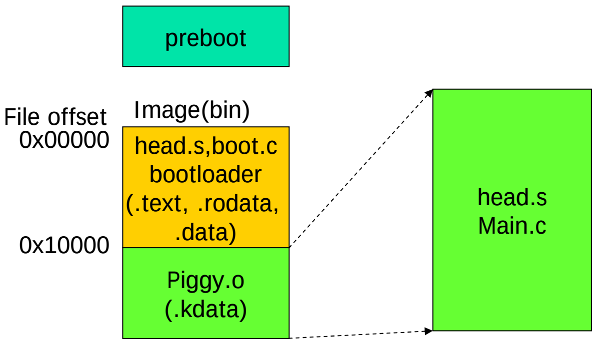
# Mini-example component overview

- preboot
  - preboot.c
  - This stage doesn't exist on real system. It is a helper loader to load boot image into ROM area
- Boot
  - head.s, boot.c
  - The boot loader, copy kernel from ROM to RAM
- Kernel
  - head.s, main.c
  - This is the kernel ☺



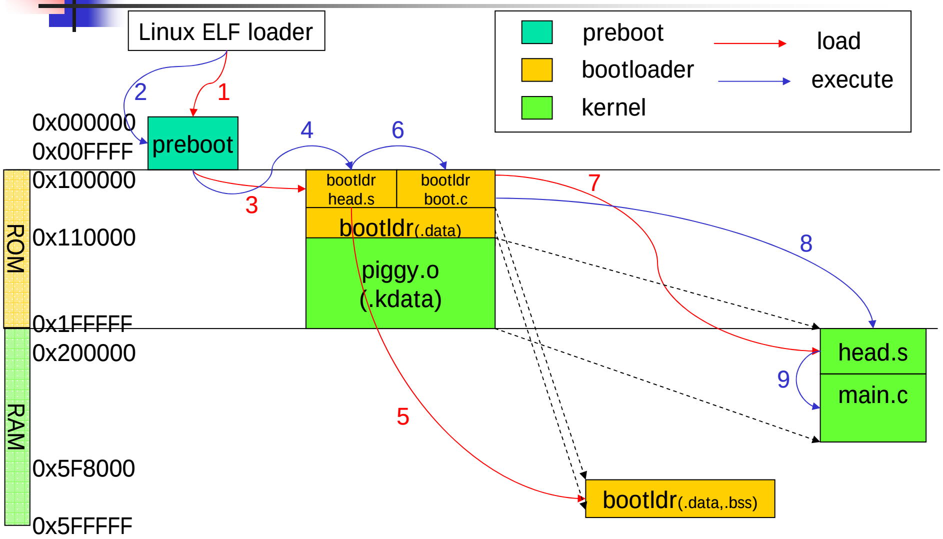
# Mini-example file layout

- There are 2 runnable files in this example
  - Preboot
  - Boot image
    - Boot loader
    - kernel




* Kernel (vmkernel.bin) is embedded inside piggy.o as a .kdata section

# Mini-example loading: step by step



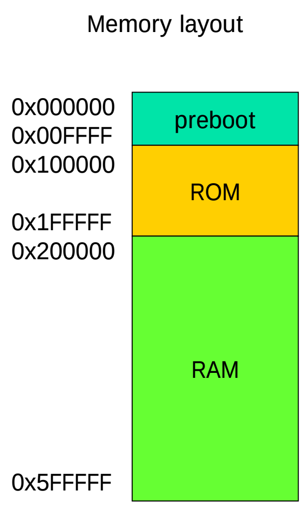
# The design of preboot.c

- To simplify the layout of runtime memory, no C library function, only system calls!
- _start() is the entry point! call _exit() system call to end the function
- Loaded at 0x0000 by Linux and then:
  - brk() to increase the boundary of data segment to 0x00600000
  - open() file “Image” and copy it to 0x100000
  - Jump to 0x100000

# Simplified preboot.c

```
#define READSIZE 1024
#define IMG_FILENAME "Image"
const unsigned long brkptr = 0x00600000;
const unsigned long boot_start = 0x00100000;

void _start() {
    int imgfd, i, byte_read;
    char *ptr = (char *)boot_start;

        write(STDOUT_FILENO, pbmsg, sizeof(pbmsg));

    /* get more space to copy Image to ROM */
    brk(brkptr);

    imgfd = open("Image", O_RDONLY, 0));

    /* copy Image contents to ROM */
    i = 0;
    while (1) {
          byte_read = read(imgfd, ptr+i, READSIZE);
           if (byte_read < READSIZE) break;
    }
    /* jump to boot loader */
    ((void (*)())boot_start)();
}
```

System calls

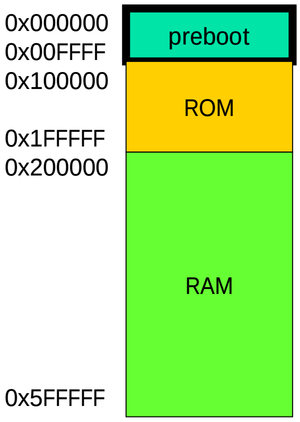


## Makefile related to preboot

```
# pre-boot loader address map
PREBOOT_TEXT = 0x00000000
PREBOOT_DATA = 0x00002000  1
PREBOOT_LDFLAGS = -Ttext $(PREBOOT_TEXT) \
                  -Tdata $(PREBOOT_DATA)

2 %.o: %.c
          $(CC) $(CFLAGS) -c $< 3

# preboot is in ELF format. run it to load Image
preboot: preboot.o
          $(LD) $(PREBOOT_LDFLAGS) -o $@ $<
                                       4
```

1. Tell linker the runtime start address of text and data sections
2. Generic pattern rules to make a .o file from a .c
   ex: preboot.o -> preboot.c
3. The name of first prerequisite. That is %.c
4. The filename of the target of the rule

Memory layout




## Let's check preboot memory layout - objdump & readelf

objdump

```
start address 0x00000000

Program Header:
    LOAD off    0x00001000 1vaddr 0x00000000 paddr 0x00000000 align 2**12
         filesz 0x00000487 memsz 0x00000487 flags r-x
    LOAD off    0x00002000 vaddr 0x00002000 paddr 0x00002000 align 2**12
         filesz 0x00000000 memsz 0x00000000 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000322  00000000  00000000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00002000  00002000  00002000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rodata       00000147  00000340  00000340  00001340  2**5
                  CONTENTS, 2ALLOC, LOAD, READONLY, DATA
  3 .bss          00000000  00002000 3 00002000  00002000  2**2
                  ALLOC
```

1 = 2 + 3

readelf

```
 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata
   01
```

# Preboot memory layout - verify from procfs

- /proc/<pid>/maps
  - This shows runtime memory map

```
>cat /proc/3167/maps
00000000-00001000 r-xp 00001000 08:02 303766     /home/josephl/embed_example/boot/preboot
00002000-00600000 rwxp 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0

    brk()
```

# The design of the boot loader

- Typical boot loader does:
  - Initialize CPU DRAM register
  - Setup stack register
  - Copy .data from ROM to RAM and zero-init .bss on RAM
  - Copy kernel to RAM
- The only difference here
  - No DRAM register init!
- NOTE
  - bootloader executes on the ROM!
  - If there is no global or static variable => we don't need to copy .data or zero-init .bss
  - Here, copy of .data and zero-init of .bss can be written in C!

# Simplified boot loader - head.s and boot.c

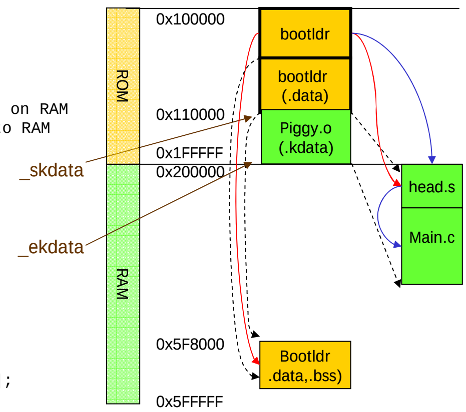
head.s

```
.text
     .globl startup_32

startup_32:
     movl $0x600000,%esp # setup stack pointer
     call clear_bss      # zero-init bss section on RAM
     call init_data      # copy .data from ROM to RAM
     call start_boot
```

defined in piggy.lds

boot.c

```
extern char _skdata[], _ekdata[];
const unsigned long kernel_start = 0x00200000;

void start_boot() {
     int i, copylen;

     /* copy kernel to RAM */
     copylen = _ekdata - _skdata;
     for (i=0; i < copylen; i++)
          *(char *)(kernel_start+i) = _skdata[i];

     /* jump to kernel */
     write(STDOUT_FILENO, bootjump, sizeof(bootjump));
     ((void (*)())kernel_start)();
}
```

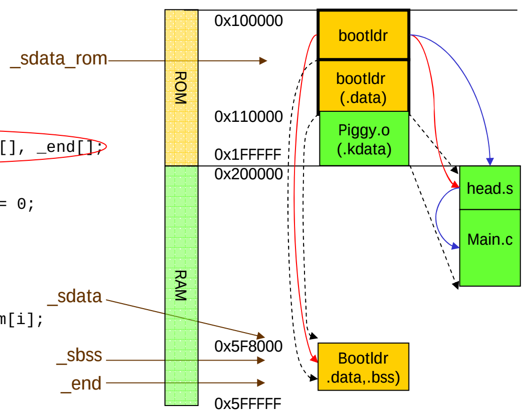



# Simplified boot loader code - boot.c

Those ***symbols*** are defined in boot.lds

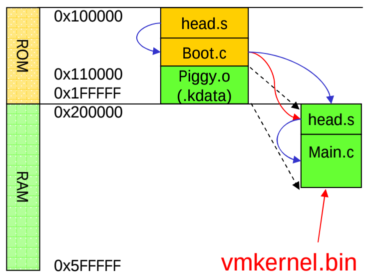
boot.c

```
extern char _sdata_rom[], _sdata[], _sbss[], _end[];
void clear_bss() {
          int i, len = _end - _sbss;
          for (i=0; i<len; i++) _sbss[i] = 0;
}
void init_data() {
          int i, len = _sbss - _sdata;

          for (i=0; i<len; i++)
                    _sdata[i] = _sdata_rom[i];
}
```



## The design of the kernel

- Loading the kernel
  - The actual kernel got loaded is in binary format and can be run directly once it is copied to the RAM.
  - vmkernel.bin is the runtime image of the kernel with .text and .data sections ready!
  - Kernel needs to initialize .bss and stack pointer by itself.




## Simplified kernel code - head.s, main.c

- stack area is inside the .bss section

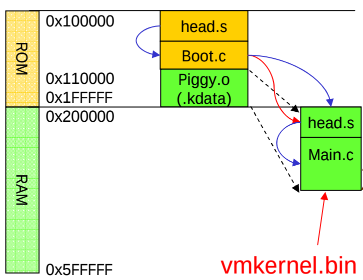
head.s

```
.text
            .globl startup_32
startup_32:
            movl stack_start,%esp # setup stack pointer
            call start_kernel
```

main.c

```
#define STACK_SIZE 8192
char stack_space[STACK_SIZE]; /* in .bss section */
char *stack_start = &stack_space[STACK_SIZE];
extern char __bss_start[];
extern char _end[];

void clear_bss() {
            int i, len = _end - __bss_start;
            for (i=0; i<len; i++)
                        __bss_start[i] = 0;
}

start_kernel() {
            clear_bss();
            for(;;);
}
```

clear_bss() in start_kernel() makes start_kernel() unable to return, but this is not an issue since kernel never returns.



# Building the kernel - vmkernel.lds, Makefile

- Output object format: ELF

Makefile

```
LDFLAGS = -T vmkernel.lds        1

all: $(SYSTEM)        2

%.o: %.S
		$(CC) $(CFLAGS) -c $<
%.o: %.c
		$(CC) $(CFLAGS) -c $<
$(SYSTEM): head.o main.o
		$(LD) $(LDFLAGS) -o $@ $^
```

vmkernel.lds

```
3 ENTRY(startup_32)
SECTIONS {
	. = 0x00200000;  4
	.text : { *(.text) }
	.rodata : { *(.rodata) }
	.data : { *(.data) }
	.bss : {
		__bss_start = .;
		*(.bss)
	}
	_end = .;
}
```

1. Specify a linker script to be used
2. The filename of kernel in ELF format
3. Entry point of the program
4. Start address of the kernel

# Building the Image file - Makefile

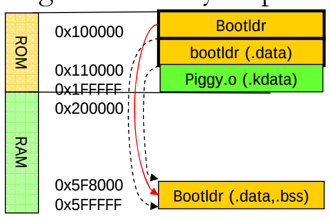
- Output object format: binary

1. Specify a linker script to be used
2. The filename of kernel in ELF format
3. Generate a relocateable output
4. The format of input object (vmkernel.bin)
5. The format of output object (piggy.o)
6. Make a binary object from a ELF one

```
IMAGE_LDFLAGS = -T boot.lds        1

# generic pattern rules to compile a .c to a .o
%.o: %.c
		$(CC) $(CFLAGS) -c $<
%.o: %.S
		$(CC) $(CFLAGS) -c $<

# kernel must be in binary format since ELF loader is not available
piggy.o: $(SYSTEM)  2
		$(OBJCOPY) -O binary $(SYSTEM) vmkernel.bin        (3)
		$(LD) -o $@ -r --format binary --oformat elf32-i386 vmkernel.bin -T piggy.lds        (1)
		rm -f vmkernel.bin        (4)        (5)

Image: head.o boot.o piggy.o
		$(LD) $(IMAGE_LDFLAGS) -o $@.elf $^
		$(OBJCOPY) -O binary $@.elf $@  6
```

# The design of Image file - piggy.lds, boot.lds

- Image.elf memory map:




| section name | Image.elf File offset | LMA | VMA | size |
|---|---|---|---|---|
| .text | 0x00001000 | 0x00100000 | 0x00100000 | 0x253 |
| .rodata | 0x00001260 | 0x00100260 | 0x00100260 | 0x10F |
| .data | 0x00002000 | 0x0010036F | 0x005F8000 | 0x050 |
| .bss | N/A | N/A | 0x005F8050 | 0x004 |
| .kdata | 0x00003000 | 0x00110000 | 0x00110000 | 0x1D0 |

piggy.lds

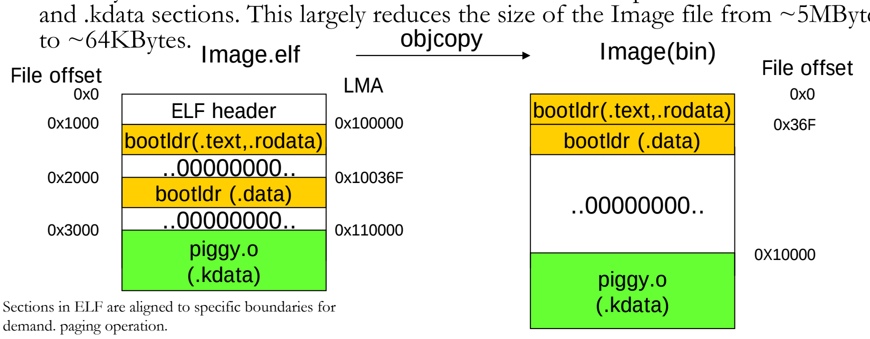
```
SECTIONS
{
  .kdata : {
      _skdata = .;
      *(.data)
      _ekdata = .;
  }
}
```

boot.lds

```
ENTRY(startup_32)
SECTIONS {
  . = 0x00100000;
  .text : {*(.text) }
  .rodata : { *(.rodata) }
  _sdata_rom = .;
  . = 0x00110000;
  .kdata : { *(.kdata) }
  . = 0x005F8000;
  _sdata = .;
  .data : AT(_sdata_rom) { *(.data) }

  _sbss = .;
  .bss : { *(.bss) }
  _end = .;
}
```

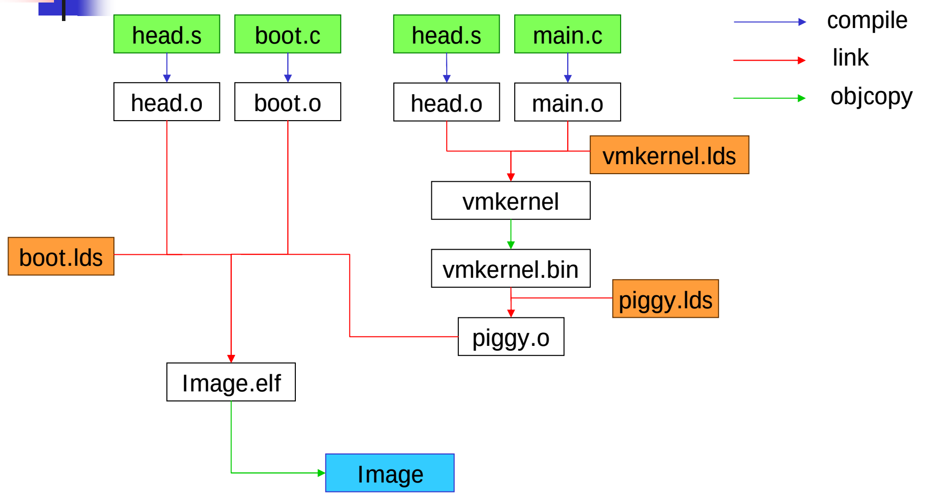
# The design of Image file - objcopy

- objcopy consults the **LMA address** of each section in the input object when making a binary object. It reorders sections by their LMA addresses in ascending order and copy those sections in that arranged order to the output object, starting from the first LMA address. If there is any gap between two sections, it fill the gap with zeros.
- 'AT' keyword moves .data section from 0x5f8000 to the space between .rodata and .kdata sections. This largely reduces the size of the Image file from ~5MBytes to ~64KBytes.




Sections in ELF are aligned to specific boundaries for demand. paging operation.

# The complete picture of building the 'Image' file




# Use objdump and gdb to verify the design

- objdump
  - Verify Image.elf section header and symbol table
  - Disassemble Image.elf
- gdb
  - dump runtime memory contents to a file
    - Use 'hexdump –C' to compare the file contents with the corresponding section in the Image.elf

## Reference

- John R. Levine, Linkers and Loaders, Morgan Kaufmann, 2000
- Using ld, Free Software Foundation, 2000
- Linux 2.4 kernel source

# Kernel Debugging

Hao-Ran Liu

# Debugging tools available

- printk() – print message on screen
- /proc file system
- strace – system call trace
- Kernel oops
- sysrq magic key
- Kernel profiling
- gdb / kdb/ kgdb
- User-mode Linux
- Dynamic Probes

# printk()

- Kernel-space equivalent of printf()
- Each kernel message are prepended a string representing its loglevel n
  – “<n>Hello world!”
- Loglevel determines the severity of the message

# Printk loglevel

- Messages with level lower than *console_loglevel* are shown to the console
- *console_loglevel* can be changed via
  – At klogd command line (klogd –c n)
  – syslog system call
  – echo n > /proc/sys/kernel/printk

```
#define KERN_EMERG    "<0>" /* system is unusable    */
#define KERN_ALERT    "<1>" /* action must be taken immediately */
#define KERN_CRIT     "<2>" /* critical conditions   */
#define KERN_ERR      "<3>" /* error conditions */
#define KERN_WARNING  "<4>" /* warning conditions */
#define KERN_NOTICE   "<5>" /* normal but significant condition */
#define KERN_INFO     "<6>" /* informational */
#define KERN_DEBUG    "<7>" /* debug-level messages */
```

# Kernel log buffer

- kernel log buffer stores kernel messages
- It is a circular buffer. Old messages are overwritten when the buffer is full
  - Use klogd daemon to keep old msgs in a file
  - Log buffer size is configurable
- Kernel log buffer can be manipulated via syslog system call
  - or dmesg command line tool

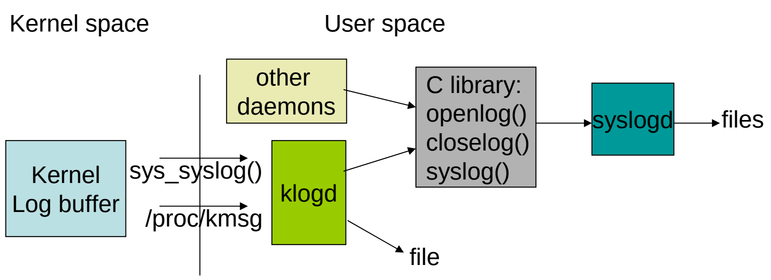
# syslog system call

- int syslog(int type, char *bufp, int len)

```
/*
 * Commands to sys_syslog:
 *
 *      0 -- Close the log.  Currently a NOP.
 *      1 -- Open the log. Currently a NOP.
 *      2 -- Read from the log.
 *      3 -- Read up to the last 4k of messages in the ring buffer.
 *      4 -- Read and clear last 4k of messages in the ring buffer
 *      5 -- Clear ring buffer.
 *      6 -- Disable printk to console
 *      7 -- Enable printk to console
 *      8 -- Set level of messages printed to console
 *      9 -- Return number of unread characters in the log buffer
 */c
```

# Klogd and syslogd

- Klogd is “kernel log daemon”. It receives kernel messages via syslog system call (or /proc/kmsg) and redirect them to syslogd
- syslogd differentiate messages by facility.priority (ex. LOG_KERN.LOG_ERR) and consults /etc/syslog.conf to know how to deal with them (discard or save in a file)




# Turn individual or all messages off

- Do not remove debug printk, you may need it later (debug another related bug)
- Remove MY_DEBUG to turn all debug messages off in production kernel

```
#undef PDEBUG /* undef it, just in case */
#ifdef MY_DEBUG
# ifdef __KERNEL__
/* This one if debugging is on, and kernel space */
# define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ## args)
# else
/* This one for user space */
# define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
# endif
#else
# define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif
#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */
```

# Limit the rate of your printk

- Printk may overwhelm the console if
  - printk in a code which get executed very often
  - printk in a frequently-triggered IRQ handler (eg. Timer)
- printk_ratelimit() return 0 when message to be printed should be surpressed

```
if (printk_ratelimit( ))
        printk(KERN_NOTICE "The printer is still on fire\n");
```

# printk_ratelimit() implementation

- The two variable can be modified via /proc/sys/kernel/

```
/* minimum time in jiffies between messages */
int printk_ratelimit_jiffies = 5*HZ;

/* number of messages we send before ratelimiting */
int printk_ratelimit_burst = 10;

int printk_ratelimit(void)
{
        return __printk_ratelimit(printk_ratelimit_jiffies,
                                  printk_ratelimit_burst);
}
```

# printk_ratelimit() implementation

```
int __printk_ratelimit(int ratelimit_jiffies, int ratelimit_burst) {
   static spinlock_t ratelimit_lock = SPIN_LOCK_UNLOCKED;
   static unsigned long toks = 10*5*HZ, last_msg;
   static int missed;
   unsigned long flags, now = jiffies;

   spin_lock_irqsave(&ratelimit_lock, flags);
   toks += now - last_msg;
   last_msg = now;
   if (toks > (ratelimit_burst * ratelimit_jiffies))
      toks = ratelimit_burst * ratelimit_jiffies;
   if (toks >= ratelimit_jiffies) {
      int lost = missed;
      missed = 0;
      toks -= ratelimit_jiffies;
      spin_unlock_irqrestore(&ratelimit_lock, flags);
      if (lost)
         printk(KERN_WARNING "printk: %d messages suppressed.\n", lost);
      return 1;
   }
   missed++;
   spin_unlock_irqrestore(&ratelimit_lock, flags);
   return 0;
}
```

# /proc file system

- A software-created, pseudo file system
- Contains many system information, ex:
  - /proc/<pid>/maps
  - /proc/sys/kernel/*
  - /proc/interrupts
  - /proc/meminfo
- Use of /proc fs is discouraged, they should contain only inoformation about process
- You should use sysfs instead, new in 2.6 kernel

# Create/Remove an entry in /proc

- Create

```
struct proc_dir_entry *create_proc_read_entry(const char *name,
                        mode_t mode, struct proc_dir_entry *base,
                        read_proc_t *read_proc, void *data);
```

  – name: name of the file to create
  – mode: access permission bit (0 = default)
  – base: the directory in which file is created (0 = /proc)
  – read_proc: pointer to the function providing contents of the file
  – data: private data, will be passed to read_proc

- Remove

```
void remove_proc_entry(const char *name,
                        struct proc_dir_entry *parent)
```

# read_proc_t

- Prototype

```
typedef int (read_proc_t)(char *page, char **start, off_t off,
                          int count, int *eof, void *data);
```

- Argument
  – page: page where you write your data to
  – start: where (within page) the data to be returned is found
  – offset, count: same as normal read()
  – eof: fill by read_proc, 1 indicates EOF
  – data: private data

# read_proc Example

```
void __init proc_misc_init(void) {
   struct proc_dir_entry *entry;
   static struct {
        char *name;
        int (*read_proc)(char*,char**,off_t,int,int*,void*);
   } *p, simple_ones[] = {
        {"loadavg",      loadavg_read_proc},
        {"uptime",       uptime_read_proc},
        {"meminfo",      meminfo_read_proc},
        {"version",      version_read_proc},
        {"devices",      devices_read_proc},
        {"filesystems",  filesystems_read_proc},
        {"cmdline",      cmdline_read_proc},
        {"locks",        locks_read_proc},
        {"execdomains",  execdomains_read_proc},
        {NULL,}
   };
   for (p = simple_ones; p->name; p++)
       create_proc_read_entry(p->name, 0, NULL, p->read_proc, NULL);
```

# read_proc Example

```
static int version_read_proc(char *page, char **start, off_t off,
                                int count, int *eof, void *data)
{
        extern char *linux_banner;
        int len;

        strcpy(page, linux_banner);
        len = strlen(page);
        return proc_calc_metrics(page, start, off, count, eof, len);
}
```

# strace: system call trace

- Intercepts and records
  - system calls issued by a process
  - signals a process received
- Where to use
  - Have a in indepth understanding of the exactly behavior of a program
  - Debug the exactly argument or system call a program issued
  - When you don't have access to the source code
- Syntax
  - strace [option] <command [args]>
- Common option
  - -c -- count time, calls, and errors for each syscall and report summary
  - -f -- follow forks
  - -T -- print time spent in each syscall
  - -e expr -- a qualifying expression: option=[!]all or option=[!]val1[,val2]... (options: trace, abbrev, verbose, raw, signal, read, or write)

# strace output example

```
execve("/bin/dmesg", ["dmesg"], [/* 22 vars */]) = 0
...
syslog(0x3, 0x95d3858, 0x4008)          = 16384
write(1, "amily 2\nIP: routing cache hash t"..., 4096amily
write(1, "to accept 2 bytes to c1bd7f9e fr"...,
...
munmap(0xb7d6b000, 4096)                = 0
exit_group(0)                           = ?
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 92.75    0.013263          35       374           write
  5.02    0.000718         718         1           syslog
  0.51    0.000073          18         4         1 open
  0.47    0.000067          34         2           munmap
  0.41    0.000058          12         5           old_mmap
  0.34    0.000048          24         2           mmap2
  0.11    0.000016           4         4           fstat64
  0.10    0.000015          15         1           read
  0.10    0.000015           8         2           mprotect
  0.08    0.000012           3         4           brk
  0.04    0.000006           2         3           close
  0.04    0.000006           6         1           uname
  0.02    0.000003           3         1           set_thread_area
------ ----------- ----------- --------- --------- ----------------
100.00    0.014300                   404         1 total
```

# Kernel oops

- When kernel detects some bug in itself
  - Fault: Kernel kill faulting process and try to continue
  - Panic: system halts, usually in interrupt context or in idle, init task where kernel think it cannot recover itself
- Oops message contains
  - Error message
  - Contents of registers
  - Stack dump
  - Function call trace
- Enable CONFIG_KALLSYMS at kernel configuration to have symbolic call trace (otherwise all you see are binary addresses)

# Kernel Oops Example

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
printing eip:
d083a064
Oops: 0002 [#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=cfa0a6c0)
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
```

**kernel func. ret.**

**User space addr.**

## sysrq magic key

- Useful when the kernel or keyboard hang but still responsive to keyboard interrupts
- Sysrq magic key prints out the information you request according to the magic key you pressed
- How to enable?
  - CONFIG_MAGIC_SYSRQ option must be selected
  - sysrq can be enabled/disabled at runtime
    - Echo 0 or 1 > /proc/sys/kernel/sysrq
- How to use?
  - Press alt-sysrq(printscreen)-<key> at your console or;
  - Echo sysrq key character to /proc/sysrq-trigger and read kernel message from kernel logs

## List of sysrq magic keys

```
'r'     - Turns off keyboard raw mode and sets it to XLATE.
'k'     - Secure Access Key (SAK) Kills all programs on the
          current virtual console.
'b'     - Will immediately reboot the system without syncing or
          unmounting your disks.
'o'     - Will shut your system off (if configured and supported).
's'     - Will attempt to sync all mounted filesystems.
'u'     - Will attempt to remount all mounted filesystems read-only.
'p'     - Will dump the current registers and flags to your console.
't'     - Will dump a list of current tasks and their information to
          your console.
'm'     - Will dump current memory info to your console.
'v'     - Dumps Voyager SMP processor info to your console.
'0'-'9' - Sets the console log level, controlling which kernel
          messages will be printed to your console. ('0', for example
          would make it so that only emergency messages like PANICs
          or OOPSes would make it to your console.)
'e'     - Send a SIGTERM to all processes, except for init.
'i'     - Send a SIGKILL to all processes, except for init.
'l'     - Send a SIGKILL to all processes, INCLUDING init.
          (Your system will be non-functional after this.)
'h'     - Will display help
```

# Kernel profiling

- Get a picture of where in the kernel CPU spends most of its time
- How to enable?
  - Enable CONFIG_PROFILING
  - Boot the kernel with profile=2 command
- How to use?
  - 'readprofile –r' to reset profile buffer
  - 'readprofile –m /boot/System.map'

# Kernel profiling Example

| Number of clock ticks occurred | The name of the function where ticks occurred | Ratio |
|---|---|---|
| 1334 | default_idle | 27.7917 |
| 4 | cpu_idle | 0.0208 |
| 2 | get_wchan | 0.0125 |
| 1 | arch_align_stack | 0.0312 |
| 1 | restore_sigcontext | 0.0027 |
| 1 | setup_frame | 0.0020 |
| 1 | handle_signal | 0.0026 |
| 2 | ret_from_intr | 0.0625 |
| 26 | sysenter_past_esp | 0.2149 |
| 5 | system_call | 0.1000 |
| 1 | syscall_exit | 0.0667 |
| 1 | do_gettimeofday | 0.0048 |
| 1 | do_mmap2 | 0.0048 |
| 2 | sched_clock | 0.0139 |
| 103 | delay_pmtmr | 3.2188 |

# The idea behind kernel profiling

- The kernel text section address range are mapped to a counter array
- If the granularity is n (profile=n), the array length is address range / n
- The corresponding counter is increased by 1 when the clock ticks.
- The result of the profiling may be misleading if the granularity is coarse

# gdb – observe kernel variables

- gdb can observe variables in the kernel
- How to use?
  - gdb /usr/src/linux/vmlinux /proc/kcore
  - p jiffies /* print the value of jiffies variable */
  - p jiffies /* you get the same value, since gdb cache value readed from the core file */
  - core-file /proc/kcore /* flush gdb cache */
  - p jiffies /* you get a different value of jiffies */
- vmlinux is the name of the uncompressed ELF kernel executable, not bzImage
- kcore represent the kernel executable in the format of a core file
- Disadvantage
  - Read-only access to the kernel

# kdb – assembly-level debugger

- Kernel built-in debugger
  - Not available in the official kernel
  - Available as a patch from oss.sgi.com
- When your kdb is enabled, press 'Pause' key to start the debugger
- You can disassembling, set breakpoint, single step, modify memory, continue execution of the kernel
- Disadvantage
  - Assembly-level debugging

# kgdb –source-level debugger

- Patch the kernel to include gdb stub in
  - Download patch from http://kgdb.sf.net
- gdb and the target kernel must be run on separate machine; gdb connects to the target through the serial port or ethernet
- How to use?
  - Boot the kernel with the command: kgdbwait kgdb8250=0,115200
  - When the kernel displays "waiting for connection from remote gdb…", start gdb
  - At gdb, load vmlinux, set baud rate and connect to the target
    - file vmlinux
    - set remotebaud 115200
    - target remote /dev/ttyS0

# User mode Linux

- Running the Linux kernel as a user mode process (on top of Linux system call interface, instead of hardware)
- You can use gdb to debug the kernel just like a normal user process
- Download at http://user-mode-linux.sf.net

# Dynamic Probes

- Allow you to place ‘probe’ anywhere
  – To observe system status
  – To change system flow
- Without recompiling the kernel
- Download at http://dprobes.sf.net

# References

- Linux Device Drivers, 3rd Edition, Jonathan Corbet
- Linux kernel source 2.6.10

# Kernel modules

Hao-Ran Liu

# Kernel modules

- Kernel modules can be loaded into or unloaded from the kernel at runtime
  - Extend the the functionality of the kernel
  - Save memory (Unused feature are not loaded)
  - User do not have to recompile the kernel for their hardware (a base kernel image plus a set of binary modules works!)

# Hello world! module

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
        printk(KERN_ALERT "Hello, world\n");
        return 0;
}

static void hello_exit(void)
{
        printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

# Understanding the Hello world! Module

- module_init()
  - Register the function to be called when the module is loaded
- module_exit()
  - Register the function to be called when the module is unloaded
  - Exit function will not be compiled into the kernel if the module is compiled into the kernel image
- MODULE_LICENSE()
  - specifies the copyright license for this module
  - Kernel complains when a GPL-incompatible module is loaded
  - Non-GPL module cannot use GPL-only symbols

# Kernel module programming

- Like event-driven programming, kernel modules registers to the kernel the capability (functions) it have.
- Kernel module can use only symbols (variables and functions) kernel exports
- Your code must be reentrant; use the right synchronization tools to deal with the concurrency issue
- The task_struct of the current process can be accessed via **current** MACRO

# Compiling kernel modules in the kernel tree

- At the parent directory of your driver, ex. /drivers/char/, add a line in the Makefile
  - obj-$(CONFIG_MYDRIVER) += mydriver/
  - This like cause the kbuild system to go into your directory when it compiles the kernel or kernel modules
- In your driver's dir., you add a new Makefile with this line
  - obj-$(CONFIG_MYDRIVER) += mydriver.o
  - This line cause the kbuild to build a kernel module mydriver.ko from mydriver.c
- If your driver contains more than one file, add one more line
  - mydriver-objs := file1.o file2.o
  - This causes the kbuild to build mydriver.ko from file1.c and file2.c

# Compiling kernel modules outside the kernel tree

```
# To build modules outside of the kernel tree, we run "make"
# in the kernel source tree; the Makefile these then includes this
# Makefile once again. This conditional selects whether we are being
# included from the kernel Makefile or not.
ifeq ($(KERNELRELEASE),)
    # Assume the source tree is where the running kernel was built
    # You should set KERNELDIR in the environment if it's elsewhere
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    # The current directory is passed to sub-makes as argument
    PWD := $(shell pwd)
modules:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules_install:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
else
    # called from kernel build system: just declare what our modules
are
    obj-m := mydriver.o
endif
```

# Installing the module

- ‘make module_install’ will install your module at /lib/modules/<kernel version>/kernel

# Module related tools

- insmod <module.ko> [module parameters]
  - Load the module
- rmmod
  - Unload the module
- modprobe [-r] <module name>
  - Load the module specified and modules it depends
  - Unload the module specified and other modules it depends if they have no other user
- lsmod
  - List all modules loaded into the kernel
- depmod
  - Regenerate module dependency information

# Kconfig

- Kconfig is a kernel configuration-option file
  - make menuconfig read this file to give you a list of all optional config items.
  - make menuconfig saves user's configuration at <Kernel source root>/.config
- Creating Kconfig for your driver
  - At the parent directory of your driver, ex drivers/char, add a line to the Kconfig file there
    - source "drivers/char/mydriver/Kconfig"
  - This line tells the kbuild system to read your own Kconfig file

# Kconfig

- In your directory, /drivers/char/mydriver, create a Kconfig file there

```
config FISHING_POLE
        tristate "Fish Master XL support"
        default n
        help
                If you say Y here, support for the Fish Master XL 2000 Titanium with
                computer interface will be compiled into the kernel and accessible via
                device node. You can also say M here and the driver will be built as a
                module named fishing.ko.

                If unsure, say N.
```

  – Don't write CONFIG_ prefix
  – tristate means the module can be built into the kernel (y), not built at all (n) or as a kernel module (m)
  – Default is n

# Other Kconfig directive

- bool: like tristate, but allows only (y/n)
- depends on: this option cannot be enabled unless the other it depends is enabled
- select XXX: XXX is automatically enabled if this option is selected
- bool, tristate and 'depends on' can be followed by an 'if', which makes the entire option conditional on another configuration option. If the condition is not met, the configuration is not only disabled but also disappear in the menuconfig

# Another Kconfig example

```
config NR_CPUS
        int "Maximum number of CPUs (2-255)"
        range 2 255
        depends on SMP
        default "32" if X86_NUMAQ || X86_SUMMIT || X86_BIGSMP || X86_ES7000
        default "8"
        help
          This allows you to specify the maximum number of CPUs which this
          kernel will support.  The maximum supported value is 255 and the
          minimum value which makes sense is 2.

          This is purely to save memory - each supported CPU adds
          approximately eight kilobytes to the kernel image.
```

# Module parameters

- Defining module parameters
  - module_param(name, type, perm);
- argument:
  - name: the name of the exported parameter and internal variable
  - type: can be byte, short, ushort, int, uint, long, ulong, charp, bool, invbool
  - perm: the access permission for corresponding file in sysfs

```
module parameter controlling the capability to allow live bait on the pole */
static int allow_live_bait = 1;           /* default to on */
module_param(allow_live_bait, bool, 0644); /* a Boolean type */
```

Accusys

# Module parameters

- module_param_named(name, variable, type, perm);
  - Different name for  parameter and variable
- module_param(strptr, charp, 0)
  - Kernel copy the string from the user space and assign strptr to point to it
- module_param_string(name, string, len, perm);
  - Kernel copy parameter string into the buffer you provide

```
static char species[BUF_LEN];
module_param_string(specifies, species, BUF_LEN, 0);
```

- module_param_array(name, type, nump, perm);
  - Comma-separated list of parameters can be stored in an array

```
static int fish[MAX_FISH];
static int nr_fish;
module_param_array(fish, int, &nr_fish, 0444);
```

Accusys

# Exported symbols

- Kernel modules can only access symbols explicitly exported
- EXPORT_SYMBOL()
- EXPORT_SYMBOL_GPL()
  - This symbol can only be used by modules licensed under GPL

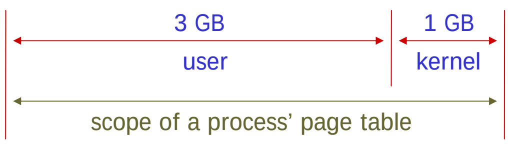
# Physical Memory Management in Linux

Hao-Ran Liu

# Table of Contents

# Virtual Address Space and Memory Allocators in Linux

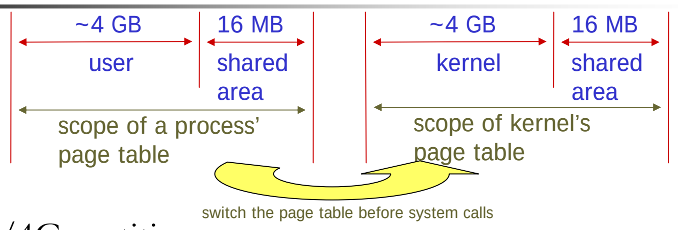
# Linux Virtual Address Layout




- 3G/1G partition
  - The way Linux partition a 32-bit address space
  - Cover user and kernel address space at the same time
  - Advantage
    - Incurs no extra overhead (no TLB flushing) for system calls
  - Disadvantage
    - With 64 GB RAM, `mem_map` alone takes up 512 MB memory from lowmem (ZONE_NORMAL).

Accusys

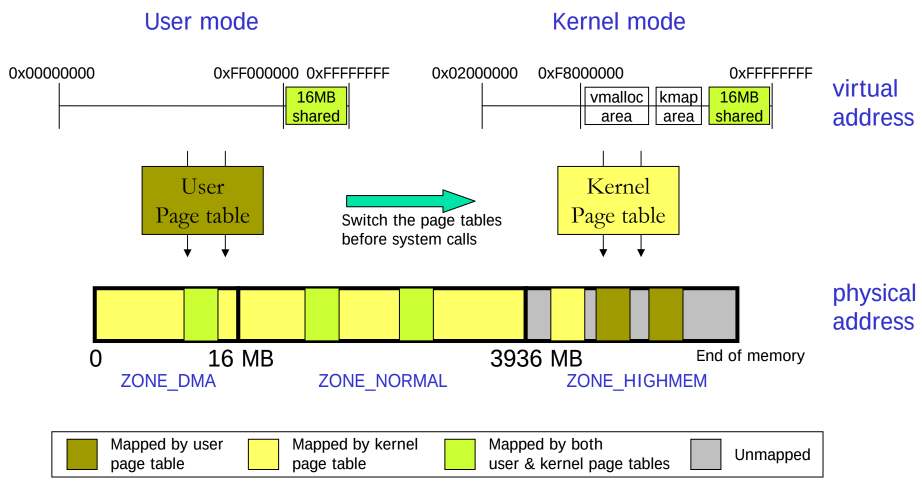
# Linux Virtual Address Layout



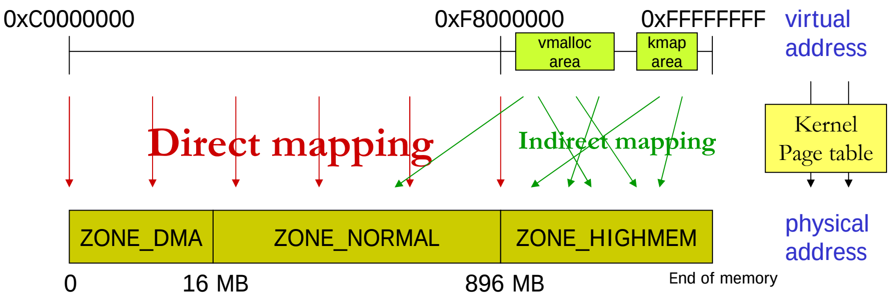

- 4G/4G partition
  - Proposed by Red Hat to solve `mem_map` problem
  - Disadvantage (Performance drop!)
    - Switch page table and flush TLB for every system call!
    - Data is copied "indirectly" (with the help of `kmap`) between user and kernel space
  - Advantage
    - Only on machine with large RAM

Accusys

# Page Table Switch in a 4G/4G Configuration



# Partition of Physical Memory (Zone)

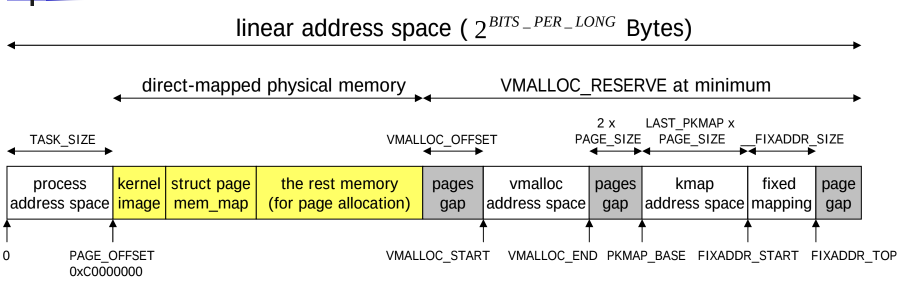



This figure shows the partition of physical memory and its mapping to virtual address in 3G/1G layout

# Why not map kernel memory indirectly?

- Reasons for direct mapping
  - No changes of kernel page table for contiguous allocation in physical memory
  - Faster translation between virtual and physical addresses
- Implications of direct mapping
  - kernel memory is not swappable

# Kernel Virtual Address Space



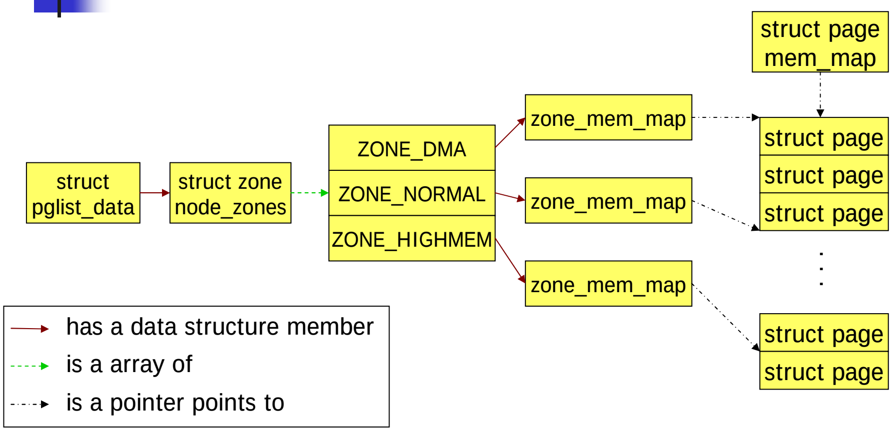

- vmalloc address space
  - Noncontiguous physical memory allocation
- kmap address space
  - Allocation of memory from ZONE_HIGHMEM
- Fixed mapping
  - Compile-time virtual memory allocation

# Memory Allocators in Linux

| | Description | Used at | functions |
|---|---|---|---|
| Boot Memory Allocator | 1. A first-fit allocator, to allocate and **free** memory during kernel boots<br>2. Can handle allocations of sizes smaller than a page | System boot time | `alloc_bootmem()`<br>`free_bootmem()` |
| Physical Page Allocator (buddy system) | 1. **Page-size** physical frame management<br>2. Good at dealing with **external fragmentation** | After `mem_init()`, at which boot memory allocator retires | `alloc_pages()`<br>`__get_free_pages()` |
| Slab Allocator | 1. Deal with **Internal fragmentation** (**for allocations < page-size**)<br>2. Caching of commonly used objects<br>3. Better use of the hardware cache | After `mem_init()`, at which boot memory allocator retires | `kmalloc()`<br>`kfree()` |
| Virtual Memory Allocator | 1. Built on top of page allocator and **map noncontiguous physical pages to logically contiguous vmalloc space**<br>2. Required altering the kernel page table<br>3. Size of all allocations <= vmalloc address space | 1. Large allocation size<br>2. contiguous physical memory is not available | `vmalloc()`<br>`vfree()` |

## Describing Physical Memory

## Data Structures to Describe Physical Memory




All these data structures are initialized by `free_area_init()` at `start_kernel()`

## Page Tables vs. `struct page`s

- Page tables
  - Used by CPU memory management unit to map virtual address to physical address
- `struct page`s
  - Used by Linux to keep track of the status of all physical pages
  - Some status (eg. dirty, accessed) is read from the page tables.

## Nodes

- Designed for NUMA (Non-Uniform Memory Access) machine
- Each bank (The memory assigned to a CPU) is called a node and is represented by `struct pglist_data`
- On Normal x86 PCs (which use UMA model), Linux uses a single node (`contig_page_data`) to represent all physical memory.

# struct pglist_data

| Type | Name | Description |
|---|---|---|
| struct zone [] | node_zones | Array of zone descriptors of the node |
| struct zonelist [] | node_zonelists | The order of zones that allocations are preferred from |
| int | nr_zones | Number of zones in the node |
| struct page * | node_mem_map | This is the first page of the struct page array that represents each physical frame in the node |
| struct bootmem_data * | bdata | Used by boot memory allocator during kernel initialization |
| unsigned long | node_start_pfn | The starting physical page frame number of the node |
| unsigned long | node_present_pages | Total number of physical pages in the node |
| unsigned long | node_spanned_pages | Total size of physical page range, including holes |
| int | node_id | Node ID (NID) of the node |
| struct pglist_data * | pgdat_next | Pointer to next node in a NULL terminated list |

# Zones

- Because of hardware limitations, the kernel cannot treat all pages as identical
  - Some hardware devices can perform DMA only to certain memory address
  - Some architectures cannot map all physical memory into the kernel address space.
- Three zones in Linux, described by `struct zone`
  - ZONE_DMA
    - Contains pages capable of undergoing DMA
  - ZONE_NORMAL
    - Contains regularly mapped pages
  - ZONE_HIGHMEM
    - Contains pages not permanently mapped into the kernel address space

# struct zone (1)

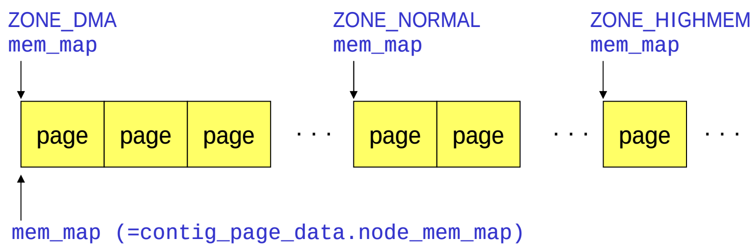
| Type | Name | Description | Notes |
|---|---|---|---|
| spinlock_t | lock | Spin lock protecting the descriptor | |
| unsigned long | free_pages | Number of free pages in the zone | |
| unsigned long | pages_min | Minimum number of pages of the zone that should remain free | Kswapd |
| unsigned long | pages_low, pages_high | Lower and upper threshold value for the zone's page balancing algorithm | Kswapd |
| spinlock_t | lru_lock | Spin lock protecting the following two linked lists | Page cache |
| struct list_head | active_list, inactive_list | Active and inactive lists (LRU lists) of pages in the zone | Page cache |
| unsigned long | nr_active, nr_inactive | The number of pages on the active_list and inactive_list | Page cache |

# struct zone (2)

| Type | Name | Description |
|---|---|---|
| struct free_area [] | free_area | Free area bitmaps used by the buddy allocator |
| wait_queue_head_t * | wait_table | A hash table of wait queues of processes waiting on a page to be freed |
| unsigned long | wait_table_size | The number of queues in the hash table |
| unsigned long | wait_table_bits | The number of bits in a page address from left to right being used as an index within the wait_table |
| struct per_cpu_pageset [] | pageset | Per CPU pageset for order-0 page allocation<br>(to avoid interrupt-safe spinlock on SMP system) |
| struct pglist_data * | zone_pgdat | Points to the descriptor of the parent node |
| struct page * | zone_mem_map | The first page in the global mem_map that this zone refers to |
| unsigned long | zone_start_pfn | The starting physical page frame number of the zone |
| char * | name | The string name of the zone: "DMA", "Normal" or "HighMem" |
| unsigned long | spanned_pages | Total size of physical page range, including holes |
| unsigned long | present_pages | Total number of physical pages in the zone |

# Pages

- To keep track of all physical pages, all physical pages are described by an array of `struct page` called `mem_map`




# struct page

| Type | Name | Description |
|---|---|---|
| `page_flag_t` | `flags` | The status of the page and mapping of the page to a zone |
| `atomic_t` | `_count` | The reference count to the page.<br>If it drops to zero, it may be freed |
| `unsigned long` | `private` | Mapping private opaque data: usually used for `buffer_heads` if PagePrivate set |
| `struct address_space *` | `mapping` | Points to the address space of a inode when files or devices are memory mapped. |
| `pgoff_t` | `index` | Our offset within mapping |
| `struct list_head` | `lru` | Linked to LRU lists of pages if the page is in page cache<br>Linked to `free_area` lists if the page is free and is managed by buddy allocator |

## Flags describing page status

| Flag name | Meaning |
|---|---|
| `PG_locked` | The page is involved in a disk I/O operation |
| `PG_error` | An I/O error occurred while transferring the page |
| `PG_referenced` | The page has been recently accessed for a disk I/O operation. This bit is used during page replacement for moving the page around the LRU lists. |
| `PG_uptodate` | When a page is read from disk without error, this bit will be set |
| `PG_dirty` | This indicates if a page needs to be flushed to disk. |
| `PG_lru` | The page is in the active or inactive page list |
| `PG_active` | The page is in the active page list |
| `PG_highmem` | The page frame belongs to the `ZONE_HIGHMEM` zone |
| `PG_reserved` | The page frame is reserved to kernel code or is unusable |

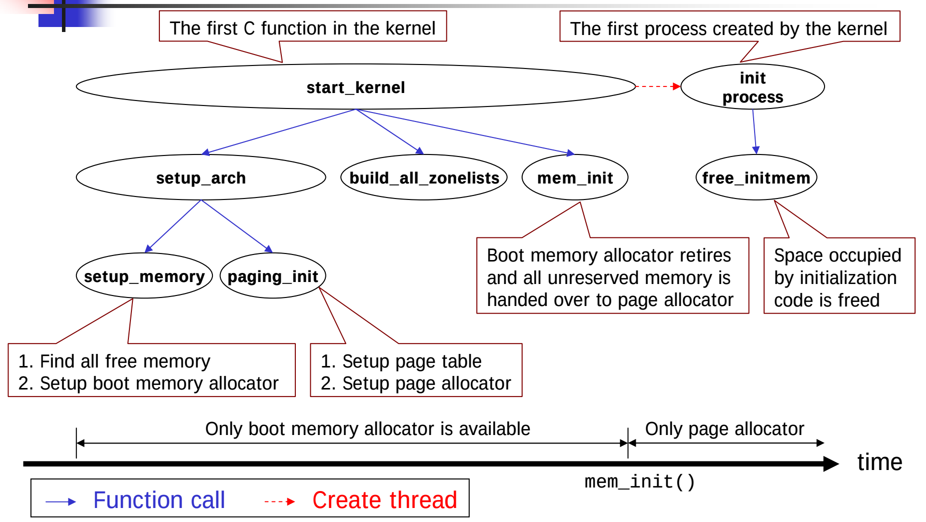
## Translating kernel virtual address

- Recall: memory in `ZONE_DMA` and `ZONE_NORMAL` is direct-mapped and all page frames are described by `mem_map` array
- Kernel virtual address -> physical address
- Physical address -> `struct page`
  - Use physical address as an index into the `mem_map` array

```
#define __pa(x)                   ((unsigned long)(x)-PAGE_OFFSET)
#define pfn_to_page(pfn)          (mem_map + (pfn))
#define virt_to_page(kaddr)       pfn_to_page(__pa(kaddr) >> PAGE_SHIFT)

static inline unsigned long virt_to_phys(volatile void * address)
{
        return __pa(address);
}
```

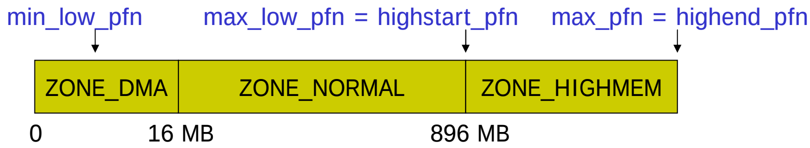
# Boot Memory Allocator

# The Flowchart of Initializing All Memory Allocators



# Determining the size of each zone

| Global variables | Description |
|---|---|
| `max_pfn` | The last page frame in the system. `find_max_pfn()` determine the value by reading through the e820 map from the BIOS |
| `min_low_pfn` | the lowest PFN available (the end of kernel image) |
| `max_low_pfn` | the end PFN of `ZONE_NORMAL`, determined by `find_max_low_pfn()` |
| `highstart_pfn`, `highend_pfn` | the start and end PFN of `ZONE_HIGHMEM` |

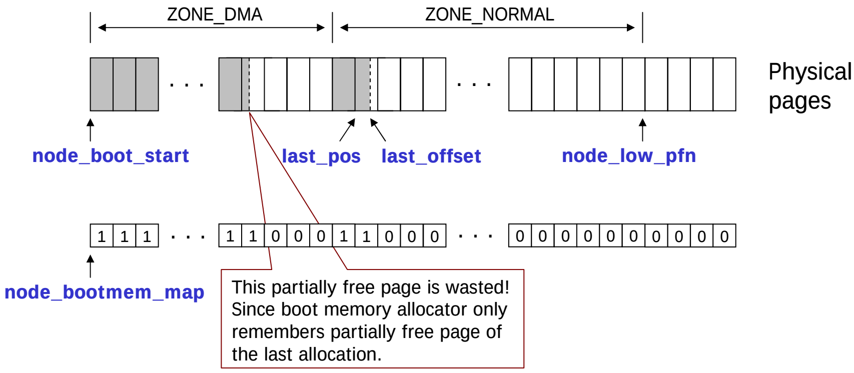



# Data Structures for Boot Memory Allocator

- A `struct bootmem_data` for each node of memory

| Type | Name | Description |
|---|---|---|
| `unsigned long` | `node_boot_start` | The starting physical address of the represented block |
| `unsigned long` | `node_low_pfn` | The end physical address in PFN (end of `ZONE_NORMAL`) |
| `void *` | `node_bootmem_map` | The location of the bitmap representing allocated or free pages with each bit |
| `unsigned long` | `last_offset` | The offset within the end page of the last allocation. If 0, the page used is full. |
| `unsigned long` | `last_pos` | The PFN of the end page of the last allocation. By using this with the `last_offset` field, a test can be made to see if allocations can be merged with the page used for the last allocation rather than using up a full new page. |
| `unsigned long` | `last_success` | The PFN of the start page of the last allocation. It is used to speed up the search of a block of free memory. |

# Example of boot memory allocation




Pages allocated are gray-colored and marked "1" in the bitmap

# `init_bootmem()` & `free_all_bootmem()`

`unsigned long init_bootmem(unsigned long start, unsigned long page)`

Initialized `contig_page_data.bdata` for page PFN between 0 and `page`. The beginning of usable memory is at the PFN `start` (for bootmem bitmap). The entire bitmap is initialized to 1

`unsigned long free_all_bootmem()`

Used at the boot allocator end of life. It cycles through all pages in the bitmap. For each unallocated page, the `PG_reserved` flag in its `struct page` is cleared, and the page is freed to the physical page allocator (`__free_pages()`) so that it can build its free lists. The pages for boot allocator bitmap are freed too

- Since there is no architecture independent way to detect holes in memory, `init_bootmem()` initializes the entire bitmap to 1. The bitmap will be updated by architecture dependent code later.

Accusys The RAID Architects

# reserve_bootmem() & free_bootmem()

```
void reserve_bootmem(unsigned long addr, unsigned long size)
```

Marks the pages between the address `addr` and `addr+size` reserved (allocated). Requests to partially reserve a page will result in the full page being reserved

```
void free_bootmem(unsigned long addr, unsigned long size)
```

Marks the pages between the address `addr` and `addr+size` as free. An important restriction is that only full pages may be freed. It is never recorded when a page is partially allocated, so, if only partially freed, the full page remains reserved

- Pages used by kernel code, bootmem bitmap are reserved by calling `reserve_bootmem()`
- `free_bootmem()` is used together with `alloc_bootmem()`

Accusys The RAID Architects

# alloc_bootmem()

```
void * alloc_bootmem(unsigned long size)
```

Allocates `size` number of bytes from `ZONE_NORMAL`. The allocation will be aligned to the L1 hardware cache to get the maximum benefit from the hardware cache.

```
void * alloc_bootmem_low(unsigned long size)
```

Allocates `size` number of bytes from `ZONE_DMA`. The allocation will be aligned to the L1 hardware cache.

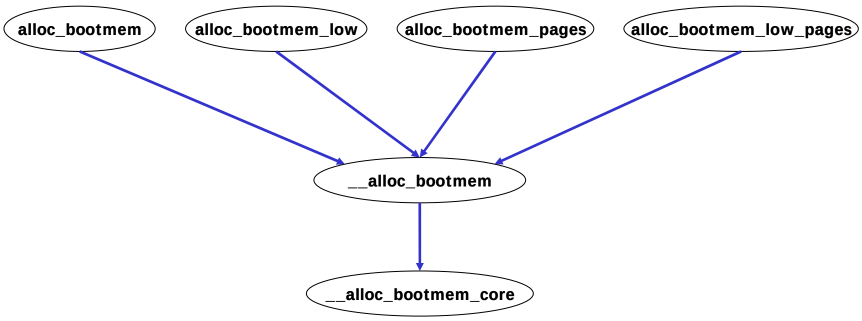
```
void * alloc_bootmem_pages(unsigned long size)
```

Allocates `size` number of bytes from `ZONE_NORMAL` aligned on a page size so that full pages will be returned to the caller.

```
void * alloc_bootmem_low_pages(unsigned long size)
```

Allocates `size` number of bytes from `ZONE_DMA` aligned on a page size so that full pages will be returned to the caller.

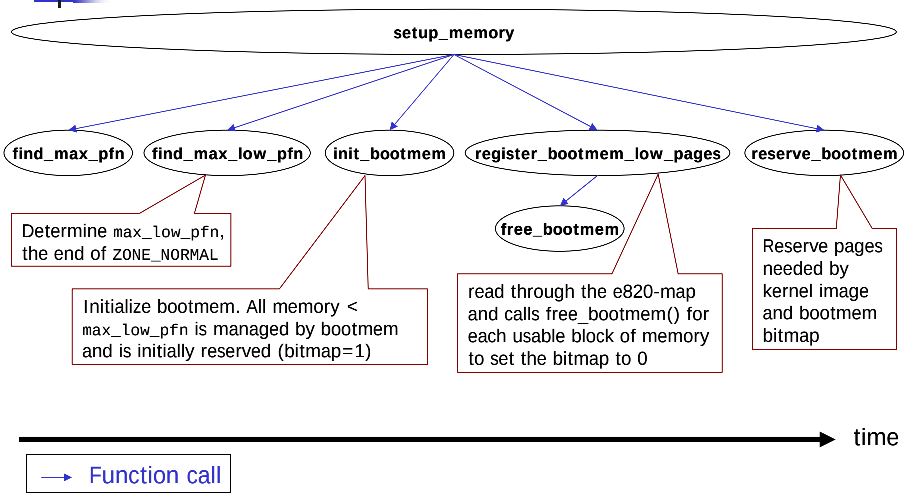
# Call Graph of `alloc_bootmem()`



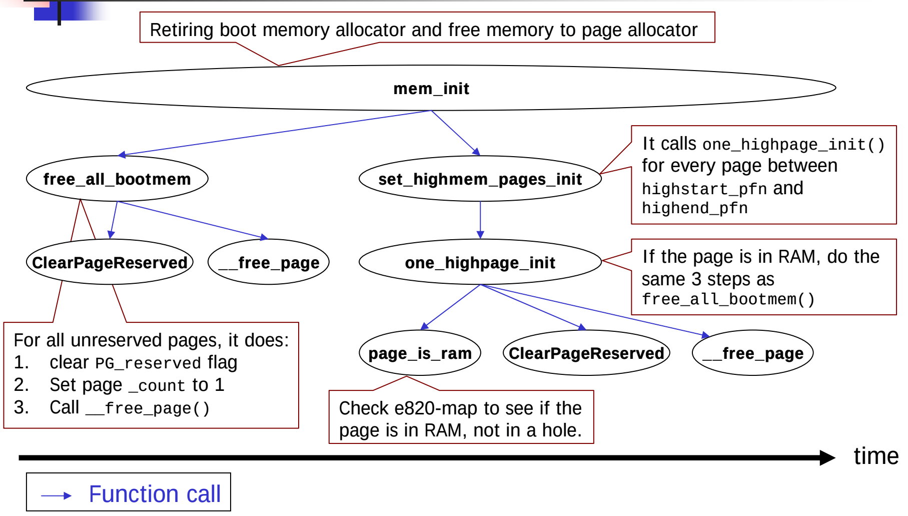

# The core function: `__alloc_bootmem_core()`

- It linearly scans memory starting from preferred address for a block of memory large enough to satisfy the allocation
  - Preferred address may be:
    1. the starting address of a zone or
    2. the address of last successful allocation
- When a satisfied memory block is found, this new allocation can be merged with the previous one if all of the following conditions hold:
  - The page used for the previous allocation (`bootmem_data.pos`) is adjacent to the page found for this allocation
  - The previous page has some free space in it (`bootmem_data.offset != 0`)
  - The alignment is less than `PAGE_SIZE`

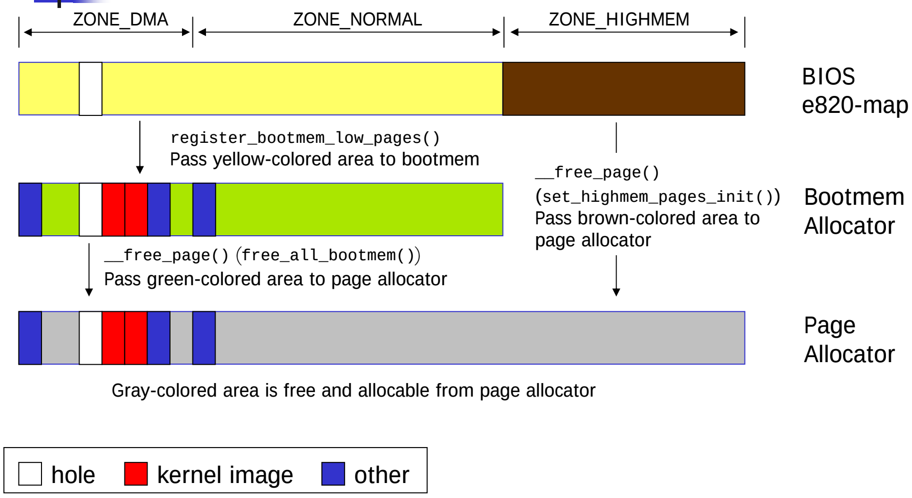
# The Flowchart of Initializing Boot Memory Allocator




# mem_init() - Retiring the Boot Memory Allocator



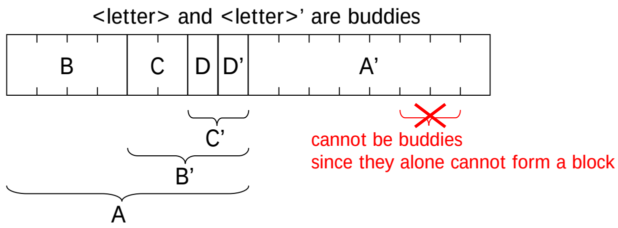
# From Boot Memory Allocator to Page Allocator




# Physical Page Allocator

# The Buddy System: the Algorithm of the Page Allocator

- An allocation scheme that combines free buffer coalescing with a power-of-two allocator
- Memory is split into blocks of pages where each block is a power of two number of pages.
- It create small blocks by repeatedly halving a large block and coalescing adjacent free blocks whenever possible.
- When a block is split, each half is called the buddy of the other.




# struct free_area

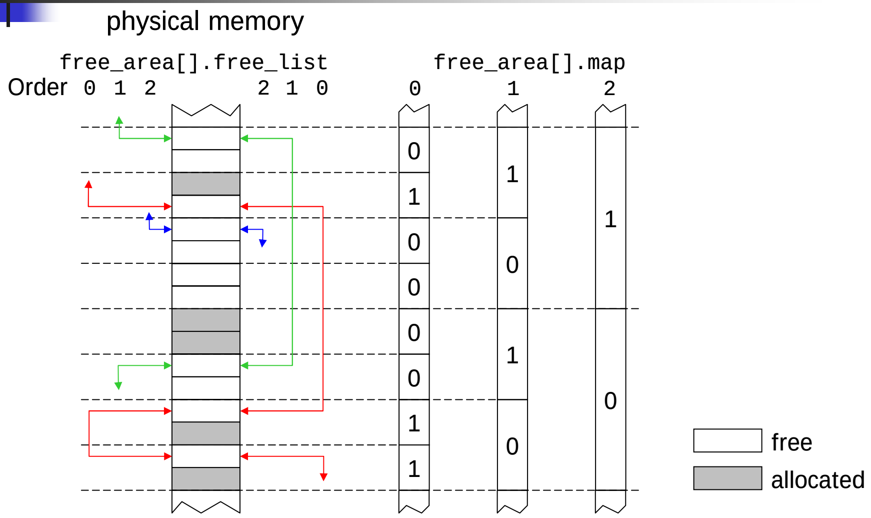
| Type | Name | Description |
|---|---|---|
| `struct list_head` | `free_list` | A linked list of free page blocks |
| `unsigned long *` | `map` | A bitmap representing the state of a pair of buddies |

- The exponent for the power of two-sized block is referred to as the *order*. An array of `free_area` of size MAX_ORDER is maintained for *orders* from `0` to `MAX_ORDER-1`
- `free_area[i].free_list` is a linked list of free blocks of $2^i$ page size
- `free_area[i].map` represents the allocation status of all pairs of buddies of $2^i$ page size. Each time a buddy is allocated or freed, the bit representing the pair of buddies is toggled so that the bit is 0 if the pair of pages are both free or both full and 1 if only one buddy is in use

## Think in another way about the meaning of maps in `free_area`

- Each bit in the `free_area[i].map` tells if a pair of buddies is in `free_area[i].free_list`
  - If a bit of the map is 0, the represented buddies are not in the free list. It may be both allocated, or both free and in the free list of higher order
  - If it is 1, exactly one of the buddies is in the free list. It may be reunified with its buddy when it is freed.

## Example of the contents of maps in `free_area`



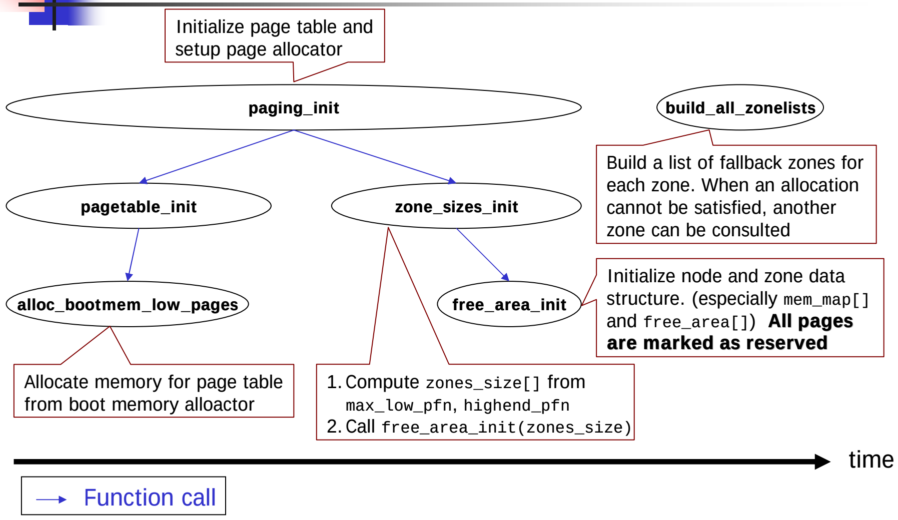
## Pseudo Code: Allocating Pages in `free_area`

1. Get a block out from the free list of the desired-`order` free area. If the area is empty, get it from `order+1` free area. Repeat this step until we get a block
2. Toggle the associated bit in the bitmap
3. If the block gotten is from a higher `order` free area, halve it, keep the first half, add the second half to `order-1` free list and toggle the associated bit in the bitmap. Repeat this step until we have a desired-size block.

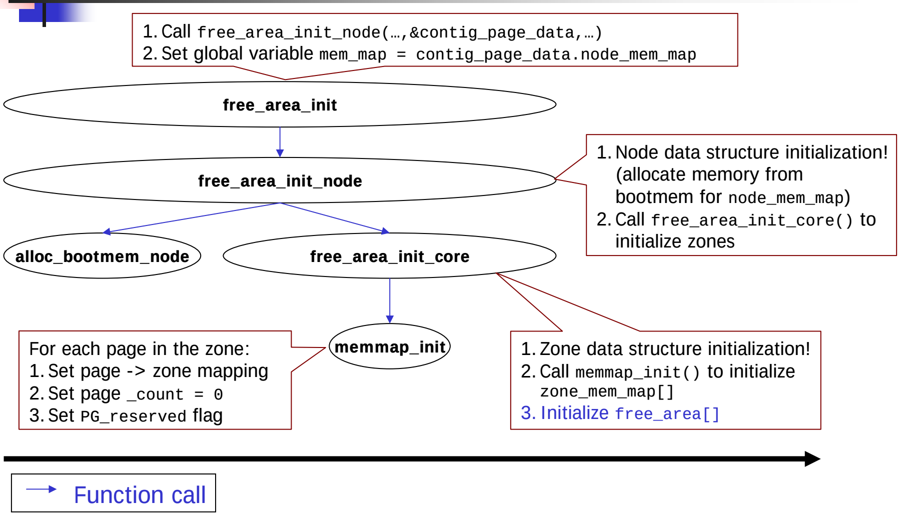
## Pseudo Code: Freeing Pages in `free_area`

1. For the block being freed, toggle the associated bit in the free area's bitmap. If the value of the bit before the toggle is 0 (i.e. the buddy is still allocated), go to step 3
2. Remove the buddy from the free list and merge it with the block. Then carry the resulting block to `order+1` free area and repeat step 1 and 2.
3. Put the block into the free list.

# The Flowchart of Initializing Physical Page Allocator




# The Flowchart of free_area_init()



# Initializing `free_area[]` for each zone

```
for (i = 0; ; i++) {
        unsigned long bitmap_size;

        INIT_LIST_HEAD(&zone->free_area[i].free_list);
        if (i == MAX_ORDER-1) {
                zone->free_area[i].map = NULL;
                break;
        }

        bitmap_size = (size-1) >> (i+4);
        bitmap_size = LONG_ALIGN(bitmap_size+1);
        zone->free_area[i].map =
          (unsigned long *) alloc_bootmem_node(pgdat, bitmap_size);
}
```

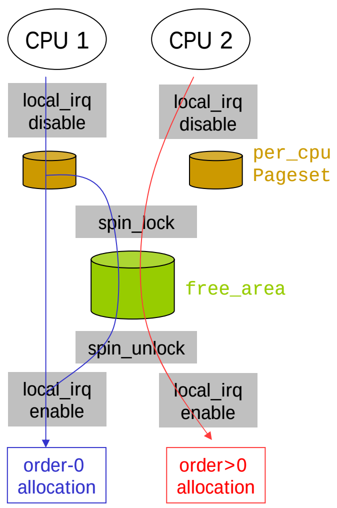
Since MAX_ORDER-1 is the highest order, blocks at this order are not merged. So free area map is not needed.

`size` = number of pages in a zone

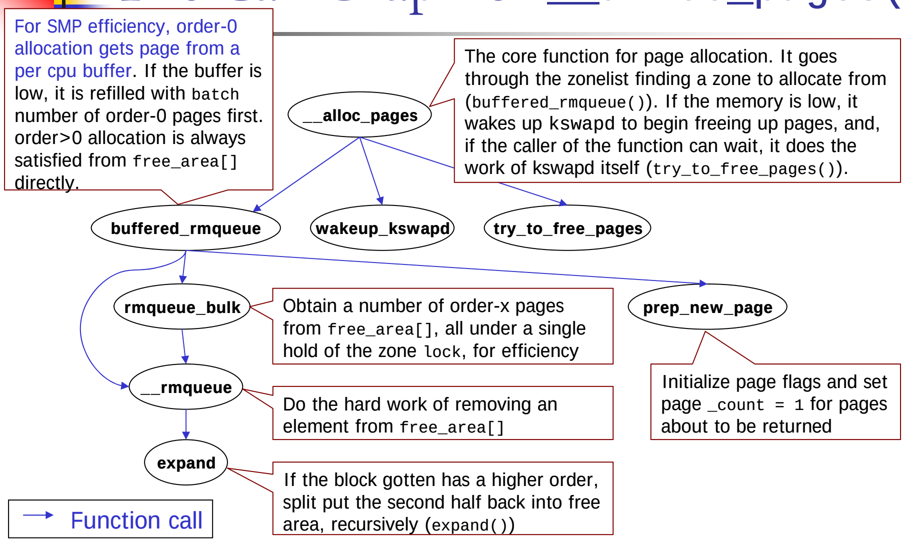
The calculation here (since Linux 2.4) is correct but hard to understand. It may be a little larger than the actual bytes needed. It should be ***`bitmap_size = LONG_ALIGN(((size >> (i+1)) + 7) >> 3)`***. The ***`i`*** is the order of the free area. The ***`+1`*** is because the buddy system uses a single bit to represent two blocks. ***`(size >> i+1)`*** is the number of bits in the bitmap. This value is shifted down by ***`3`*** to get the number of bytes, but we need to have a ***`+7`*** first to round up to byte size.

# Per-CPU Page Sets in Linux 2.6

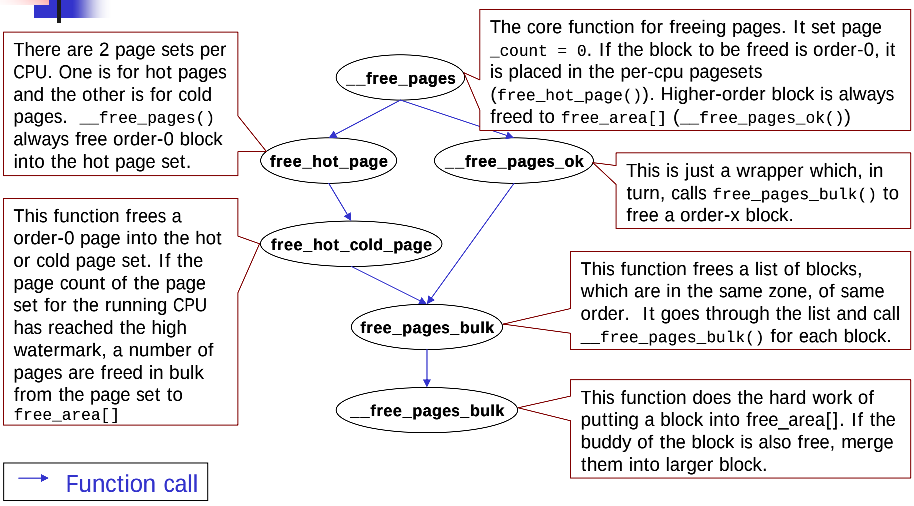
- Recall: `zone[].lock` spinlock protects the `free_area` from concurrent access
  - Lock contention between multiple CPUs may degrade the performance
- Linux 2.6 reduces the number of times acquiring this spinlock by introducing a per CPU page set (`per_cpu_pageset`)
  - It stores only order-0 pages since higher order allocations are rare
  - Order-0 block allocation requires no spinlock being held. But if the page set is low, a number of pages will be allocated in bulk with the spinlock held
  - Side effect: splits and coalescing of blocks for order-0 allocation are delayed



# The Call Graph of `__alloc_pages()`




# The Call Graph of `__free_pages()`



# Physical Pages Allocation API

| | |
|---|---|
| `struct page * alloc_page(unsigned int gfp_mask)` | |
| Allocates a single page and return a pointer to its `page` structure. | |
| `struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)` | |
| Allocates $2^{order}$ pages and return a pointer to the first page's `page` structure. | |
| `unsigned long __get_free_page(unsigned int gfp_mask)` | |
| Allocates a single page and return a pointer to its virtual address. | |
| `unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)` | |
| Allocates $2^{order}$ pages and return a pointer to the first page's virtual address. | |
| `unsigned long __get_dma_pages(unsigned int gfp_mask, unsigned int order)` | |
| Allocates $2^{order}$ pages from `ZONE_DMA` and return a pointer to the first page's virtual address. | |
| `unsigned long get_zeroed_page(unsigned int gfp_mask)` | |
| Allocates a single page, zero its contents, and return a pointer to its virtual address. | |

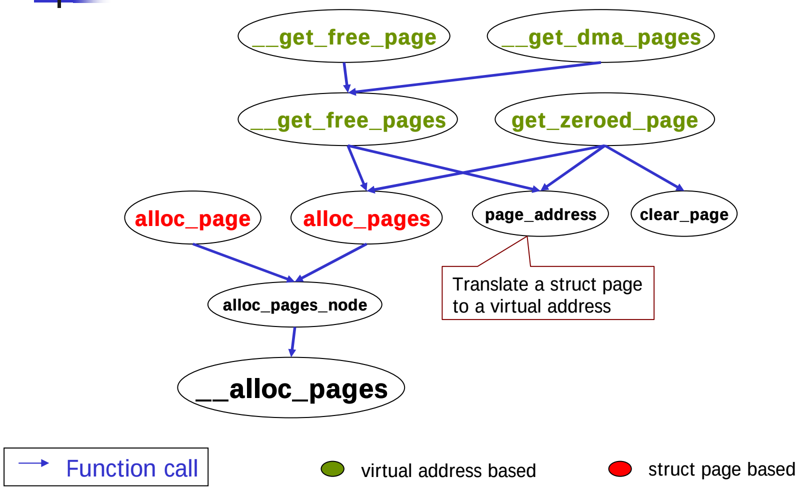
# Physical Pages Free API

| | |
|---|---|
| `void __free_page(struct page *page)` | |
| Frees a single page. | |
| `void __free_pages(struct page *page, unsigned int order)` | |
| Frees $2^{order}$ pages from the given page. | |
| `void free_page(unsigned long addr)` | |
| Frees a single page from the given virtual address. | |
| `void free_pages(unsigned long addr, unsigned int order)` | |
| Frees $2^{order}$ pages from the given virtual address. | |

- There are only two core function for page allocation and free, but two namespaces to them.
  - Pointer to `struct page`: `alloc_page*()` and `__free_page*()`
  - Virtual address: `*get*page*()` and `free_page*()`

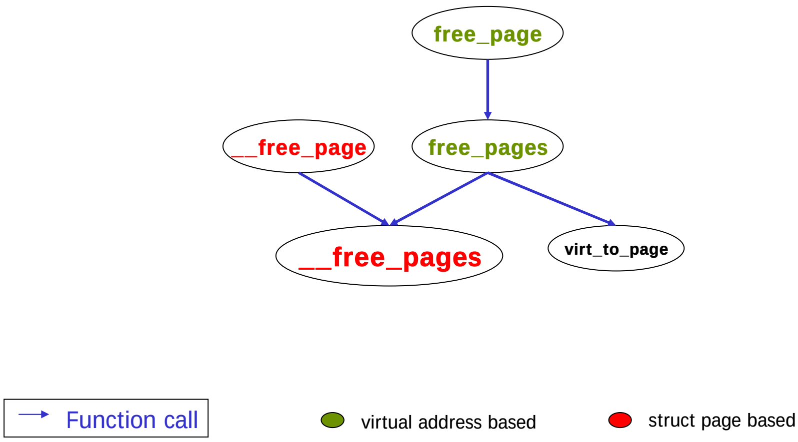
# The Call Graph of Physical Pages Allocation API

Accusys
The RAID Architects




# The Call Graph of Physical Pages Free API

Accusys
The RAID Architects



Accusys

# Get Free Page (gfp_mask) Flags

- 3 categories of flags
  - Zone modifiers
    - Specify from where to allocate memory
  - Action modifiers
    - Specify how the kernel is supposed to allocate the requested memory
  - Type flags
    - Specify a combination of action and zone modifiers as needed by a certain type of memory allocation
- Don’t use zone or action modifiers directly. Use type flags if there are suitable type flags.

Accusys

# gfp_mask: Zone Modifiers

- The kernel allocates memory from ZONE_NORMAL if none of the zone modifiers are specified
- If the memory is low, the allocations can fall back on another zone according to the fallback zonelists
- The fallback order
  - ZONE_HIGHMEM->ZONE_NORMAL->ZONE_DMA
- Don’t use __GFP_HIGHMEM with *get*page*() or kmalloc()
  - They may return an invalid virtual address since the allocated pages are not mapped in the kernel’s virtual address space

| Flags | Description |
|---|---|
| __GFP_DMA | Allocate only from ZONE_DMA |
| __GFP_HIGHMEM | Allocate from ZONE_HIGHMEM or ZONE_NORMAL |

## gfp_mask: Action Modifiers

| Flags | Description |
|---|---|
| __GFP_WAIT | The allocator can sleep |
| __GFP_HIGH | The allocator can access emergency pools of memory |
| __GFP_IO | The allocator can start disk I/O |
| __GFP_FS | The allocator can start filesystem I/O |
| __GFP_COLD | The allocator should use cache cold pages |
| __GFP_NOWARN | The allocator will not print failure warnings |
| __GFP_REPEAT | The allocator will repeat the allocation if it fails |
| __GFP_NOFAIL | The allocator will indefinitely repeat the allocation |
| __GFP_NORETRY | The allocator will never retry if the allocation fails |
| __GFP_NOGROW | Used internally by the slab layer |

## gfp_mask: Type Flags

| Flags | Description (AC = Allocator) | Modifier flags |
|---|---|---|
| GFP_ATOMIC | AC is high priority and must not sleep. This flag is used in interrupt handlers, bottom halves, and other situations where you cannot sleep | __GFP_HIGH |
| GFP_NOIO | AC may block, but won't start disk I/O. This flag is used in block I/O code when you cannot cause more disk I/O | __GFP_WAIT |
| GFP_NOFS | AC may block and start disk I/O, but won't start filesystem I/O. This flag is used in filesystem code when you cannot start another filesystem operation | (__GFP_WAIT \| __GFP_IO) |
| GFP_KERNEL | This is for normal allocation. AC may block. This flag is used in process context code when it is safe to sleep | (__GFP_WAIT \| __GFP_IO \| __GFP_FS) |
| GFP_USER | This is for normal allocation. AC may block. This flag is used to allocate memory for user-space processes. | (__GFP_WAIT \| __GFP_IO \| __GFP_FS) |
| GFP_HIGHUSER | AC may block. This flag is used to allocate memory from ZONE_HIGHMEM for user-space processes. | (__GFP_WAIT \| __GFP_IO \| __GFP_FS \|__GFP_HIGHMEM) |
| GFP_DMA | Device drivers that need DMA-able memory use this flag, usually in combination with one of the above. | __GFP_DMA |

# Reference

- Understanding the Linux Virtual Memory Manager, Mel Gorman, Prentice Hall, 2004
- Understanding the Linux Kernel, Bovet & Cesati, O'REILLY, 2002
- Linux Kernel Development, Robert Love, Sams Publishing, 2003

# Memory Allocators

Hao-Ran Liu

# kmalloc() & kfree()

- Features
  - Based on slab allocator and page allocator
  - Allocate physically continuous memory
  - Allow request size smaller than a page
  - Similar to malloc() in user space
- Prototype
  - void * kmalloc(size_t size, int flags)
  - void kfree(const void *ptr)

```
struct dog * ptr;
ptr = kmalloc (sizeof(struct dog), GFP_KERNEL);
If (!ptr)
        /* error handling */
…
kfree(ptr);
```

# Which flag to use when

| Situation | Solution |
| --- | --- |
| process context | GFP_KERNEL |
| interrupt handler, softirq, tasklet | GFP_ATOMIC |
| need DMA-able memory, can sleep | GFP_DMA \| GFP_KERNEL |
| need DMA-able memory, cannot sleep | GFP_DMA \| GFP_ATOMIC |

# vmalloc() & vfree()

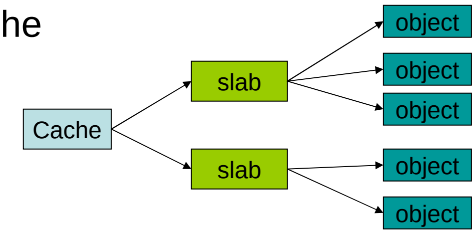
- Features
  - Based on page allocator & kmap()
  - Allocate virtually continuous memory
  - Allow allocation of a large-size chunk of memory which kmalloc() cannot
- Performance Overhead
  - Page table mapping and TLB thrashing
- Prototype
  - void *vmalloc(unsigned long size)
  - void vfree(void *addr)

# Slab Allocator

- Features
  - Based on page allocator
  - Allocation of smaller-than-a-page chunk of memory efficiently
  - Create a cache for frequently used data structures; this avoids internal fragmentation
  - Caching objects reduces call to buddy allocators; this increases hardware cache performance (buddy allocator footprint)
  - Objects are colored to prevent multiple objects mapping to the same cache lines
  - kmalloc() creates a set of caches storing power-of-2 objects (from 32 to 131072 bytes)

# Slab Allocator

- Term definition
  - Cache: A group of a kind of objects; e.g. you create a cache for caching task_struct objects
  - Slab: composed of one or more physically continuous pages
  - Object: the data structure you designated for a cache



# Slab Allocator interface

- Create a cache

```
kmem_cache_t kmem_cache_create(const char *name, size_t size,
                    size_t align, unsigned long flags,
                    void (*ctor) (void *, kmem_cache_t *, unsigned long),
                    void (*dtor) (void *, kmem_cache_t *, unsigned long))
```

- Arguments
  – name: name of the cache shown in /proc/slabinfo
  – size: size of each object in the cache
  – flags: see next slide
  – ctor: constructor function to call when new objects are added to the cache
  – dtor: destructor function to call when objects are removed from the cache
- ctor and dtor are usually NULL
- This function returns a pointer to the created cache

# Slab allocator flags

| flags | description |
|---|---|
| SLAB_NO_REAP | slab layer will reap objects in the cache when memory is low |
| SLAB_HWCACHE_ALIGN | align each object within a slab to different cache lines. This improves performance at the cost of memory consumption |
| SLAB_MUST_HWCACHE_ALIGN | For debugging purpose |
| SLAB_POISON | Fill memory with 0xa5a5a5a5. Useful for catching access to uninitialized memory |
| SLAB_RED_ZONE | Insert “red zone” around the allocated memory to detect buffer overruns |
| SLAB_PANIC | Make slab panic() if allocation fails |
| SLAB_CACHE_DMA | Memory allocated from the slab must come from ZONE_DMA |

# Slab Allocator interface

- Destroy a cache

```
kmem_cache_t kmem_cache_destroy(kmem_cache_t * cachep)
```

- This function can sleep since it must ensure that
  – All objects allocated from this cache are freed
  – No one access the cache during the execution of kmem_cache_destroy()

# Slab allocator interface

- Allocate from a cache

```
void *kmem_cache_alloc(kmem_cache_t *cachep, int flags)
```

- Returning an object to its cache

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
```

# Slab allocator example

- Network buffer allocator and free

```
void __init skb_init(void) {
        skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
                        sizeof(struct sk_buff), 0,
                        SLAB_HWCACHE_ALIGN, NULL, NULL);
        if (!skbuff_head_cache) panic("cannot create skbuff cache");
}
struct sk_buff *alloc_skb(unsigned int size, unsigned int __nocast gfp_mask) {
        struct sk_buff *skb;
        skb = kmem_cache_alloc(skbuff_head_cache, gfp_mask & ~__GFP_DMA);
        if (!skb)   goto out;
        …
}
void kfree_skbmem(struct sk_buff *skb) {
        skb_release_data(skb);
        kmem_cache_free(skbuff_head_cache, skb);
}
```

# Kernel stack

- Kernel stack is usually 8k for 32bit CPU
  - Interrupts share the stack of interrupted process
- In 2.6 kernel (optional)
  - The stack size for a process is shrinked to 4k
    - Reduce memory consumption on system with large number of processes since kernel stack is not swappable
  - Interrupts has its own interrupt stack (4k)

Accusys

# Permanent high memory mappings

- Background
  - Memory allocated from ZONE_HIGHMEM does not have a valid logical address
  - Highmem pages allocated from page allocator must be mapped into kernel address space first
- Prototype
  - void *kmap(struct page *page)
  - Void kunmap(struct page *page)

Accusys

# Temporary high memory mapping

- Why temporary mapping?
  - Permanent mapping may be unavailable;
  - When a mapping must be created but the current context is unable to sleep
- The kernel provide a range of reserved address space for temporary mapping
  - But you must not sleep while you hold the mapping
- Prototype
  - kmap_atomic(struct page *page, enum km_type type)
  - kunmap_atomic(void *kvaddr, enum km_type type)

# Temporary high memory mapping

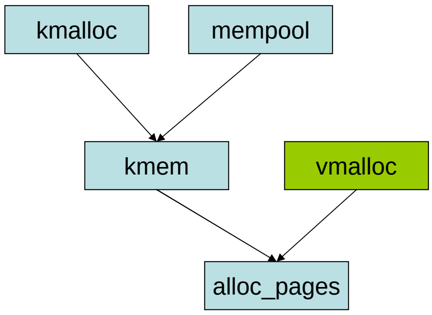
- Only 14 pages address space available for temporary mapping

```
enum km_type {
        KM_BOUNCE_READ,
        KM_SKB_SUNRPC_DATA,
        KM_SKB_DATA_SOFTIRQ,
        KM_USER0,
        KM_USER1,
        KM_BIO_SRC_IRQ,
        KM_BIO_DST_IRQ,
        KM_PTE0,
        KM_PTE1,
        KM_IRQ0,
        KM_IRQ1,
        KM_SOFTIRQ0,
        KM_SOFTIRQ1,
        KM_TYPE_NR
};
```

# Which allocator to use?

| | kmalloc | vmalloc | kmem | mempool | alloc_pages |
|---|---|---|---|---|---|
| **Contiguous** physical/ logical | P | L | P | P | P |
| **Unit size** | byte | byte (page) | object | object | page |
| **Max alloc size** | 128KB | < 128MB | 128KB | 128KB | 8MB |
| **Fragmentation** | internal | external | external | external | external |
| **Highmem?** | No | Yes | No | No | Yes |
| **Cache utilization** | | | cacheline, per cpu arraycache | | per cpu hot/cold pages |
| **SMP sync overhead** | No | Yes | No | Yes | No |

# The hierarchy of memory allocators




# References


- Linux Kernel Development, 2nd edition, Robert Love, 2005
- Linux kernel 2.6.10 source

# Linux Char Drivers

Hao-Ran Liu

# References

- This tutorial use examples and material from Linux Device Drivers, 3rd edition
- All code fragments are from:
  - LDD3 source code: http://examples.oreilly.com/linuxdrive3/examples.tar.gz
  - Linux kernel source 2.6.13

## Example use in this tutorial

- **scull** is a char driver that acts on a memory area as though it were a device
- **scull** device is global and persistent by design
  - Data contained within the device is shared by all file descriptors that opened it
  - If the device is closed, data isn't lost
- It is hardware-independent; anyone can compile and run **scull**

## Device files

- In UNIX world, devices are treated like normal files
- Try "ls –l /dev", you will see a list like this:

Block device

char device

```
brw-rw----  1 root disk 3,  0 2005-03-20 03:36 hda
brw-rw----  1 root disk 3, 64 2005-03-20 03:36 hdb
crw-r-----  1 root kmem 1,  2 2005-03-20 03:36 kmem
crw-r-----  1 root kmem 1,  1 2005-03-20 03:36 mem
crw-rw-rw-  1 root root 1,  3 2005-03-20 03:36 null
crw-rw-rw-  1 root root 1,  8 2005-03-20 03:36 random
crw-------  1 uucp uucp 4, 64 2006-03-02 18:07 ttyS0
crw-rw----  1 root uucp 4, 65 2006-01-19 18:58 ttyS1
crw-rw-rw-  1 root root 1,  5 2005-03-20 03:36 zero
```

major minor

# Major and minor numbers

- Traditionally, the major number identifies the driver associated with the device file, and the minor number tells which device is being referred to
- In the kernel, device number is represented by a 32-bit integer (`dev_t`)
  - 12 bits for major number
  - 20 bits for minor number
- Do not assume the structure of dev_t
  - Use kernel macro to encode or decode these numbers

```
#define MINORBITS        20
#define MINORMASK        ((1U << MINORBITS) - 1)

#define MAJOR(dev)       ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)       ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)     (((ma) << MINORBITS) | (mi))
```

# Allocating and freeing device numbers

- Static allocation

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

`first` is the beginning device number of the range to allocate. The minor number portion of first is often 0. `count` is the total number of device numbers. `name` is the name of the device; it will appear in `/proc/devices` and sysfs.

- Dynamic allocation

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
                        unsigned int count, char *name);
```

`dev` is an output-only parameter that will hold the first number in your allocated range when the function returns. `firstminor` should be the requested first minor number to use; it is usually 0. `count` and `name` is the same as those in `register_chrdev_region`

- Freeing device numbers

```
Void unregister_chrdev_region(dev_t first, unsigned int count);
```

## Dynamic allocation of major numbers

- Check this file Documentation/devices.txt for all device numbers statically allocated
- New drivers should use dynamic allocation to obtain device numbers, rather than choosing a number randomly from the ones that are currently free
  - The device file in /dev directory should be recreated every time when the driver is loaded into the kernel

## /proc/devices

- A list of registered major numbers

```
josephl@moon:~> cat /proc/devices
Character devices:
  1 mem
  2 pty
  4 ttyS
  6 lp
 10 misc
 13 input
 21 sg
180 usb

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  8 sd
  9 md
 22 ide1
```

## **scull** to register a major number

```
int scull_major = 0, scull_minor = 0;

// major number can be specified at module load time
module_param(scull_major, int, S_IRUGO);

int scull_init_module(void) {
    int result, i; dev_t dev = 0;

    if (scull_major) {
        dev = MKDEV(scull_major, scull_minor);
        result = register_chrdev_region(dev, scull_nr_devs, "scull");
    } else {
        result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs, "scull");
        scull_major = MAJOR(dev);
    }
    if (result < 0) {
        printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
        return result;
    }

    ...
}
```

## File operations

- A structure containing pointers to device driver's functions, which implements a set of system calls for each open file

```
struct file_operations {
        // pointer to the module that owns the structure, usually
        // initialized to THIS_MODULE
        struct module *owner;

        // change the current read/write position in a file. The return
        // value is the new position
        loff_t (*llseek) (struct file *, loff_t, int);

        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t,
                          loff_t *);
        ...
};
```

# File operations

```
struct file_operations {
        ...

        // return a bit mask indicating wheter nonblocking reads or
        // writes are possible
        unsigned int (*poll) (struct file *, struct poll_table_struct *);

        // offer a way to issue device-specific commands
        int (*ioctl) (struct inode *, struct file *, unsigned int,
                      unsigned long);

        // map device memory into a process's address space
        int (*mmap) (struct file *, struct vm_area_struct *);

        // called when the device file is opened. If this entry is NULL,
        // opening the device always succees
        int (*open) (struct inode *, struct file *);

        ...
};
```

# File operations

```
struct file_operations {
        ...

        // invoked when a process closes its copy of a file descriptor
        // for a device
        int (*flush) (struct file *);

        // invoked when the file structure is being released. Whenever a
        // file structure is shared, release won't be invoked until all
        // copies are closed. Like open, release can be NULL
        int (*release) (struct inode *, struct file *);

        // the back end of fsync system call, which a user calls to flush
        // any pending data
        int (*fsync) (struct file *, struct dentry *, int datasync);
};
```

Accusys
The RAID Architects

## scull file operations

- Tagged structure initialization syntax is preferred because it allows the reordering of structure members (to put frequently accessed members in the same hardware cache line)

```
struct file_operations scull_fops = {
        .owner =    THIS_MODULE,
        .llseek =   scull_llseek,
        .read =     scull_read,
        .write =    scull_write,
        .ioctl =    scull_ioctl,
        .open =     scull_open,
        .release =  scull_release,
};
```

Accusys
The RAID Architects

## The file structure

- Represents an open file in the kernel and is passed to any function that operates on the file

```
// only most important fields of struct file are shown here
struct file {

        // identifies the file as either readable or writable (or both),
        // by means of the bits FMODE_READ and FMODE_WRITE
        mode_t                    f_mode;

        // The current reading or writing position. It is a 64-bit value.
        // read and write should update a position using the pointer they
        // receive as the last argument instead of acting on filp->f_pos
        // directly
        loff_t                    f_pos;

        ...
};
```

# The file structure

```
struct file {
        atomic_t                 f_count; // file object's usage count

        // file flags, such as O_RDONLY, O_NONBLOCK and O_SYNC.
        // Read/write permission should be checked using f_mode rather
        // than f_flags
        unsigned int             f_flags;

        // The operations assoiated with the file. The kernel assigns the
        // pointer as part of its implementation of open. You can change
        // the file operations associated with your file, and the new
        // methods will be effective after you return to the caller.
        struct file_operations   *f_op;

        // Used by drivers to keep state information across system calls
        void                     *private_data;

        // the dentry structure associated with the file.
        struct dentry            *f_dentry;

        ...
};
```

# The inode structure

- Used by the kernel internally to represent files
  - It is different from the file structure that represents an open file descriptor
  - There can be numerous file structures representing multiple open descriptors on a single file

# The inode structure

```
struct inode {
        unsigned long               i_ino;   // inode number
        atomic_t                    i_count; // reference count
        umode_t                     i_mode;  // access permissions
        unsigned int                i_nlink; // number of hard links
        uid_t                       i_uid;   // user id of owner
        gid_t                       i_gid;   // group id of owner

        // for inodes that represents device files, this field contains
        // the actual device number.
        dev_t                       i_rdev;
        loff_t                      i_size;  // file size in bytes
        struct timespec             i_atime; // last access time
        struct timespec             i_mtime; // last modification time
        struct timespec             i_ctime; // file creation time
        struct inode_operations     *i_op;   // inode operations
        struct block_device         *i_bdev; // block device structure
        struct cdev                 *i_cdev; // char device structure
};
```

# Obtain major and minor number from an inode

- These macros should be used instead of manipulating `i_rdev` directly

```
static inline unsigned iminor(struct inode *inode)
{
        return MINOR(inode->i_rdev);
}

static inline unsigned imajor(struct inode *inode)
{
        return MAJOR(inode->i_rdev);
}
```

## Char device registration

- Before the kernel can invoke your device's operations, you must allocate and register one or more of the `struct cdev`, which represents char devices in the kernel

```
struct cdev *cdev_alloc(void);
```

Allocate and return a `struct cdev`.

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Initialize the given `struct cdev` and associate the `fops` with the it.

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

Register the given `cdev` to the kernel. `dev` is the first device number to which this device responds, and `count` is the number of device numbers that should be associated with the device.

```
void cdev_del(struct cdev *p);
```

Remove a char device from the kernel

## Char device registration

- Registration in the old way

```
int register_chrdev(unsigned int major, const char *name,
                    struct file_operations *fops);
```

This function is composed of `register_chrdev_region()`, `cdev_alloc()` and `cdev_add()`. It registers minor numbers 0-255 for the given `major`, and sets up a `cdev` structure.

# Device registration in **scull**

- **scull** represents each device with a structure of type `struct scull_dev`

```
struct scull_dev {
        struct scull_qset *data;  /* Pointer to first quantum set */
        int quantum;              /* the current quantum size */
        int qset;                 /* the current array size */
        unsigned long size;       /* amount of data stored here */
        unsigned int access_key;  /* used by sculluid and scullpriv */
        struct semaphore sem;     /* mutual exclusion semaphore     */
        struct cdev cdev;         /* Char device structure          */
};
```

# Device registration in **scull**

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
        int err, devno = MKDEV(scull_major, scull_minor + index);

        cdev_init(&dev->cdev, &scull_fops);
        dev->cdev.owner = THIS_MODULE;
        dev->cdev.ops = &scull_fops;
        err = cdev_add(&dev->cdev, devno, 1);
        /* Fail gracefully if need be */
        if (err)
                printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

## The open method

- **open** should perform the following tasks:
  - Check for device-specific errors (such as device-not-ready or similar hardware problems)
  - Initialize the device if it is being opened for the first time
  - Update the `f_op` pointer, if necessary
  - Allocate and fill any data structure to be put in `filp->private_data`

## scull's open method

- If you register your device with `register_chrdev()`, you must use minor number stored in the inode structure to identify the device being opened

```
int scull_open(struct inode *inode, struct file *filp)
{
        struct scull_dev *dev; /* device information */

        dev = container_of(inode->i_cdev, struct scull_dev, cdev);
        filp->private_data = dev; /* for other methods */

        /* now trim to 0 the length of the device if open was write-only */
        if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
                if (down_interruptible(&dev->sem))
                        return -ERESTARTSYS;
                scull_trim(dev); /* ignore errors */
                up(&dev->sem);
        }
        return 0;          /* success */
}
```

# Macro: `container_of()`

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)

/* container_of - cast a member of a structure out to the containing
 * structure
 * @ptr: the pointer to the member.
 * @type:       the type of the container struct this is embedded in.
 * @member:     the name of the member within the struct.
 */
#define container_of(ptr, type, member) ({                  \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);\
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

# The release method

- The same `file` structure may be used by many file descriptors. `fork()` and `dup()` increment the `f_count`, and `close()` decrements it. Only when the counter reaches zero, the release method is executed.
- **release** should perform the following tasks:
  - Deallocate anything that open allocated in `filp->private_data`
  - Shut down the device on last close
- **scull**'s release method

```
int scull_release(struct inode *inode, struct file *filp)
{
        return 0;
}
```

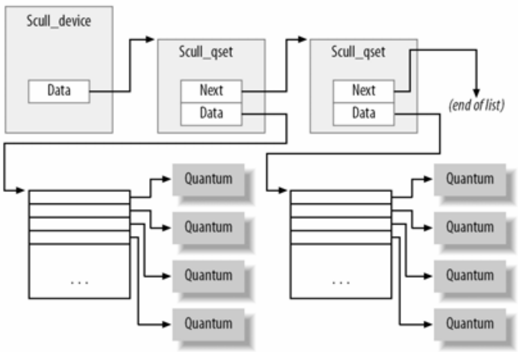
# scull's memory usage

- The region of memory used by scull is variable in length; the more you write, the more it grows
  - If you write more data than available memory, Kernel will invoke OOM killer to kill applications
    - Eg. `cp /dev/zero /dev/scull0`
- Each quantum is 4000 bytes allocated from kmalloc(), and a quantum set has an array of 1000 pointers to quantum.

# scull's memory usage

- Structure of quantum set

```
struct scull_qset {
        void **data;
        struct scull_qset *next;
};
```



# scull's memory usage

- `scull_trim()` is in charge of freeing the whole data area. It is called by **open** and module cleanup function

```
int scull_trim(struct scull_dev *dev) {
    struct scull_qset *next, *dptr; int i;
    int qset = dev->qset;   /* "dev" is not-null */

    for (dptr = dev->data; dptr; dptr = next) { /* all the list items */
        if (dptr->data) {
            for (i = 0; i < qset; i++) kfree(dptr->data[i]);
            kfree(dptr->data);
            dptr->data = NULL;
         }
         next = dptr->next;
         kfree(dptr);
    }
    dev->size = 0; dev->data = NULL;
    dev->quantum = scull_quantum; dev->qset = scull_qset;
    return 0;
}
```

# The read and write method

```
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);
// count: the size of requested data transfer
// buff:  pointer to user space buffer
// offp:  pointer to a "long offset type" object that indicates the
//        file position the user is accessing. Drivers should update
//        the file position at *offp after successful completion of the
//        system call
```

- `buff` cannot be dereferenced directly for 3 reasons
  - The address stored in `buff` may not be mapped in the kernel mode
  - the page referenced by `buff` may be paged out
  - The address is supplied by a user program, which could be buggy or malicious

## Accessing user space buffer

- These functions might sleep
  - Any function that accesses user space must be reentrant

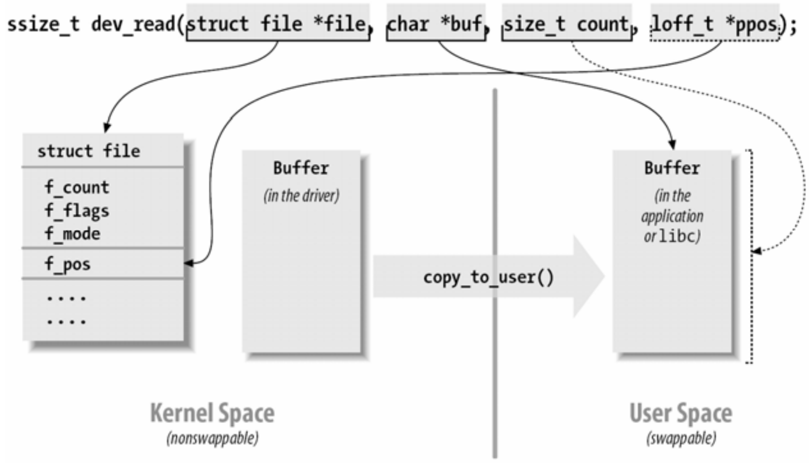
```
// Check if user pointer is valid and return the number of bytes
// not copied. On success, return value will be zero
unsigned long copy_to_user(void __user *to, const void *from,
                                  unsigned long count);
unsigned long copy_from_user(void *to, const void __user *from,
                                  unsigned long count);

// the underline version do not check if user pointer is valid or not
unsigned long __copy_from_user(void *to, const void __user *from,
                                  unsigned long n);
unsigned long __copy_to_user(void __user *to, const void *from,
                                  unsigned long n);

// check if a user space pointer is valid by ensuring it does not point to
// kernel space memory. type should be VERIFY_READ or VERIFY_WRITE. For read
// and write, use VERIFY_WRITE. return true if the memory may be valid, false
// if it is definitely invalid
int access_ok(int type, const void *addr, unsigned long size);
```

## Accessing user space buffer

```
// write/read a simple value(x) into/from user space(ptr). ptr must
// have pointer-to-simple-value type. return 0 on success or –EFAULT on error
put_user(x,ptr);
get_user(x, ptr);

// like above function but user pointer is not verified
__put_user(x, ptr);
__get_user(x, ptr);
```

Accusys

# Typical use of read in the driver




Accusys

# The return value for read and write

- Return value
  - Positive if a number of bytes has been transferred (may be smaller than `count`)
  - 0 (end-of-file was reached for read or nothing was written)
  - Negative if there was an error
- Drivers are free to complete only a portion of data transfer and return a positive value smaller than `count`
  - Network, pipe, fifo, terminal device often do this
  - C library will reissue the system call until completion of the requested data transfer

Accusys

The RAID Architects

## scull's read method

```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                loff_t *f_pos)
{
        struct scull_dev *dev = filp->private_data;
        struct scull_qset *dptr;  /* the first listitem */
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset; /* how many bytes in the listitem */
        int item, s_pos, q_pos, rest;
        ssize_t retval = 0;

        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;
        if (*f_pos >= dev->size)
                goto out;
        if (*f_pos + count > dev->size)
                count = dev->size - *f_pos;

        /* find listitem, qset index, and offset in the quantum */
        item = (long)*f_pos / itemsize;
        rest = (long)*f_pos % itemsize;
        s_pos = rest / quantum; q_pos = rest % quantum;
```

Accusys

The RAID Architects

## scull's read method

```
        /* follow the list up to the right position (defined elsewhere) */
        dptr = scull_follow(dev, item);

        if (dptr == NULL || !dptr->data || ! dptr->data[s_pos])
                goto out; /* don't fill holes */

        /* read only up to the end of this quantum */
        if (count > quantum - q_pos)
                count = quantum - q_pos;

        if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
                retval = -EFAULT;
                goto out;
        }
        *f_pos += count;
        retval = count;

  out:
        up(&dev->sem);
        return retval;
}
```

# scull's write method

```
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
                loff_t *f_pos)
{
        struct scull_dev *dev = filp->private_data;
        struct scull_qset *dptr;
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset;
        int item, s_pos, q_pos, rest;
        ssize_t retval = -ENOMEM; /* value used in "goto out" statements */

        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;

        /* find listitem, qset index and offset in the quantum */
        item = (long)*f_pos / itemsize;
        rest = (long)*f_pos % itemsize;
        s_pos = rest / quantum; q_pos = rest % quantum;

        /* follow the list up to the right position */
        dptr = scull_follow(dev, item);
```

# scull's write method

```
        if (dptr == NULL)
                goto out;
        if (!dptr->data) {
                dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
                if (!dptr->data)
                        goto out;
                memset(dptr->data, 0, qset * sizeof(char *));
        }
        if (!dptr->data[s_pos]) {
                dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
                if (!dptr->data[s_pos])
                        goto out;
        }
        /* write only up to the end of this quantum */
        if (count > quantum - q_pos)
                count = quantum - q_pos;

        if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
                retval = -EFAULT;
                goto out;
        }
```

Accusys

## scull's write method

```
        *f_pos += count;
        retval = count;

       /* update the size */
        if (dev->size < *f_pos)
                dev->size = *f_pos;

  out:
        up(&dev->sem);
        return retval;
}
```

- Try to play with the new devices now
  - Use **free** command to see how the amount of free memory shrinks
  - Use **strace** utility to trace a **cp** or **ls –l > /dev/scull0** (it shows you quantized reads and writes)

Accusys

## ioctl – beyond read and write

- Operations other than read and write are supported via **ioctl** system call
  - Use of **ioctl** is not recommended as it is essential an undocumented system call (new drivers uses sysfs instead)
- `File_operations.ioctl()` prototype

```
// cmd: ioctl command
// arg: the argument for the ioctl command. Usually this is a pointer
// to user space
int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,
                  unsigned long arg);
```

## ioctl command format

- **ioctl** command should be unique across the system to prevent errors caused by issuing the right command to the wrong device
- `Documentation/ioctl-number.txt` contains a list of magic numbers used in the kernel

| direction (2 bits) | size (14 bits) | type (8 bits) | number (8 bits) |
|---|---|---|---|

31 0

type: magic number associated with the device
number: sequence number unique within the device
direction: none, read from device, write to device, rw
size: the size of user data involved

## Macros encoding **ioctl** commands

```
/*
 * Direction bits.
 */
#define _IOC_NONE  0U
#define _IOC_WRITE 1U
#define _IOC_READ  2U  /* read from device, i.e. write to user space */

#define _IOC(dir,type,nr,size) \
        (((dir)  << _IOC_DIRSHIFT) | ((type) << _IOC_TYPESHIFT) | \
         ((nr)   << _IOC_NRSHIFT) | ((size) << _IOC_SIZESHIFT))

// for command that has no argument
#define _IO(type,nr)        _IOC(_IOC_NONE,(type),(nr),0)
#define _IOR(type,nr,size)  _IOC(_IOC_READ,(type),(nr),sizeof(size))
#define _IOW(type,nr,size)  _IOC(_IOC_WRITE,(type),(nr),sizeof(size))
#define _IOWR(type,nr,size) _IOC(_IOC_READ|IOC_WRITE,(type),(nr),sizeof(size))
```

## scull's ioctl command definition

```
/* Use 'k' as magic number */
#define SCULL_IOC_MAGIC  'k'
#define SCULL_IOC_MAXNR   14
#define SCULL_IOCRESET    _IO(SCULL_IOC_MAGIC, 0)

/*
 * S means "Set" through a ptr,
 * T means "Tell" directly with the argument value
 * G means "Get": reply by setting through a pointer
 * Q means "Query": response is on the return value
*/
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC,  1, int)
#define SCULL_IOCSQSET    _IOW(SCULL_IOC_MAGIC,  2, int)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC,   3)
#define SCULL_IOCTQSET    _IO(SCULL_IOC_MAGIC,   4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC,  5, int)
#define SCULL_IOCGQSET    _IOR(SCULL_IOC_MAGIC,  6, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC,   7)
#define SCULL_IOCQQSET    _IO(SCULL_IOC_MAGIC,   8)
...
```

## scull's ioctl implementation

```
int scull_ioctl(struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg) {
    int err = 0, tmp;
    int retval = 0;

    // extract the type and number bitfields, and don't decode
    // wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
    if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
    if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

    // the direction is a bitmask, and VERIFY_WRITE catches R/W transfers
    // `Type' is user-oriented, while access_ok is kernel-oriented, so
    // the concept of "read" and "write" is reversed
    if (_IOC_DIR(cmd) & _IOC_READ)
        err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
    else if (_IOC_DIR(cmd) & _IOC_WRITE)
        err =  !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
    if (err) return -EFAULT;
```

# scull's ioctl implementation

```
    switch(cmd) {
        case SCULL_IOCRESET:
            scull_quantum = SCULL_QUANTUM;
            scull_qset = SCULL_QSET;
            break;
        case SCULL_IOCSQUANTUM: /* Set: arg points to the value */
            if (! capable (CAP_SYS_ADMIN)) return -EPERM;
            retval = __get_user(scull_quantum, (int __user *)arg);
            break;
        case SCULL_IOCTQUANTUM: /* Tell: arg is the value */
            if (! capable (CAP_SYS_ADMIN)) return -EPERM;
            scull_quantum = arg;
            break;
        case SCULL_IOCGQUANTUM: /* Get: arg is pointer to result */
            retval = __put_user(scull_quantum, (int __user *)arg);
            break;
        case SCULL_IOCQQUANTUM: /* Query: return it (it's positive) */
            return scull_quantum;
```

# scull's ioctl implementation

```
        case SCULL_IOCSQSET:
            if (! capable (CAP_SYS_ADMIN)) return -EPERM;
            retval = __get_user(scull_qset, (int __user *)arg);
            break;
        case SCULL_IOCTQSET:
            if (! capable (CAP_SYS_ADMIN)) return -EPERM;
            scull_qset = arg;
            break;
        case SCULL_IOCGQSET:
            retval = __put_user(scull_qset, (int __user *)arg);
            break;
        case SCULL_IOCQQSET:
            return scull_qset;
        default:  /* redundant, as cmd was checked against MAXNR */
            return -ENOTTY;
    }
    return retval;
}
```

## The llseek method

- If llseek method is not defined, the default implementation in the kernel performs seeks by modifying `filp->f_pos`
  - Override this function if the seek operation involves physical operation on the device
- If your device does not support seek operation
  - Inform the kernel by calling `nonseekable_open()` in your open method
  - Set llseek method in your `file_operations` to `no_llseek()`

## scull's llseek implementation

```
loff_t scull_llseek(struct file *filp, loff_t off, int whence) {
        struct scull_dev *dev = filp->private_data;
        loff_t newpos;

        switch(whence) {
          case 0: /* SEEK_SET */
                newpos = off;
                break;
          case 1: /* SEEK_CUR */
                newpos = filp->f_pos + off;
                break;
          case 2: /* SEEK_END */
                newpos = dev->size + off;
                break;
          default: /* can't happen */
                return -EINVAL;
        }
        if (newpos < 0) return -EINVAL;
        filp->f_pos = newpos;
        return newpos;
}
```

## Process sleeping

- Processes need to sleep when requests cannot be satisfied immediately
  - Kernel output buffer is full or no data is available
- Rule for sleeping
  - Never sleep in an atomic context
    - Holding a spinlock, seqlock or RCU lock
    - Interrupts are disabled
  - Always check to ensure that the condition the process was waiting for is indeed true after the process wakes up

## Wait queue

- Wait queue contains a list of processes, all waiting for a specific event
- Declaration and initialization of wait queue

```
// defined and initialized statically with
DECLARE_WAIT_QUEUE_HEAD(name);

// initialized dynamically
Wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

# wait_event macros

```
// queue: the wait queue head to use. Note that it is passed "by value"
// condition: arbitrary boolean expression, evaluated by the macro before
//            and after sleeping until the condition becomes true. It may
//            be evaluated an arbitrary number of times, so it should not
//            have any side effects.
// timeout: wait for the specific number of clock ticks (in jiffies)

// uninterruptible sleep until a condition gets true
wait_event(queue, condition);
// interruptible sleep until a condition gets true, return –ERESTARTSYS if
// interrupted by a signal, return 0 if condition evaluated to be true
wait_event_interruptible(queue, condition);
// uninterruptible sleep until a condition gets true or a timeout elapses
// return 0 if the timeout elapsed, and the remaining jiffies if the
// condition evaluated to true before the timout elapsed
wait_event_timeout(queue, condition, timeout);
// interruptible sleep until a condition gets true or a timeout elapses
// return 0 if the timeout elapsed, -ERESTARTSYS if interrupted by a
// signal, and the remaining jiffies if the condition evaluated to true
// before the timout elapsed
wait_event_interruptible_timeout(queue, condition, timeout);
```

# wake_up macros

- Within a real device driver, a process blocked in a `read` call is awaken when data arrives; usually the hardware issues an interrupt to signal such an event, and the driver awakens waiting processes as part of handling the interrupt

```
// Wake processes that are sleeping on the queue q. The _interruptible
// form wakes only interruptible processes. Normally, only one exclusive
// waiter is awakened (to avoid thundering herd problem), but that
// behavior can be changed with the _nr or _all forms. The _sync version
// does not reschedule the CPU before returning.

void wake_up(struct wait_queue **q);
void wake_up_interruptible(struct wait_queue **q);
void wake_up_nr(struct wait_queue **q, int nr);
void wake_up_interruptible_nr(struct wait_queue **q, int nr);
void wake_up_all(struct wait_queue **q);
void wake_up_interruptible_all(struct wait_queue **q);
void wake_up_interruptible_sync(struct wait_queue **q);
```

## A simple example of putting processes to sleep

- **sleepy** device behavior: any process that attempts to read from the device is put to sleep. Whenever a process writes to the device, all sleeping processes are awaken
- Note that on single processor, the second process to wake up would immediately go back to sleep

## **sleepy**'s read and write

```
ssize_t sleepy_read (struct file *filp, char __user *buf,
                         size_t count, loff_t *pos) {
        printk(KERN_DEBUG "process %i (%s) going to sleep\n",
                         current->pid, current->comm);
        wait_event_interruptible(wq, flag != 0);
        flag = 0;
        printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
        return 0; /* EOF */
}

ssize_t sleepy_write (struct file *filp, const char __user *buf,
                         size_t count, loff_t *pos) {
        printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
                         current->pid, current->comm);
        flag = 1;
        wake_up_interruptible(&wq);
        return count; /* succeed, to avoid retrial */
}
```

## Implementation of **wait_event**: How to implement sleep manually

```
#define wait_event(wq, condition)           \
do {                                         \
        if (condition)                       \
                break;                       \
        __wait_event(wq, condition);         \
} while (0)

#define __wait_event(wq, condition)          \
do {                                         \
        DEFINE_WAIT(__wait);                 \
                                             \
        for (;;) {                           \
                prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE); \
                if (condition)               \
                        break;               \
                schedule();                  \
        }                                    \
        finish_wait(&wq, &__wait);           \
} while (0)
```

## Implementation of **wait_event**: How to implement sleep manually

- `prepare_to_wait`
  - add wait queue entry to the wait queue and set the process state
- `finish_wait`
  - set task state to `TASK_RUNNING` and remove wait queue entry from wait queue
- Questions:
  - What if the '`if (condition) ..`' statement is moved to the front of prepare_to_wait()?
  - What if the '`wake_up`' event happens just after the '`if (condition) ..`' statement but before the execution of the `schedule()` function?

## Nonblocking I/O

- If there are other tasks to be performed, applications may not want to be blocked in the kernel when its requests cannot be satisified immediately
  - Example: single process event driven web server
- When files are opened with `O_NONBLOCK` flag, the behavior of `read`, `write` and `open` is changed

## **scullpipe** – example of blocking and nonblocking I/O

- **scullpipe** is a pipe-like device

```
struct scull_pipe {
    wait_queue_head_t inq, outq;       /* read and write queues */
    char *buffer, *end;                /* begin of buf, end of buf */
    int buffersize;                    /* used in pointer arithmetic */
    char *rp, *wp;                     /* where to read, where to write */
    int nreaders, nwriters;            /* number of openings for r/w */
    struct fasync_struct *async_queue; /* asynchronous readers */
    struct semaphore sem;              /* mutual exclusion semaphore */
    struct cdev cdev;                  /* Char device structure */
};
```

## scullpipe's read method

```
static ssize_t scull_p_read (struct file *filp, char __user *buf, size_t count,
                loff_t *f_pos) {
    struct scull_pipe *dev = filp->private_data;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    while (dev->rp == dev->wp) { /* nothing to read */
        up(&dev->sem); /* release the lock */
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        PDEBUG("\"%s\" reading: going to sleep\n", current->comm);
        if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
            return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
        /* otherwise loop, but first reacquire the lock */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
```

## scullpipe's read method

```
    /* ok, data is there, return something */
    if (dev->wp > dev->rp)
        count = min(count, (size_t)(dev->wp - dev->rp));
    else /* the write pointer has wrapped, return data up to dev->end */
        count = min(count, (size_t)(dev->end - dev->rp));
    if (copy_to_user(buf, dev->rp, count)) {
        up (&dev->sem);
        return -EFAULT;
    }
    dev->rp += count;
    if (dev->rp == dev->end)
        dev->rp = dev->buffer; /* wrapped */
    up (&dev->sem);

    /* finally, awake any writers and return */
    wake_up_interruptible(&dev->outq);
    PDEBUG("\"%s\" did read %li bytes\n",current->comm, (long)count);
    return count;
}
```

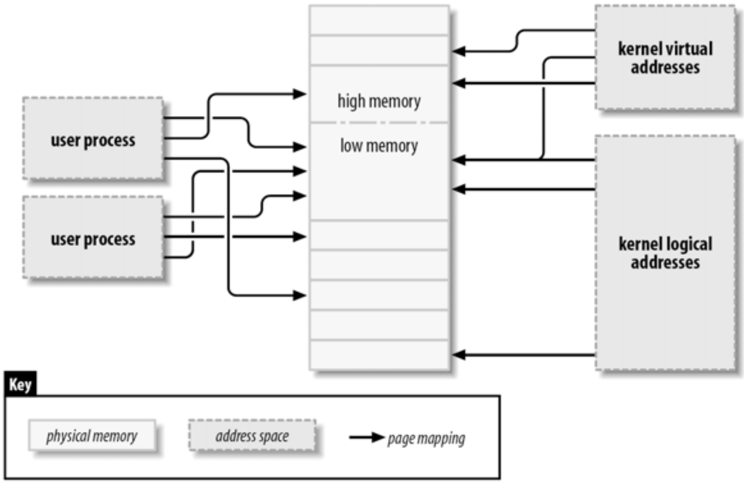
## Address types in Linux

- User virtual addresses
  - Regular addresses seen by user-space programs
  - Each process has its own virtual address space
- Physical addresses
  - Used between the processor and the system's memory
- Bus addresses
  - Used between peripheral buses and memory
  - Often they are the same as the physical addresses used by the processor
  - Some architectures has a IOMMU to remap addresses between bus and memory

## Address types in Linux

- Kernel logical addresses
  - Normal address space of the kernel mapping portion (perhaps all) of main memory
  - Logical addresses and their associated physical addresses differ only by a constant offset
- Kernel virtual addresses
  - The addresses do not necessarily have the linear, one-to-one mapping to physical addresses
  - All logical addresses are kernel virtual addresses, but many kernel virtual addresses are not logical addresses
  - Physical pages are mapped into the reserved virtual address space when needed
  - vmalloc(), kmap(), ioremap()

# Address types used in Linux




# High and low memory

- On 32 bit architecture, 4GB virtual address space is split.


- The amount of memory that is statically mapped into kernel address space is less than 1GB
  - If kernel want to access a page that is not mapped, kernel will have to map it dynamically into a small reserved address space before it can access the page
- **Low memory**: memory for which logical address exist in kernel space
- **High memory**: memory for which logical addresses do not exist

## Physical pages management

- Memory are usually managed in units of pages, and typical page size on most architecture is 4096 bytes
- `struct page` is used to keep track of all information about a page; kernel maintains an array of `struct page` called `mem_map` that tracks all physical memory on a system
- Kernel functions that deal with memory often use pointers to `struct page` instead of logical addresses because logical addresses are not available for high memory

## Address translation in the kernel

- A list of address translation macros for x86
  - Virtual address translation does not work with memory from `vmalloc` or high memory

```
#define __pa(x)                     ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x)                     ((void *)((unsigned long)(x)+PAGE_OFFSET))
static inline unsigned long virt_to_phys(volatile void * address)
{
        return __pa(address);
}
static inline void * phys_to_virt(unsigned long address)
{
        return __va(address);
}
#define pfn_to_page(pfn)           (mem_map + (pfn))
#define virt_to_page(kaddr)        pfn_to_page(__pa(kaddr) >> PAGE_SHIFT)
#define page_to_pfn(page)          ((unsigned long)((page) - mem_map))
#define page_to_phys(page)         ((dma_addr_t)page_to_pfn(page) << PAGE_SHIFT)
#define page_address(page)         __va(page_to_pfn(page) << PAGE_SHIFT)
#define virt_to_bus                virt_to_phys
#define bus_to_virt                phys_to_virt
```

Accusys

# Dynamic address mapping

```
// return a kernel virtual address for any page in the system. For low-memory
// pages, it just returns the logical address of the page; for high-memory
// pages, kmap creates a special mapping in a dedicated part of the kernel
// address space. Since a limited number of such mappings is available,
// mappings created with kmap should always be freed with kunmap. Note also
// that kmap can sleep if no mappings are available.
void *kmap(struct page *page);
void kunmap(struct page *page);

// a high-performance form of kmap. This function must be used in an atomic
// context. You code cannot sleep while holding one.
// type: which reserved slot (dedicated page table entries) to use (KM_USER0,
// KMUSER1, KM_IRQ0, KM_IRQ1)
void *kmap_atomic(struct page *page, enum km_type type);
void *kunmap_atomic(void *addr, enum km_type type);
```

Accusys

# Virtual memory areas

- A VMA represents a homogeneous region in the virtual memory of a process
  - A contiguous range of virtual addresses that have the same permission flags and are backed up by the same object (a file or swap space)
- When a process calls `mmap` to map a file or device memory into its address space, a new VMA is created to represent that mapping

# Memory maps of init process

```
# cat /proc/1/maps look at init
08048000-0804e000 r-xp 00000000 03:01 64652       /sbin/init text
0804e000-0804f000 rw-p 00006000 03:01 64652       /sbin/init data
0804f000-08053000 rwxp 00000000 00:00 0           zero-mapped BSS
40000000-40015000 r-xp 00000000 03:01 96278       /lib/ld-2.3.2.so text
40015000-40016000 rw-p 00014000 03:01 96278       /lib/ld-2.3.2.so data
40016000-40017000 rw-p 00000000 00:00 0           BSS for ld.so
42000000-4212e000 r-xp 00000000 03:01 80290       /lib/tls/libc-2.3.2.so text
4212e000-42131000 rw-p 0012e000 03:01 80290       /lib/tls/libc-2.3.2.so data
42131000-42133000 rw-p 00000000 00:00 0           BSS for libc
bffff000-c0000000 rwxp 00000000 00:00 0           Stack segment
ffffe000-fffff000 ---p 00000000 00:00 0           vsyscall page

start    end      perm offset major:minor inode    image

Each field corresponds to a field in struct vm_area_struct
```

# struct vm_area_struct

- Data structure representing a VMA

```
struct vm_area_struct {
   struct mm_struct * vm_mm;      // The address space we belong to.
   unsigned long vm_start;        // Our start address within vm_mm.
   unsigned long vm_end;          // first byte after our end address within
                                  // vm_mm
   struct file * vm_file;         // File we map to (can be NULL).

   // VM_IO marks a VMA as being a memory-mapped I/O region. It prevents the
   // region from being included in process core dumps.
   // VM_RESERVED tells the system not to swap out this VMA; it should be set
   // in most device mappings
   unsigned long vm_flags;
   struct vm_operations_struct * vm_ops;
   void *vm_private_data;         // used by the driver to store its own info
   ...
}
```

# struct vm_operations_struct

```
struct vm_operations_struct {
        // invoked everytime a new reference to the VMA is made. The one
        // exception happens when the VMA is first created by mmap; in this
        // case, the driver's mmap method is called instead
        void (*open)(struct vm_area_struct * area);
        // when an area is destroyed (unmap or process exit), the kernel calls
        // its close operation
        void (*close)(struct vm_area_struct * area);
        // when a process tries to access a page that belongs to a valid VMA,
        // but that is currently not in memory, the nopage method is called
        // the method returns a struct page pointer to the requested page.
        // Kernel will set the page table for the page we return
        // address: the virtual address that caused the fault
        struct page * (*nopage)(struct vm_area_struct * area,
                                unsigned long address, int *type);
        // This method allows the kernel to prefault pages into memory before
        // they are accessed by user space
        int (*populate)(struct vm_area_struct * area, unsigned long address,
                        unsigned long len, pgprot_t prot, unsigned long pgoff,
                        int nonblock);
}
```

# The benefits of using `mmap()` in user application

- Allows user program with direct access to device memory (zero-copy)
- Mapping must be done in units of PAGE_SIZE and the mapped area must be aligned on PAGE_SIZE boundary
- `mmap()` prototype

```
// The mmap function asks to map length bytes starting at offset offset from
// the file (or other object) specified by the file descriptor fd  into
// memory, preferably at address start. This latter address is a hint only,
// and is usually specified as 0.
void * mmap(void *start, size_t length, int prot , int flags, int fd,
                off_t offset);
int munmap(void *start, size_t length);
```

# Memory mapping of X server

- X server maps video memory into user space for direct access

```
cat /proc/731/maps
000a0000-000c0000 rwxs 000a0000 03:01 282652 /dev/mem
000f0000-00100000 r-xs 000f0000 03:01 282652 /dev/mem
00400000-005c0000 r-xp 00000000 03:01 1366927 /usr/X11R6/bin/Xorg
006bf000-006f7000 rw-p 001bf000 03:01 1366927 /usr/X11R6/bin/Xorg
2a95828000-2a958a8000 rw-s fcc00000 03:01 282652 /dev/mem
2a958a8000-2a9d8a8000 rw-s e8000000 03:01 282652 /dev/mem
...

Cat /proc/iomem
000a0000-000bffff : Video RAM area
000c0000-000ccfff : Video ROM
000d1000-000d1fff : Adapter ROM
000f0000-000fffff : System ROM
d7f00000-f7efffff : PCI Bus #01
e8000000-efffffff : 0000:01:00.0
fc700000-fccfffff : PCI Bus #01
fcc00000-fcc0ffff : 0000:01:00.0
```

# Add `mmap` support to your driver

- There are two ways of building page tables
  - All at once by calling `remap_pfn_range`
  - Doing it a page at a time via the `nopage` VMA method

```
// vma: the virtual memory area into which the page range is being mapped
// virt_addr: the user virtual address where remapping should begin
// pfn: the page frame number corresponding to the physical address to which
//      the virtual  address should be mapped
// size: the size of the area being mapped, in bytes
// prot: protection flag, driver should use the value in vma->vm_page_prot

int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr,
                 unsigned long pfn, unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma, unsigned long virt_addr,
                 unsigned long phys_addr, unsigned long size, pgprot_t prot);
```

# A **simple** implementation – all at once

- This example maps system memory into user space and is derived from `drivers/char/mem.c`

```
static struct file_operations simple_remap_ops = {
        .mmap    = simple_remap_mmap,
        ...
};

static int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
        if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,
                            vma->vm_end - vma->vm_start,
                            vma->vm_page_prot))
                return -EAGAIN;

        vma->vm_ops = &simple_remap_vm_ops;
        // since open is not invoked on the initial mmap
        simple_vma_open(vma);
        return 0;
}
```

# A **simple** implementation – all at once

```
static struct vm_operations_struct simple_remap_vm_ops = {
        .open =  simple_vma_open,
        .close = simple_vma_close,
};

void simple_vma_open(struct vm_area_struct *vma)
{
        printk(KERN_NOTICE "Simple VMA open, virt %lx, phys %lx\n",
                         vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void simple_vma_close(struct vm_area_struct *vma)
{
        printk(KERN_NOTICE "Simple VMA close.\n");
}
```

# A **simple** implementation – a page at a time

- `nopage` VMA method is needed when:
  - Drivers want to support `mremap`. Since the kernel does not notify drivers directly when a mapped VMA is expanded (kernel only adjusts mapping range in VMA structure, driver are notified by calling its `nopage`)
  - Recall: when a page fault occurs and it is legal, `nopage` is invoked
- What to do in `nopage`:
  - Locate the page user requested
  - Increment the usage count for the page

# A **simple** implementation – a page at a time

```
static int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma) {
        unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

        if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
                vma->vm_flags |= VM_IO;
        vma->vm_flags |= VM_RESERVED;

        vma->vm_ops = &simple_nopage_vm_ops;
        simple_vma_open(vma);
        return 0;
}
```

# A **simple** implementation – a page at a time

```
struct page *simple_vma_nopage(struct vm_area_struct *vma,
                unsigned long address, int *type) {
        struct page *pageptr;
        unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
        unsigned long physaddr = address - vma->vm_start + offset;
        unsigned long pageframe = physaddr >> PAGE_SHIFT;

        if (!pfn_valid(pageframe))
                return NOPAGE_SIGBUS; // cause bus signal to be delivered
        pageptr = pfn_to_page(pageframe);
        get_page(pageptr);
        if (type)
                *type = VM_FAULT_MINOR; // minor since no disk I/O involved
        return pageptr;
}
```

# Notes on `remap_pfn_range`

- It only maps reserved pages and physical addresses above the top of physical memory
  - Kernel code, 640KB-1MB ISA I/O range
  - Pages allocated from page allocator are not mapped. Instead, it maps in the zero page
- Why? (hint: reference count)

```
morgana.root# ./mapper /dev/mem 0x10000 0x1000 | od -Ax -t x1
mapped "/dev/mem" from 65536 to 69632
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001000
```

## Notes on **nopage**

- PCI memory is mapped above the highest system memory, and there are no `struct page` entries in the `mem_map` array for those addresses
- If `nopage` method is left `NULL`, kernel maps the zero page to the faulting virtual address
  - If a process extends a mapped region by calling mremap and `nopage` is not defined, the process ends up with zero-filled memory
  - To avoid:

```
struct page *simple_nopage(struct vm_area_struct *vma, unsigned long address,
                           int *type) {
        return NOPAGE_SIGBUS; /* send a SIGBUS */
}
```

## **scullp** with mmap implementation -- use **nopage** to map normal RAM

- **scullp** allocates memory in units of pages
- For reference counting, **scullp** does not allow allocation with order > 0
  - Linux only increment the reference count of the first page of the allocated pages returned from page allocator (`alloc_pages`, `get_free_pages`)
  - This is the reason why remap_pfn_range does not allow mapping of non-reserved normal pages

## sculLp with mmap implementation -- use nopage to map normal RAM

```
int scullp_mmap(struct file *filp, struct vm_area_struct *vma)
{
        struct inode *inode = filp->f_dentry->d_inode;

        /* refuse to map if order is not 0 */
        if (scullp_devices[iminor(inode)].order)
                return -ENODEV;

        /* don't do anything here: "nopage" will set up page table entries */
        vma->vm_ops = &scullp_vm_ops;
        vma->vm_flags |= VM_RESERVED;
        vma->vm_private_data = filp->private_data;
        scullp_vma_open(vma);
        return 0;
}
```

## scullp with mmap implementation -- use nopage to map normal RAM

```
/* The nopage method: the core of the file. It retrieves the
 * page required from the scullp device and returns it to the
 * user. The count for the page must be incremented, because
 * it is automatically decremented at page unmap.
 *
 * For this reason, "order" must be zero. Otherwise, only the first
 * page has its count incremented, and the allocating module must
 * release it as a whole block. Therefore, it isn't possible to map
 * pages from a multipage block: when they are unmapped, their count
 * is individually decreased, and would drop to 0.
 */
struct page *scullp_vma_nopage(struct vm_area_struct *vma,
                                unsigned long address, int *type)
{
        unsigned long offset;
        struct scullp_dev *ptr, *dev = vma->vm_private_data;
        struct page *page = NOPAGE_SIGBUS;
        void *pageptr = NULL; /* default to "missing" */

        down(&dev->sem);
        offset = (address - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT);
        if (offset >= dev->size) goto out; /* out of range */
```

# **scullp** with mmap implementation -- use **nopage** to map normal RAM

```
        /* Now retrieve the scullp device from the list,then the page.
         * If the device has holes, the process receives a SIGBUS when
         * accessing the hole.
         */
        offset >>= PAGE_SHIFT; /* offset is a number of pages */
        for (ptr = dev; ptr && offset >= dev->qset;) {
                ptr = ptr->next;
                offset -= dev->qset;
        }
        if (ptr && ptr->data) pageptr = ptr->data[offset];
        if (!pageptr) goto out; /* hole or end-of-file */
        page = virt_to_page(pageptr);

        /* got it, now increment the count */
        get_page(page);
        if (type)
                *type = VM_FAULT_MINOR;
  out:
        up(&dev->sem);
        return page;
}
```

# Testing mmap in **scullp**

- Use mapper to see if the data we wrote using write() can be seen in the remapped memory range

```
morgana% ls -l /dev > /dev/scullp
morgana% ./mapper /dev/scullp 0 140
mapped "/dev/scullp" from 0 (0x00000000) to 140 (0x0000008c)
total 232
crw------- 1 root root 10, 10 Sep 15 07:40 adbmouse
crw-r--r-- 1 root root 10, 175 Sep 15 07:40 agpgart
morgana% ./mapper /dev/scullp 8192 200
mapped "/dev/scullp" from 8192 (0x00002000) to 8392 (0x000020c8)
d0h1494
brw-rw---- 1 root floppy 2, 92 Sep 15 07:40 fd0h1660
brw-rw---- 1 root floppy 2, 20 Sep 15 07:40 fd0h360
brw-rw---- 1 root floppy 2, 12 Sep 15 07:40 fd0H360
```

## Remapping vmalloc'ed memory

- Unlike alloc_pages or get_free_pages, there is no reference counting problem with vmalloc() since vmalloc() allocates one page at a time

```
struct page *scullv_vma_nopage(struct vm_area_struct *vma,
                                unsigned long address, int *type)
{
        unsigned long offset;
        struct scullv_dev *ptr, *dev = vma->vm_private_data;
        struct page *page = NOPAGE_SIGBUS;
        void *pageptr = NULL; /* default to "missing" */

        down(&dev->sem);
        offset = (address - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT);
        if (offset >= dev->size) goto out; /* out of range */

        /* Now retrieve the scullv device from the list,then the page.
         * If the device has holes, the process receives a SIGBUS when
         * accessing the hole.
         */
```

## Remapping vmalloc'ed memory

```
        offset >>= PAGE_SHIFT; /* offset is a number of pages */
        for (ptr = dev; ptr && offset >= dev->qset;) {
                ptr = ptr->next;
                offset -= dev->qset;
        }
        if (ptr && ptr->data) pageptr = ptr->data[offset];
        if (!pageptr) goto out; /* hole or end-of-file */

        /* After scullv lookup, "page" is now the address of the page
         * needed by the current process. Since it's a vmalloc address,
         * turn it into a struct page.
         */
        page = vmalloc_to_page(pageptr);

        /* got it, now increment the count */
        get_page(page);
        if (type)
                *type = VM_FAULT_MINOR;
  out:
        up(&dev->sem);
        return page;
}
```

# Linux Kernel
# Access Hardware

Paul Chu
Hao-Ran Liu

# Agenda

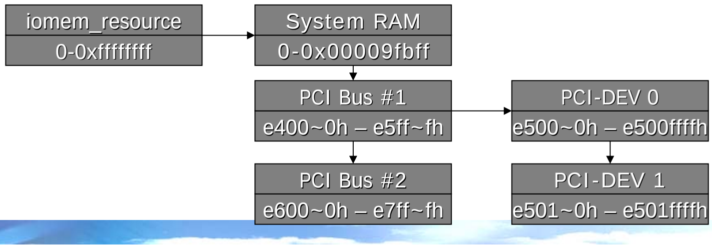
- Hardware resources
- Access IO ports
- Access memory-mapped IO
- DMA
- PCI

# Hardware Resource Management



- A resource is a range of h/w address space
  - /proc/iomem & /proc/ioports
- Resources are managed in trees
  - Parent node contains all ranges of children nodes

```
struct resource {
   const char *name;                          // name of resource
   unsigned long start, end;                  // space start and end
   unsigned long flags;                       // IO or memory
   struct resource *parent, *sibling, *child; // resource trees
};
```




# Request Hardware Resources



- Allocate an empty resource

```
int allocate_resource(root, new, size, min, max, align, alignfn, data);
```

- Request a resource

```
int request_resource(parent, new); // searching siblings under the parent
```

- Release a resource

```
void release_resource(new);
```

- Request for IO port and memory resource
  - IO port region (ioport_resource)

```
resource* request_region(start, n, name); → request_resource(...)
void release_region(resource, start, n); → release_resource(...)
int check_region(start, n); → no need to call
```

  - Memory region (iomem_resource)

```
void release_mem_region(resource, start, n); → request_resource(...)
resource* request_mem_region(start, n, name); → release_resource(...)
int check_mem_region(start, n); → no need to call
```

# Using IO Ports



- Access an IO port
  - Output data to an IO port
    - outb(byte, addr), outw(word, addr), outl(long,addr)
  - Input data from an IO port
    - inb(addr), inw(addr), inl(addr)
- Perform string IO
  - Output a string to an IO port
    - outsb/outsw/outsl(addr, *data, count)
  - Input a string from an IO port
    - insb/insw/insl(addr, *data, count)
- Slow IO access
  - Inserting delay after an IO port
  - xxx_p(); e.g. outb_p(…), inl_p(…)
- When IO space (native IO instruction) does not exist, memory-mapped IO is used instead

# Memory-mapped IO Space



- Memory-mapped IO
  - Instead of extra input/output (x86) instructions, all devices occupy a range of physical address space as the communication interface with CPU (no IO space).
  - Accesses to the range will be forwarded to the peripheral bus and received by device on the bus
  - Inserting barriers to flush pending accesses
- IO remap
  - Mapping the space of memory-mapped IO to kernel virtual address
  - Kernel can access the device by issuing read/write to the remapped addresses

```
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
```

# Access Memory-mapped IO Space

*Access Devices*

- Comparing IO memory and conventional memory
  - No pre-read, no out-of-order request, no aggregated request, and no cache
  - Only FIFO pending request queue is allowed
  - Endianess translation may be needed
- Access an IO memory address
  - iowrite8/16/32(data, addr)
  - ioread8/16/32(addr)
- Perform string IO
  - iowrite8/16/32_rep(addr, *data, count)
  - ioread8/16/32_rep(addr, *data, count)
- IO memory utilities
  - memset_io(addr, data, count)
  - memcpy_fromio(to, from, count)
  - memcpy_toio(to, from, count)

# Ports as I/O memory

*Access Devices*

- Different versions of the some hardware may export different access method to the processor
  - To make driver programming easier, kernel provides API to make IO ports appear to be IO memory

```
/* We encode the physical PIO addresses (0-0xffff) into the
 * pointer by offsetting them with a constant (0x10000) and
 * assuming that all the low addresses are always PIO. That means
 * we can do some sanity checks on the low bits, and don't
 * need to just take things for granted.
 */
#define PIO_OFFSET          0x10000UL
#define PIO_MASK            0x0ffffUL
#define PIO_RESERVED        0x40000UL
void __iomem *ioport_map(unsigned long port, unsigned int nr) {
        if (port > PIO_MASK) return NULL;
        return (void __iomem *) (unsigned long) (port + PIO_OFFSET);
}
```

# Ports as I/O memory



```
void ioport_unmap(void __iomem *addr) {
        /* Nothing to do */
}

#define VERIFY_PIO(port) BUG_ON((port & ~PIO_MASK) != PIO_OFFSET)
#define IO_COND(addr, is_pio, is_mmio) do {                      \
        unsigned long port = (unsigned long __force)addr;        \
        if (port < PIO_RESERVED) {                               \
                VERIFY_PIO(port);                                \
                port &= PIO_MASK;                                \
                is_pio;                                          \
        } else {                                                 \
                is_mmio;                                         \
        }                                                        \
} while (0)

unsigned int fastcall ioread8(void __iomem *addr) {
        IO_COND(addr, return inb(port), return readb(addr));
}
```

# DMA Introduction



- DMA (Direct Memory Access) controller
  - A h/w device that performs data transfer between addresses
  - Freeing CPU from doing routine and slow jobs
- DMA processing flow
  - Driver prepares buffers in memory and instructs the device to start data transfer
  - The device behaves as a bus master to read/write data from/to one address and write/read to/from another address
  - When the data transfer is done, the device signals an interrupt
  - The interrupt handler acknowledges and prepares another data transfer
- DMA controller vs. device bus master
  - External device or embedded DMA capability

## More About DMA



- DMA issues
  - Translation of virtual address to bus address
  - DMA bounce
    - limited DMA window → fixed by copying data
  - Data coherency between CPU and device
    - Flush cache for device to write to
    - Invalidate cache for device to read from
- Scatter gather list
  - Data spreading over more than one chunk, each of which has different sizes
  - struct scatterlist *sg
    - Buffer page: sg->page
    - Buffer offset: sg->offset
    - DMA address: sg->dma_address
    - Length: sg->length

## DMA mappings



- A DMA mapping is a combination of allocating a DMA buffer and generating an address for that buffer that is accessible by the device
  - A new type, dma_addr_t, is defined to represent bus addresses
- Coherent DMA mappings
  - Available to both CPU and peripheral simultaneously
- Streaming DMA mappings
  - Set up for only one operation
  - Use streaming mapping whenever possible

# Coherent DMA mappings



- Page size Allocation and free
  - void *dma_alloc_coherent(dev, size, *dma_addr, flag)
  - void dma_free_coherent(dev, size, vaddr, dma_addr)
- Pool for small-than-page-size DMA buffer
  - struct dma_pool *dma_pool_create(name, dev, size, align, allocation)
  - void dma_pool_destroy(*pool)
  - void *dma_pool_alloc(*pool, mem_flags, *dma_addr)
  - void dma_pool_free(*pool, *vaddr, dma_addr)

# Streaming DMA mappings



- Direction of data transfer (enum dma_data_direction)
  - DMA_TO_DEVICE
  - DMA_FROM_DEVICE
  - DMA_BIDIRECTIONAL
- Rules of using streaming DMA mappings
  - The buffer must be used only for a transfer that matches the direction value given when it was mapped
  - Once a buffer has been mapped, it belongs to the device, not the processor. Until the buffer has been unmapped, the driver should not touch its contents in any way
  - The buffer must not be unmapped while DMA is still active

## Streaming DMA mappings



- Single entry
  - dma_addr_t dma_map_single(struct device *dev, void *ptr, size_t size, enum dma_data_direction dir)
  - dma_unmap_single(struct device *dev, dma_addr_t addr, size_t size, enum dma_data_direction dir)
  - dma_addr_t dma_map_page(struct device *dev, struct page *page, unsigned long offset, size_t size, enum dma_data_direction direction)
  - void dma_unmap_page(struct device *dev, dma_addr_t handle, size_t size, enum dma_data_direction dir)
- Multiple entries (scatter-gather)
  - int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction dir)
  - void dma_unmap_sg(struct device *dev, struct scatterlist *sg, int nents, enum dma_data_direction dir)

## Implementation of dma_map_single()



- Linux's implementation for PowerPC architecture

```
static inline dma_addr_t
dma_map_single(struct device *dev, void *ptr, size_t size,
               enum dma_data_direction direction)
{
        BUG_ON(direction == DMA_NONE);

        __dma_sync(ptr, size, direction);

        return virt_to_bus(ptr);
}

#define dma_unmap_single(dev, addr, size, dir)    do { } while (0)
```

## Implementation of dma_map_single()

DMA

```
void __dma_sync(void *vaddr, size_t size, int direction) {
	unsigned long start = (unsigned long)vaddr;
	unsigned long end   = start + size;

	switch (direction) {
	case DMA_NONE:
		BUG();
	case DMA_FROM_DEVICE:	/* invalidate only */
		invalidate_dcache_range(start, end);
		break;
	case DMA_TO_DEVICE:		/* writeback only */
		clean_dcache_range(start, end);
		break;
	case DMA_BIDIRECTIONAL:	/* writeback and invalidate */
		flush_dcache_range(start, end);
		break;
	}
}
```

## A simple PCI DMA example

DMA

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer, size_t count) {
	dma_addr_t bus_addr;

	/* Map the buffer for DMA */
	dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
	dev->dma_size = count;
	bus_addr = dma_map_single(&dev->pci_dev->dev, buffer, count,
				dev->dma_dir);
	dev->dma_addr = bus_addr;

	/* Set up the device */
	writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
	writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
	writel(dev->registers.addr, cpu_to_le32(bus_addr));
	writel(dev->registers.len, cpu_to_le32(count));

	/* Start the operation */
	writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
	return 0;
}
```

## A simple PCI DMA example

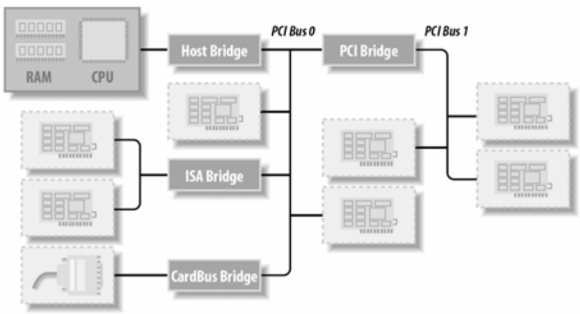


```
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        struct dad_dev *dev = (struct dad_dev *) dev_id;

        /* Make sure it's really our device interrupting */

        /* Unmap the DMA buffer */
        dma_unmap_single(dev->pci_dev->dev, dev->dma_addr,
                         dev->dma_size, dev->dma_dir);

        /* Only now is it safe to access the buffer, copy to user, etc. */
        ...
}
```

## PCI Addressing



- Each PCI device in Linux is identified by domain (16 bits), bus number (8 bits), device number (5 bits) and function number (3 bits)
- Linux uses `pci_dev` to describe PCI devices



## Outpuf of `lspci` command



```
[root@sirius root]# lspci|cut -d: -f1-2
00:00.0 Host bridge
00:01.0 PCI bridge
00:04.0 ISA bridge
00:04.1 IDE interface
00:04.2 USB Controller
00:04.3 USB Controller
00:04.4 Host bridge
00:05.0 Multimedia audio controller
00:09.0 Ethernet controller
01:00.0 VGA compatible controller
```

## PCI configuration space



- At system boot, the BIOS access configuration space of devices by reading or writing registers in the PCI controller
  - The configuration space expolits geographical addressing. Every access addresses only one slot at a time
  - All memory, I/O regions and IRQ numbers offered by the devices are remapped
- When Linux boots, instead of probing, drivers read the configuration space to get the address regions and IRQ numbers of the devices

# PCI Configuration Registers



| | 0x0 | 0x1 | 0x2 | 0x3 | 0x4 | 0x5 | 0x6 | 0x7 | 0x8 | 0x9 | 0xa | 0xb | 0xc | 0xd | 0xe | 0xf |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x00 | Vendor ID | | Device ID | | Command Reg. | | Status Reg. | | Revis-ion ID | Class Code | | | Cache Line | Latency Timer | Header Type | BIST |
| 0x10 | Base Address 0 | | | | Base Address 1 | | | | Base Address 2 | | | | Base Address 3 | | | |
| 0x20 | Base Address 4 | | | | Base Address 5 | | | | CardBus CIS pointer | | | | Subsytem Vendor ID | | Subsytem Device ID | |
| 0x30 | Expansion ROM Base Address | | | | Reserved | | | | | | | | IRQ Line | IRQ Pin | Min_Gnt | Max_Lat |

- Required Register

- Optional Register

* PCI registers are always little-endian

# Identifying a device



- Five registers identify a device:
  - vendor ID, device ID, class, subsystem vendor ID, subsystem device ID
- A PCI driver defines `pci_device_id` table to tell
  - the kernel what kinds of devices it supports
    - Register through `pci_register_driver()` with `pci_driver`
  - the hotplug system what module to load into the kernel
    - Define `MODULE_DEVICE_TABLE`, which creates a variable `__mod_pci_device_table` pointing to the `pci_device_id` array.
    - The `depmod` program will searches all modules for `__mod_pci_device_table` and pull the data out into the `/lib/modules/KERNEL_VERSION/modules.pcimap` file

# Creating PCI ids table



- PCI_DEVICE(vendor, device)
- PCI_DEVICE_CLASS(device_class, device_class_mask)

```
drivers/i2c/busses/i2c-i810.c:

static struct pci_device_id i810_ids[ ] = {
        { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG1) },
        { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG3) },
        { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810E_IG) },
        { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82815_CGC) },
        { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82845G_IG) },
        { 0, },
};

MODULE_DEVICE_TABLE(pci, i810_ids);
```

# Registering a PCI driver



Pointer to the `struct pci_device_id` table

```
static struct pci_driver pci_driver = {
    .name = "pci_skel",
    .id_table = ids,
    .probe = probe,
    .remove = remove,
};

static int __init pci_skel_init(void) {
    return pci_register_driver(&pci_driver);
}

static void __exit pci_skel_exit(void) {
    pci_unregister_driver(&pci_driver);
}
```

```
int (*probe)(struct pci_dev *dev,
const struct pci_device_id *id);
```

Pointer to the probe function in your driver. This function is called by the PCI core when it has a `pci_dev` that it thinks this driver wants to control. The `pci_device_id` that the PCI core used to make this decision is also passed. The driver should properly initialize the device and return 0 if it want to claim the device. Otherwise, it should return a negative value.

The function must call `pci_enable_device()` before accessing any device resource

Remove function is called before `pci_unregister_driver` returns

## Accessing the configuration space



- Access configuration space registers
  - int pci_write_config_dword/word/byte(struct pci_dev *dev, int where, u32/u16/u8 val)
  - int pci_read_config_dword/word/byte(struct pci_dev *dev, int where, u32/u16/u8 *val)
  - Word and dword function do endianness conversion
- PCI device data structure -- struct pci_dev *dev
  - Vendor ID & device ID: dev->vendor & dev->device
  - Device and function number: dev->devfn
  - IRQ number: dev->irq
  - Base address of resources: dev->resource[]
  - PCI bus: (struct pci_bus*) dev->bus

## Retreving resources information



- A PCI devices can have up to 6 I/O address regions and each region consists of either memory or I/O locations
  - I/O registers mapped in physical memory should not be cached by the CPU since each access can have side effects
- Getting region information
  - ulong pci_resource_start(struct pci_dev *dev, int bar)
  - ulong pci_resource_end(struct pci_dev *dev, int bar)
  - ulong pci_resource_len(struct pci_dev *dev, int bar)
  - ulong pci_resource_flags(struct pci_dev *dev, int bar)
    - IORESOURCE_IO or IORESOURCE_MEM
    - IORESOURCE_PREFETCH or IORESOURCE_READONLY

# References

- Linux Device Drivers, 3rd Edition, O'REILLY, 2005
- Linux kernel source 2.6.16

# Linux Kernel
# Interrupt Handling

Paul Chu
Hao-Ran Liu

# Interrupt Basics

- What is interrupt
  - A communication mechanism for hardware components to notify CPU of events. E.g. key strokes and timers.
  - There may be one or more interrupt request lines (IRQ), which is a physical input to the interrupt controller chip. The number of such inputs is limited. (eg. Classic PC has only 15 IRQ lines)
  - Each IRQ has a unique number, which may be used by one or more components.
- Basic flow of interrupt handling
  - When receiving an interrupt, CPU program counter jumps to a pre-defined address (interrupt vectors)
  - The state of interrupted program is saved
  - The corresponding service routine is executed
  - The interrupting component is served, and interrupt signal is removed
  - The state of interrupted program is restored
  - Resume the interrupted program at the interrupted address

# Interrupts and Exceptions

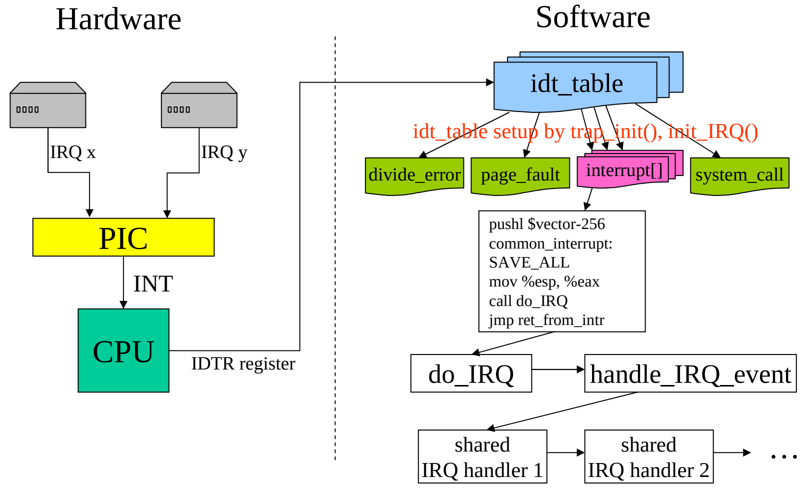
- Interrupts and exceptions are handled by the kernel in a similar way
- Interrupts
  - Asynchronous events generated by external hardware,
  - Interrupt controller chip maps each IRQ input to an interrupt vector, which locates the corresponding interrupt service routine
- Exceptions (Trap)
  - Synchronous events generated by the software
  - E.g. divide by zero, page faults

# Interrupt vectors on x86

| Vector range | Use |
|---|---|
| 0-19 (0x0-0x13) | Nonmaskable interrupts and exceptions |
| 20-31 (0x14-0x1f) | Intel-reserved |
| 32-127 (0x20-0x7f) | External interrupts (IRQs) |
| 128 (0x80) | Programmed exception for system calls |
| 129-238 (0x81-0xee) | External interrupts (IRQs) |
| 239 (0xef) | Local APIC timer interrupt |
| 240-250 (0xf0-0xfa) | Reserved by Linux for future use |
| 251-255 (0xfb-0xff) | Interprocessor interrupts |

The table is from Understanding the Linux kernel, 2nd edition

# Interrupt handling in x86 Linux




# IRQ descriptors

➢ Each IRQ line is associated with an IRQ descriptor

```
typedef struct irq_desc {
   unsigned int status;               /* IRQ status: in progress, disabled, … */
   hw_irq_controller *handler;        /* ack, end, enable, disable irq on PIC */
   struct irqaction *action;          /* IRQ action list */
   unsigned int depth;                /* nested irq disables */
   spinlock_t lock;                   /* serialize access to this structure */
} ____cacheline_aligned irq_desc_t;

irq_desc_t irq_desc[NR_IRQS] __cacheline_aligned = {
            [0 … NR_IRQS-1] = {
                        .handler = &no_irq_type,
                        .lock = SPIN_LOCK_UNLOCKED
            }
};
```

# *status* field of the IRQ descriptor

| Flag name | Description |
|---|---|
| IRQ_INPROGRESS | A handler for the IRQ is being executed. |
| IRQ_DISABLED | The IRQ line has been deliberately disabled by a device driver. |
| IRQ_PENDING | An IRQ has occurred on the line; its occurrence has been acknowledged to the PIC, but it has not yet been serviced by the kernel. |
| IRQ_AUTODETECT | The kernel uses the IRQ line while performing a hardware device probe. |
| IRQ_WAITING | The kernel uses the IRQ line while performing a hardware device probe; moreover, the corresponding interrupt has not been raised. |

The table is from Understanding the Linux kernel, 2nd edition

# Interrupt controller descriptor

- This describes operations of a interrupt controller

```
struct hw_interrupt_type {
 /* the name of the PIC, shown in /proc/interrupts */
  const char * typename;
  /* called at first time reg. of the irq */
  unsigned int (*startup)(unsigned int irq);
  /* called when all handlers on the irq unreg'ed */
  void (*shutdown)(unsigned int irq);

  void (*enable)(unsigned int irq); /* enable the specified IRQ */
  void (*disable)(unsigned int irq); /* disable the specified IRQ */
  void (*ack)(unsigned int irq);     /* ack. (may disable) the received IRQ */
  void (*end)(unsigned int irq);     /* called at termination of IRQ handler */
  void (*set_affinity)(unsigned int irq, cpumask_t dest);
};
```

# i8259A interrupt controller

- i8259A is the classic interrupt controller on x86

```
static struct hw_interrupt_type i8259A_irq_type = {
        "XT-PIC",
        startup_8259A_irq,
        shutdown_8259A_irq,
        enable_8259A_irq,
        disable_8259A_irq,
        mask_and_ack_8259A,
        end_8259A_irq,
        NULL
};
```

- mask_and_ack_8259A() acknowledges the interrupt on the PIC and **also disables the IRQ line**
- end_8259A_irq() **re-enables the IRQ line**

# irqaction

- Multiple devices can share a single IRQ; each irqaction refers to a specific hardware device and its interrupt handler

```
struct irqaction {
  /* Points to the interrupt service routine for an I/O device */
  irqreturn_t (*handler)(int, void *, struct pt_regs *);
  /* Describes the relationships between the IRQ line and the I/O device */
  unsigned long flags;
  cpumask_t mask;
  /* the name of the device, shown in /proc/interrupts */
  const char *name;
  /* a private field for the device driver */
  void *dev_id;
  /* points to next irqaction which shared the same IRQ line */
  struct irqaction *next;
  int irq;                          /* IRQ number */
  struct proc_dir_entry *dir;
};
```

# Registering Interrupt Handler

- Requesting to be invoked when a specific IRQ is signaled

```
int request_irq( unsigned int irq, irq_handler_t *handler, long irqflags,
         const char* devname, void *dev_id)
```

- irqflags
  - SA_INTERRUPT: This is a fast interrupt. All local IRQs are disabled during handler execution
  - SA_SAMPLE_RANDOM: The timing of interrupts from this device are fed to kernel entropy pool. This is for kernel random number generator
  - SA_SHIRQ: the IRQ line can be shared among multiple devices
- devname: the name of the device used by /proc/interrupts
- dev_id
  - The unique identifier of a handler for a shared IRQ
  - The argument passed to the registered handler (E.g. private structure or device number of the device driver)
  - Can be NULL only if the IRQ is not shared

# Unregistering interrupt handler

- Unregister a specified interrupt handler and disable the given IRQ line if this is the last handler on the line.

```
int free_irq( unsigned int irq, void *dev_id)
```

- If the specified IRQ is shared, the handler identified by the dev_id is unregistered

# /proc/interrupts

IRQ number

The number of interrupts signaled since system boots

The name of interrupt controller

The name of interrupt handler

```
[josephl@sirius josephl]$ cat /proc/interrupts
           CPU0
  0: 3063864340          XT-PIC  timer
  1:          8          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  7:          0          XT-PIC  parport0
  8:          1          XT-PIC  rtc
  9:          0          XT-PIC  acpi
 10:    3557288          XT-PIC  eth0
 11:          0          XT-PIC  uhci_hcd, uhci_hcd, CMI8738
 14:    3148863          XT-PIC  ide0
 15:         13          XT-PIC  ide1
NMI:          0
ERR:          0
```

# Probing Interrupt Line

- Problem to solve
  - Fail to register interrupt handler because of not knowing which interrupt line the device has been assigned to
  - Rarely to use on embedded systems or for PCI devices
- Probing procedure
  - Clear and/or mask the device internal interrupt
  - Enable CPU interrupt
  - mask = probe_irq_on()
    - return a bit mask of unallocated interrupts
  - Enable device's interrupt and make it to trigger an interrupt
  - Busy waiting for a while allowing the expected interrupt to be signaled
  - irqs = probe_irq_off(mask)
    - Returns the number of the IRQ that was signaled
    - If no intterrupt occurred, 0 is returned; if more than 1 interrupt occurred, a negative value is returned
  - Service the device and clear pending interrupt

# Writing an Interrupt Handler

- Handler prototype

```
int irqreturn_t handler(int irq, void *dev_id, struct pt_regs *regs);
```

  - dev_id: the dev_id you register at request_irq()
  - pt_regs: value of registers before being interrupted
- Return value
  - IRQ_NONE: the handler cannot handle it; the originator may be other devices sharing the same IRQ line
  - IRQ_HANDLED: the interrupt is serviced by the handler
  - IRQ_RETVAL(x): if x is nonzero, return IRQ_HANDLED; otherwise, return IRQ_NONE
- Interrupt handler is not reentrant; while it is executing:
  - its IRQ line is disabled on PIC
  - IRQ_INPROGRESS flag prevents other CPU from executing it

# Interrupt handler sharing IRQ

- To share an IRQ with other device, you must
  - register_irq() with SA_SHIRQ flag
    - The registration fails if other handler already register the same IRQ without SA_SHIRQ flag
  - The dev_id argument must be unique to each handler
  - The interrupt handler must be able to find out whether its device actually generate an interrupt
    - Hardware must provide a status register for inquiry

# Interrupt Handler Example

```
static ata_index_t do_ide_setup_pci_device (struct pci_dev *dev, ...) {
   hwif->irq = dev->irq;
}
```

```
#define ide_request_irq(irq,hand,flg,dev,id)  \
   request_irq((irq),(hand),(flg),(dev),(id))
static int init_irq (ide_hwif_t *hwif) {
   int sa = IDE_CHIPSET_IS_PCI(hwif->chipset)?SA_SHIRQ:SA_INTERRUPT;
   ide_request_irq(hwif->irq, &ide_intr, sa, hwif->name, hwgroup);
}
```

```
irqreturn_t ide_intr (int irq, void *dev_id, struct pt_regs *regs) {
   ide_hwgroup_t *hwgroup = (ide_hwgroup_t *)dev_id;
   ide_drive_t *drive = choose_drive(hwgroup);
   struct request *rq;

   rq = elv_next_request(drive->queue);
   start_request(drive, rq);
   return IRQ_HANDLED;
}
```

# Interrupt Context

- Context
  - The execution environments of a piece of code
- Process context
  - Kernel is executing on behalf of a process. E.g. executing a system call.
  - Because of process management mechanisms, code in process context can sleep or be blocked
- Interrupt context
  - Time critical; it must finish its job quickly because it may interrupts some real-time job (may be a process or another interrupt handler)
  - No backing process; interrupted process context cannot be used
  - Code in interrupt context cannot sleep or be blocked (i.e. you cannot call some kernel functions that may sleep)
  - Configurable stack: dedicated interrupt stack (4K) or sharing the kernel stack of interrupted process (<8K)
  - Both interrupt handlers and bottom halves (softirq, tasklet) run in interrupt context

## Implementation of Interrupt Handling -- do_IRQ()

- Interrupt context is not preemptive; preemption are disabled by increasing preempt_count
- Process softirq only when are not in interrupt context
  - Nested execution of interrupt handlers is possible

```
#define HARDIRQ_OFFSET          (1UL << HARDIRQ_SHIFT)
# define IRQ_EXIT_OFFSET        (HARDIRQ_OFFSET-1)
#define irq_enter()             (preempt_count() += HARDIRQ_OFFSET)
void irq_exit(void) {
        preempt_count() -= IRQ_EXIT_OFFSET;
        if (!in_interrupt() && local_softirq_pending()) do_softirq();
        preempt_enable_no_resched();
}
fastcall unsigned int do_IRQ(struct pt_regs *regs) {
        int irq = regs->orig_eax & 0xff;
        irq_enter();
        __do_IRQ(irq, regs);
        irq_exit();
        return 1;
}
```

## Implementation of Interrupt Handling -- __do_IRQ()

- IRQ Probing
  - probe_irq_on() set IRQ_WAITING for all unallocated IRQs
  - IRQ_WAITING flag is cleared when interrupt signals
  - probe_irq_off() checks this flag to find the IRQ number of expected interrupt

```
fastcall unsigned int __do_IRQ(unsigned int irq, struct pt_regs *regs)
{
        irq_desc_t *desc = irq_desc + irq;
        struct irqaction * action;
        unsigned int status;

         /* avoid concurrent execution of the same IRQ */
        spin_lock(&desc->lock);
        desc->handler->ack(irq); /* disable IRQ at PIC */
        status = desc->status & ~IRQ_WAITING;
        status |= IRQ_PENDING; /* we _want_ to handle it */
```

## Implementation of Interrupt Handling -- __do_IRQ()

- IRQ_INPROGRESS flag prevents handlers of the same IRQ from concurrent execution

```
        action = NULL;
        if (likely(!(status & (IRQ_DISABLED | IRQ_INPROGRESS)))) {
                action = desc->action;
                status &= ~IRQ_PENDING; /* we commit to handling */
                status |= IRQ_INPROGRESS; /* we are handling it */
        }
        desc->status = status;
        /*
         * If there is no IRQ handler or it was disabled, exit early.
         * Since we set PENDING, if another processor is handling
         * a different instance of this same irq, the other processor
         * will take care of it.
         */
        if (unlikely(!action)) goto out;
```

## Implementation of Interrupt Handling -- __do_IRQ()

- Take care of other CPUs' interrupt by checking IRQ_PENDING flag

```
        for (;;) {
                irqreturn_t action_ret;
                spin_unlock(&desc->lock);
                action_ret = handle_IRQ_event(irq, regs, action);
                spin_lock(&desc->lock);
                if (likely(!(desc->status & IRQ_PENDING)))
                        break;
                desc->status &= ~IRQ_PENDING;
        }
        desc->status &= ~IRQ_INPROGRESS;
out:
        desc->handler->end(irq);  /* enable IRQ at PIC */
        spin_unlock(&desc->lock);
        return 1;
}
```

## Implementation of Interrupt Handling -- handle_IRQ_event()

- Invoke all registered handlers of the IRQ line since kernel do not know the origin of the signaled interrupt

```
fastcall int handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
                                        struct irqaction *action) {
        int ret, retval = 0, status = 0;

        if (!(action->flags & SA_INTERRUPT))
                local_irq_enable();        /* fast interrupt */
        do {
                ret = action->handler(irq, action->dev_id, regs);
                if (ret == IRQ_HANDLED)
                        status |= action->flags;
                retval |= ret;
                action = action->next;
        } while (action);
        if (status & SA_SAMPLE_RANDOM)
                add_interrupt_randomness(irq);
        local_irq_disable();
        return retval;
}
```

## Implementation of Interrupt Handling -- ret_from_intr()

- Before returning to the interrupted context, call schedule() for a reschedule when:
  - The kernel is returning to user space and need_resched() is true
  - The kernel is returning to kernel space and preempt_count() is zero
- The value of registers are restored and the kernel resumes whatever was interrupted

# References

- Linux Kernel Development, 2nd Edition, Robert Love, 2005
- Understanding the Linux Kernel, Bovet & Cesati, O'REILLY, 2002
- Linux 2.6.10 kernel source

# Bottom Halves and Deferred Work


**Paul Chu**
**Hao-Ran Liu**


# Bottom Halves

- Limitations of interrupt handlers
  - Handlers must be very fast to avoid disturbing others
    - Interrupting real-time tasks or other interrupt handlers
    - Run with current IRQ line disabled on PIC or plus all IRQ lines disabled on CPU (if SA_INTERRUPT is set)
  - Running in interrupt context
    - Unable to block and thus cannot use kernel facilities that sleep
- Solution: dividing into top half and bottom half
  - Top halves: interrupt handlers
    - Simple and fast, dealing with time-critical hardware tasks
    - E.g. packets transmission and receiving
  - Bottom halves
    - Deferring work to a later point where interrupts can be enabled
    - Processing time-consuming and maybe software-only tasks
    - E.g. network protocols processing

## Bottom Halves in Kernel 2.6

- Softirqs
  - Compile-time objects; no dynamic creation and deletion
  - At most 32 softirqs
  - Reentrant at different CPUs; no serialization
  - Run in interrupt context
- Tasklets
  - Capable of dynamic creation and deletion; no upper limit
  - Built upon two softirqs: high and low priority
  - A tasklet is executed by only one CPU at a time, but different tasklets may run concurrently
- Work Queue
  - Capable of dynamic creation and deletion; no upper limit
  - Built upon kernel threads; thus run in process context
  - One kernel thread per CPU; serialization is enforced by code itself

## Softirq data structure

- Compile-time pre-allocated objects

```
struct softirq_action {
        void (*action) (struct softirq_action *);
        void *data;              /* data passed to function */
}
static struct softirq_action softirq_vec[32];
irq_cpustat_t irq_stat[NR_CPUS];   /* IRQ stat. including pending softirq bit */
```

- Softirqs are designed for time-sensitive bottom-half processing; softirq with lower numerical priority execute first

| | | |
|---|---|---|
| HI_SOFTIRQ | 0 | High priority tasklets |
| TIMER_SOFTIRQ | 1 | Timer bottom half |
| NET_TX_SOFTIRQ | 2 | Sending network packets |
| NET_RX_SOFTIRQ | 3 | Processing received network packets |
| SCSI_SOFTIRQ | 4 | SCSI bottom half |
| TASKET_SOFTIRQ | 5 | Normal priority tasklets |

# Softirq handler

- Registering softirq action handler

```
void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
{
        softirq_vec[nr].data = data;
        softirq_vec[nr].action = action;
}
```

- Note
  - When a softirq is running, local bottom-half processing is disabled. But other processors can execute the same softirq at the same time
  - Softirqs are for parallel processing of bottom halves (e.g. per-processor data); if you need to prevent another instance of the same softirq from running at the same time, you should use tasklet
  - Softirq cannot sleep

# raise_softirq() – signal a softirq

```
#define __IRQ_STAT(cpu, member)         (irq_stat[cpu].member)
#define softirq_pending(cpu)        __IRQ_STAT((cpu), __softirq_pending)
#define local_softirq_pending()     softirq_pending(smp_processor_id())
#define __raise_softirq_irqoff(nr) do { \
                local_softirq_pending() |= 1UL << (nr); } while (0)
/* This function must run with irqs disabled! */
inline fastcall void raise_softirq_irqoff(unsigned int nr) {
        __raise_softirq_irqoff(nr);
        /* If we're in an interrupt or softirq, we will
         * actually run the softirq once we return from
         * the irq or softirq. Otherwise we wake up ksoftirqd
         * to make sure we schedule the softirq soon.
         */
        if (!in_interrupt()) wakeup_softirqd();
}
void fastcall raise_softirq(unsigned int nr) {
        unsigned long flags;
        local_irq_save(flags);
        raise_softirq_irqoff(nr);
        local_irq_restore(flags);
}
```

## Execution of softirqs (do_softirq())

- Softirqs are executed in these places
  - After processing the last hardware interrupt
  - In the ksoftirqd kernel thread
  - By any code that explicitly checks for and executes pending softirqs
  - At the end of a critical section with local_bh_enable() or spin_unlock_bh()

## Why ksoftirqd?

- Softirq dilemma
  - Softirq can be raised or self-reactivated at high frequency
  - User processes starved if all pending softirqs were processed first
  - Leave reactivated softirqs until the next hardware interrupt is not acceptable especially if the system is idle
- Solution
  - One “ksoftirqd/n” thread per CPU
  - Only process reactivated softirqs a limited number of times; the rest are processed in the ksoftirqd kernel thread
  - Run ksoftirqd with lowest priority to ensure it does not block any important processes

# do_softirq()

- do_softirq() is called by
  - irq_exit()
  - ksoftirqd()
  - local_bh_enable()

```
asmlinkage void do_softirq(void) {
        __u32 pending;
        unsigned long flags;

        /* process softirq only when outside hardirq or softirq context */
        if (in_interrupt()) return;

        /* Reading and resetting pending bitmask require IRQs off */
        local_irq_save(flags);
        pending = local_softirq_pending();
        if (pending) __do_softirq();
        local_irq_restore(flags);
}
```

# __do_softirq()

- Process softirqs for limited number of rounds; wake up ksoftirqd to process the rest if any
- Note: local_bh_disable() must precede local_irq_enable()

```
/* We restart softirq processing MAX_SOFTIRQ_RESTART times,
 * and we fall back to softirqd after that.
 * This number has been established via experimentation.
 * The two things to balance is latency against fairness -
 * we want to handle softirqs as soon as possible, but they
 * should not be able to lock up the box.
 */
#define MAX_SOFTIRQ_RESTART 10
asmlinkage void __do_softirq(void) {
        struct softirq_action *h;
        __u32 pending;
        int max_restart = MAX_SOFTIRQ_RESTART;
        int cpu;

        /* interrupts already disabled */
        pending = local_softirq_pending();
        /* we enter softirq context, disable softirq processing */
        local_bh_disable();
        cpu = smp_processor_id();
```

## __do_softirq()

```
restart:
        /* Reset the pending bitmask before enabling irqs */
        local_softirq_pending() = 0;
        local_irq_enable();

        /* softirq run with all interrupts enabled */
        h = softirq_vec;
        /* execute softirq handlers in ascending order of softirq no. */
        do {
                if (pending & 1) {
                        h->action(h);
                        rcu_bh_qsctr_inc(cpu);
                }
                h++;
                pending >>= 1;
        } while (pending);

        local_irq_disable();
        /* check if there is any reactivated softirqs */
        pending = local_softirq_pending();
        if (pending && --max_restart) goto restart;
        /* wake up softirqd to process the rest */
        if (pending) wakeup_softirqd();
        __local_bh_enable();        /* we leave softirq context */
}
```

## Implementation of ksoftirqd

- The ksoftirqd is waked up at two points:
  - do_softirq(), raise_softirq()
- Yield the CPU after each do_softirq()

```
static int ksoftirqd() {
        set_user_nice(current, 19);  /* set process to lowest priority */
        current->flags |= PF_NOFREEZE;
        set_current_state(TASK_INTERRUPTIBLE);
        while (!kthread_should_stop()) {
                if (!local_softirq_pending()) schedule();
                __set_current_state(TASK_RUNNING);
                while (local_softirq_pending()) {
                        preempt_disable();
                        do_softirq();
                        preempt_enable();
                        cond_resched();
                }
                set_current_state(TASK_INTERRUPTIBLE);
        }
        __set_current_state(TASK_RUNNING);
        return 0;
}
```

# Softirq Example – SCSI softirq

```
static int __init init_scsi(void)
{
    ... ... ...
    open_softirq(SCSI_SOFTIRQ, scsi_softirq, NULL);
    ... ... ...
}
```

```
int scsi_dispatch_cmd(struct scsi_cmnd *cmd)
{
    struct Scsi_Host *host = cmd->device->host;
    ... ... ...
    scsi_add_timer(cmd, cmd->timeout_per_command, scsi_times_out);
    rtn = host->hostt->queuecommand(cmd, scsi_done);
    ... ... ...
}
```

```
irqreturn_t scsi_hba_isr(int irq, void *dev_id, struct pt_regs *regs)
{
    scsi_hba_host_t *ha = (scsi_hba_host_t*) dev_id;

    ha->completed_cmd->done();  // scsi_done()
}
```

```
void scsi_done(struct scsi_cmnd *cmd)
{
    scsi_delete_timer(cmd);

    raise_softirq_irqoff(SCSI_SOFTIRQ);
}
```

# Tasklets

- Tasklet is the most common choice for bottom-half processing

```
struct tasklet_struct {
        struct tasklet_struct *next;   // for linked list of tasklets
        unsigned long state;           // state of the tasklet
        atomic_t count;                // reference counter (nonzero=disabled)
        void (*func)(unsigned long);   // tasklet handler function
        unsigned long data;            // argument to tasklet handler
}
```

- Pending tasklets form lists in softirqs
  - HI_SOFTIRQ: tasklet_hi_vec list → tasklet_hi_action()
  - TASKET_SOFTIRQ: tasklet_vec list → tasklet_action()
- Tasklet state
  - Tasklet is scheduled to run (TASKLET_STATE_SCHED)
  - Tasklet is running on a CPU (TASKLET_STATE_RUN) (SMP only)
  - Tasklet is idle (0)

## Using Tasklets

➢ Creating a tasklet

```
DECLARE_TASKLET(tasklet, tasklet_handler, data);
DECLARE_TASKLET_DISABLED(tasklet , tasklet_handler, data);
```

```
tasklist_init(tasklet, tasklet_handler, data); // dynamic creation
```

➢ Tasklet handler prototype

```
void tasklet_handler(unsigned long data); // runs in interrupt context
```

– Tasklet in essence is softirq; it cannot sleep too
– The same tasklet cannot run concurrently on two CPUs
– Different tasklets can run concurrently on different CPUs

## Using Tasklets

➢ Schedule a tasklet

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
        unsigned long flags;

        if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state)) {
                /* prevent race condition if IRQ handlers use the list too */
                local_irq_save(flags);
                t->next = __get_cpu_var(tasklet_vec).list;
                __get_cpu_var(tasklet_vec).list = t;
                /* notify a softirq, allowing do_softirq() to execute */
                raise_softirq_irqoff(TASKLET_SOFTIRQ);
                local_irq_restore(flags);
        } /* the CPU that schedules the tasklet will do the job */
}
```

– tasklet_hi_schedule() for HI_SOFTIRQ

➢ What if you reschedule a tasklet?
– If it has been scheduled but is not run yet, it runs only once
– If it is running, it will run again

## Using Tasklets

- Enable and disable a tasklet
  - tasklet_disable() / tasklet_enable() are used to prevent race condition if your code share data with tasklets

```
static inline void tasklet_unlock_wait(struct tasklet_struct *t) {
        while (test_bit(TASKLET_STATE_RUN, &(t)->state)) { barrier(); }
}
static inline void tasklet_disable_nosync(struct tasklet_struct *t) {
        atomic_inc(&t->count);
        smp_mb__after_atomic_inc();
}
static inline void tasklet_disable(struct tasklet_struct *t) {
        tasklet_disable_nosync(t);
        tasklet_unlock_wait(t);
        smp_mb();
}
static inline void tasklet_enable(struct tasklet_struct *t) {
        smp_mb__before_atomic_dec();
        atomic_dec(&t->count);
}
```

## Using Tasklets

- Wait until a tasklet is not scheduled and not running
  - Useful when dealing with a tasklet that often reschedule itself and the driver is going to be unloaded

```
void tasklet_kill(struct tasklet_struct *t)
{
        if (in_interrupt())
                printk("Attempt to kill tasklet from interrupt\n");

        while (test_and_set_bit(TASKLET_STATE_SCHED, &t->state)) {
                do
                        yield();
                while (test_bit(TASKLET_STATE_SCHED, &t->state));
        }
        tasklet_unlock_wait(t);
        clear_bit(TASKLET_STATE_SCHED, &t->state);
}
```

## Running tasklets

- High priority tasklets and normal tasklets are registered when kernel boots

```
void __init softirq_init(void)
{
        open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
        open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
}
```

- tasklet_action() is run when do_softirq() is executed and TASKLET_SOFTIRQ is raised

```
static inline int tasklet_trylock(struct tasklet_struct *t) {
        return !test_and_set_bit(TASKLET_STATE_RUN, &(t)->state);
}

static inline void tasklet_unlock(struct tasklet_struct *t) {
        smp_mb__before_clear_bit();
        clear_bit(TASKLET_STATE_RUN, &(t)->state);
}
```

## Running tasklets

```
static void tasklet_action(struct softirq_action *a) {
  struct tasklet_struct *list;

  local_irq_disable();      /* get the list of pending tasklets out */
  list = __get_cpu_var(tasklet_vec).list;
  __get_cpu_var(tasklet_vec).list = NULL;
  local_irq_enable();

  while (list) {
    struct tasklet_struct *t = list;
    list = list->next;
    if (tasklet_trylock(t)) {  /* tasklet may be running already */
      if (!atomic_read(&t->count)) { /* tasklet may be disabled */
        if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state)) BUG();
        t->func(t->data);
        tasklet_unlock(t);
        continue;
      }
      tasklet_unlock(t);
    }
    local_irq_disable();    /* put non-runnable tasklet back to pending list */
    t->next = __get_cpu_var(tasklet_vec).list;
    __get_cpu_var(tasklet_vec).list = t;
    __raise_softirq_irqoff(TASKLET_SOFTIRQ);
    local_irq_enable();
  }
}
```

# Work Queues

- Work queue advantages
  - Execute a job needed to be in process context
  - But do the job without bothering maintaining kernel threads
- Work queues
  - One work queue has one worker thread per CPU
  - One work queue has one work list per CPU
- Worker threads
  - Worker threads retrieve works from work list of the same CPU and execute the handler on behalf of the worker thread
  - Work is executed again if it is rescheduled
  - Running in process context, code may sleep but will pause all works in the worker thread on the same CPU
  - Default worker threads: “events /n”; create your own worker thread if your work is time sensitive or heavy loaded

# Work queue and per-CPU workqueue

- Work queue

```
struct workqueue_struct {
        struct cpu_workqueue_struct cpu_wq[NR_CPUS];
        const char *name;
        struct list_head list;        /* Empty if single thread */
};
```

- Per-cpu workqueue

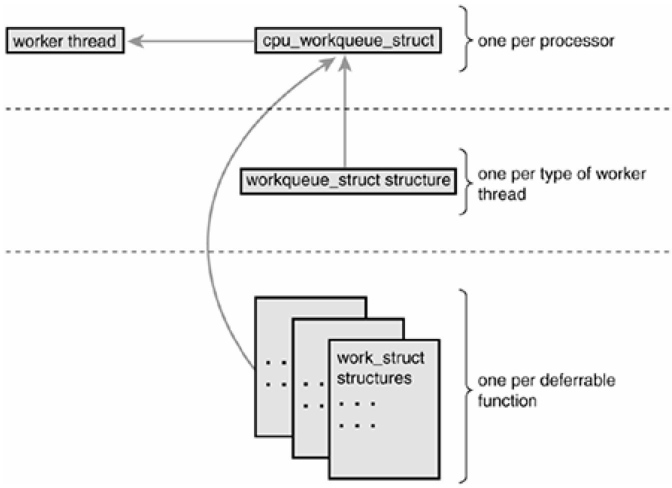
```
struct cpu_workqueue_struct {
  spinlock_t lock;
  long remove_sequence;           /* Least-recently added (next to run) */
  long insert_sequence;           /* Next to add */
  struct list_head worklist;      /* list of works queued on this CPU */
  wait_queue_head_t more_work;    /* waiting queue for worker thread */
  wait_queue_head_t work_done;    /* for threads waiting queue flush */
  struct workqueue_struct *wq;    /* point to parent structure */
  task_t *thread;             /* associated thread's task_struct */
  int run_depth;              /* Detect run_workqueue() recursion depth */
} ____cacheline_aligned;
```

# Work

➢ Work

```
struct work_struct {
        unsigned long pending;      // is the work pending ?
        struct list_head entry;     // for linking all work in work queue
        void (*func)(void *);       // handler function
        void *data;                 // argument to handler function
        void *wq_data;              // private data for internal use
        struct timer_list timer;    // for delayed work queues
}
```

# The relationship between work, workqueue and worker threads




* From Linux kernel development, 2nd edition, Robert Love

## Using Work Queues

- Create a work

```
DECLARE_WORK(work, work_handler, data); // static allocation
INIT_WORK(work, work_handler, data);     // dynamic creation
```

- Work handler prototype

```
void work_handler(void *data);                // run in process context
```

- Queue work with the default *events* queue

```
int fastcall schedule_work(struct work_struct *work) {
        return queue_work(keventd_wq, work);
}
```

- work is not queued with *events* queue immediately, but after some delay

```
int fastcall schedule_delayed_work(struct work_struct *work,
                                   unsigned long delay) {
        return queue_delayed_work(keventd_wq, work, delay);
}
```

## Using Work Queues

- Flush works in the *events* work queue
  - Wait (sleep) until all entries in the queue are executed
  - Does not flush or cancel delayed work

```
void flush_scheduled_work(void) {
        flush_workqueue(keventd_wq);
}
```

- To cancel a scheduled delayed work

```
/* Kill off a pending schedule_delayed_work(). Note that the work callback
 * function may still be running on return from cancel_delayed_work(). Run
 * flush_scheduled_work() to wait on it.
 */
static inline int cancel_delayed_work(struct work_struct *work) {
        int ret;
        ret = del_timer_sync(&work->timer);
        if (ret) clear_bit(0, &work->pending);
        return ret;
}
```

# Using Work Queues

- Create your own worker thread

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct * create_singlethread_workqueue(const char *name);
```

  - Example

```
static struct workqueue_struct *keventd_wq;
keventd_wq = create_workqueue("events");
```

- Queue work with your own work queue

```
int fastcall queue_work(struct workqueue_struct *wq,
                        struct work_struct *work) {
        int ret = 0, cpu = get_cpu();
        if (!test_and_set_bit(0, &work->pending)) {
                if (unlikely(is_single_threaded(wq))) cpu = 0;
                __queue_work(wq->cpu_wq + cpu, work);
                ret = 1;
        }
        put_cpu();
        return ret;
}
```

# __queue_work()

```
static void __queue_work(struct cpu_workqueue_struct *cwq,
                         struct work_struct *work) {
        unsigned long flags;
        spin_lock_irqsave(&cwq->lock, flags);
        work->wq_data = cwq;
        list_add_tail(&work->entry, &cwq->worklist);
        cwq->insert_sequence++;
        wake_up(&cwq->more_work);
        spin_unlock_irqrestore(&cwq->lock, flags);
}
```

## Using Work Queues

➢ Queue delayed work with your own work queue

```
int fastcall queue_delayed_work(struct workqueue_struct *wq,
                              struct work_struct *work, unsigned long delay) {
        struct timer_list *timer = &work->timer;
        if (!test_and_set_bit(0, &work->pending)) {
                /* This stores wq for the moment, for the timer_fn */
                work->wq_data = wq;
                timer->expires = jiffies + delay;
                timer->data = (unsigned long)work;
                timer->function = delayed_work_timer_fn;
                add_timer(timer);
                return 1;
        }
        return 0;
}
static void delayed_work_timer_fn(unsigned long __data) {
        struct work_struct *work = (struct work_struct *)__data;
        struct workqueue_struct *wq = work->wq_data;
        int cpu = smp_processor_id();
        if (unlikely(is_single_threaded(wq))) cpu = 0;
        __queue_work(wq->cpu_wq + cpu, work);
}
```

## Using Work Queues

➢ Flush your own work queue

```
/*flush_workqueue - ensure that any scheduled work has run to completion.
 * Forces execution of the workqueue and blocks until its completion. This is
 * typically used in driver shutdown handlers. This function will sample each
 * workqueue's current insert_sequence number and will sleep until the head
 * sequence is greater than or equal to that. This means that we sleep until all
 * works which were queued on entry have been handled, but we are not
 * livelocked by new incoming ones. This function used to run the workqueues
 * itself.  Now we just wait for the helper threads to do it. */
void fastcall flush_workqueue(struct workqueue_struct *wq) {
        might_sleep();
        if (is_single_threaded(wq)) {
                flush_cpu_workqueue(wq->cpu_wq + 0);
        } else {
                int cpu;
                lock_cpu_hotplug();
                for_each_online_cpu(cpu)
                        flush_cpu_workqueue(wq->cpu_wq + cpu);
                unlock_cpu_hotplug();
        }
}
```

## flush_cpu_workqueue()

```
static void flush_cpu_workqueue(struct cpu_workqueue_struct *cwq) {
        if (cwq->thread == current) {
                /* Probably keventd trying to flush its own queue.
                 * So simply run it by hand rather than deadlocking. */
                run_workqueue(cwq);
        } else {
                DEFINE_WAIT(wait);
                long sequence_needed;
                spin_lock_irq(&cwq->lock);
                sequence_needed = cwq->insert_sequence;
                while (sequence_needed - cwq->remove_sequence > 0) {
                        prepare_to_wait(&cwq->work_done, &wait,
                                        TASK_UNINTERRUPTIBLE);
                        spin_unlock_irq(&cwq->lock);
                        schedule();
                        spin_lock_irq(&cwq->lock);
                }
                finish_wait(&cwq->work_done, &wait);
                spin_unlock_irq(&cwq->lock);
        }
}
```

## Worker Thread

```
static int worker_thread(void *__cwq)
{
        struct cpu_workqueue_struct *cwq = __cwq;
        DECLARE_WAITQUEUE(wait, current);
        set_user_nice(current, -10);

        while (!kthread_should_stop()) {
                set_task_state(current, TASK_INTERRUPTIBLE);

                add_wait_queue(&cwq->more_work, &wait);
                if (list_empty(&cwq->worklist))
                        schedule();
                else
                        set_task_state(current, TASK_RUNNING);
                remove_wait_queue(&cwq->more_work, &wait);

                if (!list_empty(&cwq->worklist))
                        run_workqueue(cwq);
        }
        return 0;
}
```

## Run Work Queue

```
static inline void run_workqueue(struct cpu_workqueue_struct *cwq) {
        unsigned long flags;

        spin_lock_irqsave(&cwq->lock, flags);
        while (!list_empty(&cwq->worklist)) {
                struct work_struct *work = list_entry(cwq->worklist.next,
                                                struct work_struct, entry);
                void (*f) (void *) = work->func;
                void *data = work->data;

                list_del_init(cwq->worklist.next);
                spin_unlock_irqrestore(&cwq->lock, flags);

                BUG_ON(work->wq_data != cwq);
                clear_bit(0, &work->pending);
                f(data);

                spin_lock_irqsave(&cwq->lock, flags);
                cwq->remove_sequence++;
                wake_up(&cwq->work_done);
        }
        spin_unlock_irqrestore(&cwq->lock, flags);
}
```

## Synchronization of bottom halves

- Since softirq and tasklet executes asynchronously (when hardware interrupts return), we need to disable bottom halves to prevent race conditions if data is shared between process and softirq or tasklet
- Disabling bottom halves may not be enough, you need to obtain a lock to prevent SMP concurrency

```
#define local_bh_disable() \
        do { preempt_count() += SOFTIRQ_OFFSET; barrier(); } while (0)

void local_bh_enable(void)
{
        WARN_ON(irqs_disabled());
        /* Keep preemption disabled until we are done with
         * softirq processing: */
        preempt_count() -= SOFTIRQ_OFFSET - 1;
        if (unlikely(!in_interrupt() && local_softirq_pending())) do_softirq();
        dec_preempt_count();
        preempt_check_resched();
}
```

# Timers and Time Control

- Timers

```
timer.expires = jiffies + delay;     // set timeout value
timer.data = my_data;                // parameters to pass to handler
timer.function = my_handler;         // handler will be invoked when timeout
```

```
init_timer(timer_list)               // initialize a timer
add_timer(timer_list)                // activate the timer
mod_timer(timer_list, new_jiffies)   // update timeout value of a timer
del_timer(timer_list)                // deactivate the timer
del_timer_sync(timer_list)           // deactivate and wait till all handler done
```

- Busy looping
  - while (time_before(jiffies, expire));
- Small delays
  - udelay(usecs) / mdelay(msecs)
- Sleep till timeout
  - schedule_timeout(timeout)

# References


- Linux Kernel Development, 2nd edition, Robert Love, 2005
- Linux kernel source 2.6.10

# Linux Kernel Synchronization

Paul Chu
Hao-Ran Liu

# Term Definitions

- Critical sections
  - Code paths that access shared data
- Race condition
  - If two context access the same critical section at the same time
- Synchronization
  - We use synchronization primitives in the kernel to ensure that race conditions do not happen

## Examples of race condition

```
int a = 0, b = 0; /* a+b must be 0  always  */
```

```
int thread_one(int argc, char** argv) {
        a++;
        do_io_block(); // dead in non-preemptive kernel
        b--;
}
```

```
int thread_two(int argc, char** argv) {
        a++;
        b--; // dead in preemptive kernel or in SMP
}
```

```
int thread_x(int argc, char** argv) {
        do_actions(a, b); // asuming a+b == 0
}
```

```
int isr(int argc, char** argv) {
        a++;
        b--; // dead if other threads do not disable irq
}
```

## Source of concurrency in the Linux kernel

- Interrupt handling (pseudo concurrency)
  - Interrupt handlers
  - Bottom halves
- Kernel preemption (pseudo concurrency)
  - Cooperative: tasks invoke the scueduler for sleeping or synchronization
  - Noncooperative: one task in the kernel can preempt another when the kernel is preemptive
- Symmetrical multiprocessing
  - Kernel code can be executed by two or more processors

## Locking

- To prevent concurrency execution of a critical section
- Locking protocol
  - Acquire a lock before accessing shared data
  - If the lock is already held, you must wait
  - Release the lock after completing the access
- Where to use
  - Identify what data needs protection
  - Locks go with data, not code

## Deadlocks

- A condition when threads hold the locks that others are waiting for and they themselves are waiting for locks held by others
- Deadlock can be prevented if any of the following condition is not true
  - Mutual exclusion, Hold and wait, No preemption, Circular waiting
- Strategies
  - Enforce a specific locking order
  - Reduce the number of locks to hold at the same time
  - Prevent starvation

## Lock contention and scalability

- A highly contended lock can slow down a system's performance
  - Because a lock's job is to serialize access to a resource
  - This becomes worse when the number of processors is increased
- Solution
  - Divide a coarse lock into fine-grained lock
  - Eliminate the needs to lock by separating data
    - Per processor data

## Per-CPU variables

- Declaring one element for each CPU in the system
  - Avoid the need for synchronization to get better performance
  - A CPU can only read and modify its own element without fear
  - The element of each CPU falls on a different cache line; concurrent accesses of a per-CPU variable do not result in cache line snooping and invalidation
- Note
  - No protection against interrupts and bottom halves
  - Prone to race conditions caused by kernel preemption (hint: what if process preempted and migrated to a different CPU)

## Functions and macros for the per-CPU variables

| Macro or function name | Description |
|---|---|
| DEFINE_PER_CPU(type, name) | Statically allocates a per-CPU array called name of type data structures |
| per_cpu(name, cpu) | Selects the element for CPU cpu of the per-CPU array name |
| _ _get_cpu_var(name) | Selects the local CPU's element of the per-CPU array name |
| get_cpu_var(name) | Disables kernel preemption, then selects the local CPU's element of the per-CPU array name |
| put_cpu_var(name) | Enables kernel preemption (name is not used) |
| alloc_percpu(type) | Dynamically allocates a per-CPU array of type data structures and returns its address |
| free_percpu(pointer) | Releases a dynamically allocated per-CPU array at address pointer |
| per_cpu_ptr(pointer, cpu) | Returns the address of the element for CPU cpu of the per-CPU array at address pointer |

This table is from Understanding the Linux Kernel, 3rd Edition, 2005, Bovet & Cesati

## Atomic Operations

- Atomicity
  - Not dividable by interrupts
    - Eliminate pseudo concurrency
    - May be mimic by disabling interrupt during operations
  - Not dividable by other processors
    - Bus locking capability in hardware must be supported
- When to use
  - Sharing simple data types; e.g. integer, bits
  - No consistency requirement on two or more variables
  - Better efficiency than complicated locking mechanisms
- As the building blocks of complicated locking mechanisms

## Overhead of atomic operations

- Disable/enable local interrupt or lock/unlock bus is not without cost
  - Implicit memory barrier cause CPU to flush its pipeline
- Data caches invalidation of frequently modified variables shared between different CPUs
- These are the overheads RCU avoids

## Atomic Operations in Linux kernel

- Atomic integer operations
  - atomic_t ensures variables not be processed by non-atomic routines

```
ATOMIC_INIT(int i)
int atomic_read(atomic_t *v) / void atomic_set(atomic_t *v, int i)
void atomic_add(int i, atomic_t *v) / void atomic_sub(int i, atomic_t *v)
void atomic_inc(v) / void atomic_dec(v)
int atomic_dec_and_test(atomic_t *v) / int acomic_inc_and_test (atomic_t *v)
atomic_add_negative(int i, atomic_t *v)
```

- Atomic bitwise operations

```
int set_bit(int nr, void *addr) / int clear_bit(int nr, void *addr)
int test_bit(int nr, void *addr)
int change_bit(int bit, void *addr)
test_and_set_bit(int nr, void *addr) / test_and_clear_bit(int nr, void *addr)
test_and_change_bit(int nr, void *addr)
```

  - Non-atomic operations: __test_bit(), and etc.
  - Local-only atomic operations: local_add(local_t), and etc.
  - The only portable way to set a specific bit (endianess)

# Atomic operations on x86

- The processor use 3 interdependent mechanisms for carrying out locked atomic operations
  - Guaranteed atomic operations
    - Reading or writing a byte, a word aligned on 16-bit boundary, a doubleword aligned on 32-bit boundary
  - Bus locking
    - Automatic locking: accessing memory with XCHG instruction
    - Software-controlled locking: use the prefix LOCK with certain instructions
  - Cache coherency protocols
    - The area of memory being locked might be cached in the processor

# Implementing atomic operations on x86

- Accessing a doubleword is guaranteed to be atomic

```
#ifdef CONFIG_SMP
#define LOCK "lock ; "
#else
#define LOCK ""
#endif

#define atomic_read(v)          ((v)->counter)

#define atomic_set(v,i)         (((v)->counter) = (i))

static __inline__ void atomic_add(int i, atomic_t *v)
{
        __asm__ __volatile__(
                LOCK "addl %1,%0"
                :"=m" (v->counter)
                :"ir" (i), "m" (v->counter));
}
```

## Memory barriers

- Both compilers and processors reorder instructions to get better runtime performance
  - Compilers reorder instructions at compile time (e.g. to increase the throughput of pipelining)
  - CPUs reorder instructions at runtime (e.g. to fill execution units in a superscalar processor)
- Sometimes, we need memory read (load) and write (store) issued in the order specified in our program
  - Issuing I/O commands to hardware
  - Synchronized threads running on different processors
  - Protecting instructions in a critical section from bleeding out
- A memory barrier primitive ensures that the operations placed before the primitive are finished before starting the operations placed after the primitive

## Memory barriers in Linux kernel

- Compiler barrier
  - barrier(): prevents the compiler from optimizing stores/loads across it
- Hardware barrier + compiler barrier
  - read_barrier_depends(): prevents data-dependent loads from being reordered across it
  - rmb(): prevents loads from being reorder across it
  - wmb(): prevents stores from being reordered across it
  - mb(): prevents loads or stores from being reordered across it
- These macros provide a full barrier on SMP, and a compiler barrier on UP*; used to prevent race conditions only in SMP
  - smp_read_barrier_depends()
  - smp_rmb()
  - smp_wmb()
  - smp_mb()

* Memory order observed by processes on the same CPU is guaranteed by processor (*precise interrupt*)
Refer to "Computer Architecture, A Quantitative Approach" for more detailed information

# Example of using memory barriers

- Without memory barriers, it is possible that c gets the new value of b, whereas d receives the old value of a

| Thread 1 | Thread 2 |
|---|---|
| a = 3;<br>mb();<br>b = 4; | c = b;<br>rmb();<br>d = a; |

- Without memory barriers, it is possible for b to be set to pp before pp was set to p

| Thread 1 | Thread 2 |
|---|---|
| a = 3;<br>mb();<br>p = &a; | pp = p;<br>read_barrier_depends();<br>b = *pp; |

# Memory ordering of various CPUs

- x86 does not support out-of-order stores (except few string operations)
- Atomic instructions on x86 comes with implicit memory barriers.

| | Loads reordered after loads? | Loads reordered after stores? | Stores reordered after stores? | Stores reordered after loads? | Atomic instructions reordered with loads? | Atomic instructions reordered with stores? | Dependent loads reordered? |
|---|---|---|---|---|---|---|---|
| Alpha | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| AMD64 | Yes | | | Yes | | | |
| IA64 | Yes | Yes | Yes | Yes | Yes | Yes | |
| PowerPC | Yes | Yes | Yes | Yes | Yes | Yes | |
| x86 | Yes | | | Yes | | | |

Copy from "Memory ordering in modern processors, part I", Paul McKenny, Linux Journal #136, 2005

## Memory barriers instructions for x86

- Serializing instructions (implicit memory barriers)
  - All instructions that operate on I/O ports
  - All instructions prefixed by the *lock* byte
  - All instructions that write into control registers, system registers or debug registers (e.g. cli and sti)
  - A few special instructions (invd, invlpg, wbinvd, iret …)
- Memory ordering instructions (explicit memory barriers)
  - lfence, serializing all load operations
  - sfence, serializing all store operations
  - mfence, serializing all load and store operations

## Implementing memory barriers on x86

- The __volatile __ tells gcc that the instruction has important side effects. Do not delete the instruction or reschedule other instructions across it
- The "memory" tells gcc that the instruction changes memory, so that it does not cache variables in registers across the instruction.

```
#define barrier() __asm__ __volatile__("": :"memory")
#define mb() alternative("lock; addl $0,0(%%esp)", "mfence", X86_...)
#define rmb() alternative("lock; addl $0,0(%%esp)", "lfence", X86_...)
#define read_barrier_depends()     do { } while(0)
#define wmb()    __asm__ __volatile__ ("": :"memory")

#ifdef CONFIG_SMP
#define smp_mb()          mb()
#define smp_rmb()         rmb()
#define smp_wmb()         wmb()
#define smp_read_barrier_depends()        read_barrier_depends()
#else
#define smp_mb()          barrier()
#define smp_rmb()         barrier()
#define smp_wmb()         barrier()
#define smp_read_barrier_depends()        do { } while(0)
#endif
```

# Disabling Interrupts

- Disable interrupts
  - Eliminate pseudo concurrency on single processor
    - Coupled with spinlock if sharing data between multiple processors
  - Lead to longer interrupt latency
  - When to use
    - Normal path shares data with interrupt handlers
    - Interrupt handlers share data with other interrupt handlers
    - One interrupt handler for different IRQs share data within it; the interrupt handler might be reentrant
    - Shorter duration of critical sections
  - Need not to use
    - Sharing data within an interrupt handler of a IRQ; interrupt handler of a IRQ is not reentrant in SMP

# Interrupt Control Routines

- local_irq_disable() / local_irq_enable()
  - Disable or enable all interrupts of current CPU
- local_irq_save(flags) / local_irq_restore(flags)
  - Save current IRQ state and disable IRQ
  - Restore IRQ state instead of enabling it directly
  - When a routine is reached both with and without interrupts enabled
- disable_irq(irq) / enable_irq(irq)
  - Disable or enable a specific IRQ line for all CPUs
  - Return only when the specific handler is not being executed
- disable_irq_nosync(unsigned int irq)
  - Disable a specific IRQ line without waiting it (SMP)
- State checking
  - irqs_disabled(): if all local IRQs are disabled
  - in_interrupt(): if being executed in interrupt context
  - in_irq(): if being executed in an interrupt handler

## Disabling preemption

- Context switches can happened at any time with a preemptive kernel even when a process is in the kernel mode
  - Critical sections must disable preemption to avoid race condition
- preempt_disable() / preempt_enable() is nestable; kernel maintain a preempt count for every processes.
- Preemption-related functions

```
#define preempt_count()  (current_thread_info()->preempt_count)
#define preempt_disable() \
do {      inc_preempt_count(); \
          barrier(); \
} while (0)
#define preempt_enable_no_resched() \
do {      barrier(); \
          dec_preempt_count(); \
} while (0)
#define preempt_enable() \
do {      preempt_enable_no_resched(); \
          preempt_check_resched(); \
} while (0)
```

## Spin Locks

- Disabling interrupts cannot stop other processors
- Spin lock busy waits a shared lock to be release
  - Lightweight single-holder lock, all other threads will be busy looping to poll the shared lock
- When it's UP system
  - Markers to disable kernel preemption (scheduling latency)
  - Or, be removed at compile time if no kernel preemption
- When to use
  - Sharing data among threads running on processors of SMP system
  - Sharing data among preempt-able kernel threads
  - Shorter duration of critical sections

## Spin lock functions

- spin_lock_init()
  - Runtime initializing given spinlock_t
- spin_lock() / spin_unlock()
  - Acquire or release given lock
- spin_lock_irq() / spin_unlock_irq()
  - Disable local interrupts and acquire given lock
  - Release given lock and enable local interrupts
- spin_lock_irqsave() / spin_unlock_irqrestore()
  - Save current state of local interrupts, disable local interrupts and acquire given lock
  - Release given lock and restore local interrupts to given previous state
- spin_trylock()
  - Try to acquire given lock; if unavailable, returns zero
- spin_islocked()
  - Return nonzero if the given lock is currently acquired

## Spin lock implementation on x86

- Implementation for SMP and preemptive kernel

```
#define spin_lock(lock)                  _spin_lock(lock)
void __lockfunc _spin_lock(spinlock_t *lock) {
        preempt_disable();
        if (unlikely(!_raw_spin_trylock(lock)))
                __preempt_spin_lock(lock);
}
```

- *xchgb* will lock the bus; it acts as a memory barrier

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

## Spin lock implementation on x86

```
#define spin_is_locked(x) (*(volatile signed char *)(&(x)->lock) <= 0)

/* This could be a long-held lock.  If another CPU holds it for a long time,
 * and that CPU is not asked to reschedule then *this* CPU will spin on the
 * lock for a long time, even if *this* CPU is asked to reschedule.
 * So what we do here, in the slow (contended) path is to spin on the lock by
 * hand while permitting preemption. */
static inline void __preempt_spin_lock(spinlock_t *lock)  {
        if (preempt_count() > 1) {
                _raw_spin_lock(lock);
                return;
        }
        do {
                preempt_enable();
                while (spin_is_locked(lock))
                        cpu_relax();
                preempt_disable();
        } while (!_raw_spin_trylock(lock));
}
```

## Spin lock implementation on x86

```
#define spin_lock_string \
        "\n1:\t" \
        "lock ; decb %0\n\t" \          /* lock bus, memory barrier */
        "jns 3f\n" \                    /* jump if we acquire the lock */
        "2:\t" \                        /* spin lock loop below */
        "rep;nop\n\t" \                 /* = cpu_relax() */
        "cmpb $0,%0\n\t" \              /* check if lock is available */
        "jle 2b\n\t" \                  /* jump if lock not available */
        "jmp 1b\n" \                    /* lock available, try lock again */
        "3:\n\t"                        /* lock is acquired */

static inline void _raw_spin_lock(spinlock_t *lock) {
        __asm__ __volatile__(
                spin_lock_string
                :"=m" (lock->lock) : : "memory");
}
```

# Spin lock implementation on x86

```
#define spin_unlock_string \
        "movb $1,%0" \
                :"=m" (lock->lock) : : "memory"

static inline void _raw_spin_unlock(spinlock_t *lock) {
        __asm__ __volatile__(
                spin_unlock_string
        );
}
#define spin_unlock(lock) _spin_unlock(lock)
void __lockfunc _spin_unlock(spinlock_t *lock) {
        _raw_spin_unlock(lock);
        preempt_enable();
}
```

Is lock prefix needed?

- Conclusion about spin lock
  - Spin lock implementation is composed of atomic operations, memory barriers and (preemption/bottom halve/interrupt) disabling

# Reader-writer spin locks

- Multiple concurrent accesses to shared data are read-only
- Multiple read locks can be granted, but write lock is allowed only when there is no any lock.
- Favor readers over writers: writers starvation
- Operations
  - rw_lock_init(), rw_is_locked()
  - read_lock(), read_lock_irq(), and so on.
  - write_lock(), write_lock_irq(), and so on.

## Sequence Locks

- What are sequence locks (seqlock_t)
  - Similar to reader-writer spin locks
  - But favor writers over readers
  - A writer never waits unless another writer is active, while readers may be forced to read the same data several times until it gets a valid copy
- How to use
  - Writers

```
write_seqlock(seqlock);              // acquire lock & increment count
/* write the shared data */
write_sequnlock(seqlock);            // release lock & increment again
```

  - Readers

```
do {
        seq = read_seqbegin(seqlock);    // get sequence count
        /* read the shared data */
} while (read_seqretry(seqlock, seq);    // check if write lock is obtained
```

## Sequence lock implementation

- Sequence lock data structure

```
typedef struct {
        unsigned sequence;
        spinlock_t lock;
} seqlock_t;
```

- Sequence lock functions for writers

```
static inline void write_seqlock(seqlock_t *sl) {
        spin_lock(&sl->lock);
        ++sl->sequence;
        smp_wmb();
}
static inline void write_sequnlock(seqlock_t *sl) {
        smp_wmb();
        sl->sequence++;
        spin_unlock(&sl->lock);
}
```

# Sequence lock implementation

- Sequence lock functions for readers

```
static inline unsigned read_seqbegin(const seqlock_t *sl)
{
        unsigned ret = sl->sequence;
        smp_rmb();
        return ret;
}
static inline int read_seqretry(const seqlock_t *sl, unsigned iv)
{
        smp_rmb();
        return (iv & 1) | (sl->sequence ^ iv);
}
```

Reader needs to retry when the sequence number is odd or when the sequence number differs from previous saved value

# *volatile* keyword v.s. memory barriers

- Small quiz
  - ***jiffies*** *variable in Linux kernel is declared with* ***volatile*** *keyword. The keyword tells the compiler that the value of this variable may change at any time and disables compiler optimization on it.*
  - *The question is: Why is not the field* ***sequence*** *in* ***seqlock_t*** *declared as* ***volatile****? (hint: the purpose of* ***smp_rmb()*** *and* ***smp_wmb()****)*

# Semaphores

- Sleeping locks
  - The locking thread is put to sleep and be woken up when the lock is released
- When to use
  - The Lock is to be held for a long time
    - the overhead of sleeping outweigh the lock hold time
  - Can be used only in process context
  - Shared data among threads
- Notes
  - Do not hold spin lock before acquire a semaphore
  - Thread holding semaphore might be preempted
- Types of semaphores
  - Binary semaphore and mutex
  - Counting semaphore
    - More than one semaphore holders are allowed
    - When the shared resources are more than one

# More About Semaphores

- Semaphores operations

```
sema_init (semaphore*, int)
init_MUTEX(semaphore*) / init_MUTEX_LOCKED(semaphore*)
down(semaphore*) / down_interruptible(semaphore*)
down_trylock(semaphore*)
up(semaphore*)
```

- Reader-writer semaphores (rw_semaphore)
  - Same as reader-writer spin locks
  - All uninterruptible sleep
  - Converting acquired write lock to read lock

```
init_rwsem(rw_semaphore* )
down_read(rw_semaphore*) / down_write(rw_semaphore*)
up_read(rw_semaphore*) / up_write(rw_semaphore *)
down_read_trylock(rw_semaphore*) / down_write_trylock(rw_semaphore*)
downgrade_write(rw_semaphore *)
```

## Other Synchronization Controls

- Bottom halves disabling
  - Sharing data with softirqs and tasklets
    - local_bh_disable() / local_bh_enable()
    - spin_lock_bh() / spin_unlock_bh()
- Completion variables
  - One thread waits another thread to complete some tasks
    - wait_for_completion() / complete ()
- Big kernel lock (BKL)
  - Locking the whole kernel; being discouraged.
    - lock_kernel() / unlock_kernel() / kernel_locked()

## Read-Copy Update

- Goal
  - A high performance and scaling algorithm for read-mostly situations
  - Reader must not required to acquire locks, execute atomic operations, or disable interrupts
- How
  - Writers create new versions atomically
    - Create new or delete old elements
  - Readers can access old versions independently of subsequent writers
  - Old versions are garbage-collected by poor man's GC, deferring destruction
  - Readers must signal "GC" when done

The materials are about RCU are from author's website (http://www.rdrop.com/users/paulmck/RCU/)

## Read-Copy Update

- Why
  - Readers are not permitted to block in read-side critical sections
  - Once an element is deleted, any CPU that subsequently performs a context switch cannot possibly gain a reference to this element
- Overhead might incurred
  - Readers incur little or no overhead (read_barrier_depends)
  - Writers incur substantial overhead
    - Writers must synchronize with each other
    - Writers must defer destructive actions until readers are done
    - The poor man's GC also incurs some overhead

The materials are about RCU are from author's website (http://www.rdrop.com/users/paulmck/RCU/)

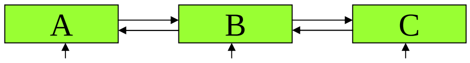
## Read-Copy Update terms

- Quiescent state
  - Context switch is defined as the quiescent state
  - Quiescent state cannot appear in a read-side critical section
  - CPU in quiescent state are guaranteed to have completed all preceding read-side critical section
- Grace period
  - Any time period during which all CPUs pass through a quiescent state
  - A CPU may free up an element (destructive action) after a grace period has elapsed from the time that it deletes the element

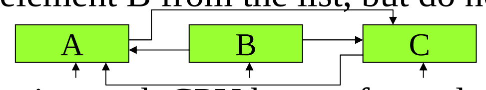
The materials are about RCU are from author's website (http://www.rdrop.com/users/paulmck/RCU/)

## Read-Copy Update example

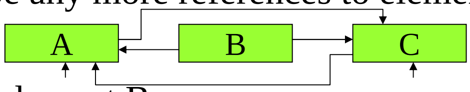
- Initial linked list



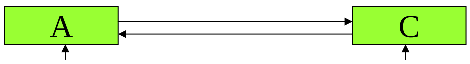

- Unlink element B from the list, but do not free it




- At this point, each CPU has performed one context switch after element B has been unlinked. Thus, there cannot be any more references to element B




- Free up element B




The materials are about RCU are from author’s website (http://www.rdrop.com/users/paulmck/RCU/)

## Read-Copy Update primitives

- rcu_read_lock() / rcu_read_unlock()
  - Mark the begin and end of a read-side critical section
  - NULL on non-preemptive kernel; disable/enable preemption on preemptive kernel

```
#define rcu_read_lock()          preempt_disable()
#define rcu_read_unlock()        preempt_enable()
```

- synchronize_rcu()
  - Mark the end of updater code and the beginning of reclaimer code
  - Wait until all pre-existing RCU read-side critical sections complete
  - Subsequently started RCU read-side critical sections not waited for
- call_rcu(struct rcu_head *, void (*func)(struct rcu_head *))
  - Asynchronous form of synchronize_rcu()
  - Instead of blocking, it registers a callback function which are invoked after all ongoing RCU read-side critical sections have completed

## Read-Copy Update primitives

- rcu_assign_pointer(p, v)
  - uses this function to assign a new value to an RCU-protected pointer
  - It returns the new value and executes any memory-barrier instructions required for a given CPU architecture

```
#define rcu_assign_pointer(p, v)    ({ smp_wmb(); (p) = (v); })
```

- rcu_dereference(p)
  - Protect an RCU-protected pointer for later safe-dereferencing; it executes any needed memory-barrier instructions for a given CPU architecture

```
#define rcu_dereference(p)    ({   typeof(p) _p1 = p; \
                                   smp_read_barrier_depends(); (_p1); })
```

## RCU for the Linux *list* API

```
static inline void __list_add_rcu(struct list_head * new,
                    struct list_head * prev, struct list_head * next)
{
        new->next = next;
        new->prev = prev;
        smp_wmb();
        next->prev = new;
        prev->next = new;
}

static inline void list_add_rcu(struct list_head *new, struct list_head *head)
{
        __list_add_rcu(new, head, head->next);
}

#define __list_for_each_rcu(pos, head) \
        for (pos = (head)->next; pos != (head); \
        pos = rcu_dereference(pos->next))
```

## rwlock, seqlock and RCU

- All of these locks has separate interfaces for readers and writers
- RCU can be used only for algorithms that can tolerate concurrent accesses and updates
- For read-mostly situation
  - Use RCU if applicable; it avoids atomic operations (cache bouncing) and memory barrier* (pipeline stall) overhead
  - Use seqlock if applicable; it has memory barrier overhead
  - Do not use rwlock if read-side critical section is short

* True for all architectures except Alpha

## Reader-writer lock versus RCU

- Mapping the primitives between rwlock and RCU

| Reader-writer lock | Read-Copy Update |
|---|---|
| rwlock_t | spinlock_t |
| read_lock() | rcu_read_lock() |
| read_unlock() | rcu_read_unlock() |
| write_lock() | spin_lock() |
| write_unlock() | spin_unlock() |

## Final Remarks

- Design protection when you start everything
- Identify the sources of concurrency
  - Callback functions
  - Event and interrupt handlers
  - Instances of kernel threads
  - When will be blocked and put to sleep
  - SMP and preemptive kernel
- Lock goes with data structures not code segments
- Keep things simple when starting
- Use the right synchronization tool for the job

## References


- Linux Kernel Development, 2nd edition, Robert Love, 2005
- Understanding the Linux Kernel, 3rd edition, Bovet & Cesati, O'REILLY, 2005
- Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, 2005
- Linux 2.6.10 kernel source
- RCU papers by Paul E. McKenney, http://www.rdrop.com/users/paulmck/RCU/

# Introduction to Linux Block Drivers

Hao-Ran Liu

## Sectors and blocks

- Sector
  - The basic unit of data transfer for the hardware device
  - Kernel expects a 512-byte sector. If you use a different hardware sector size, scale the kernel's sector numbers accordingly
- Block
  - A group of adjacent bytes involved in an I/O operation
  - Often 4096 bytes, can vary depending on the architecture and the exact filesystem being used

## Block driver registration

```
int register_blkdev(unsigned int major, const char *name);
```

- Allocating a dynamic major number if requested
- Creating an entry in `/proc/devices`

```
int unregister_blkdev(unsigned int major, const char *name);
```

- In the 2.6 kernel, the call to `register_blkdev` is entirely optional
- A separate registration interface to register disk drives and block device operations

## Block device operations

```
int (*open)(struct inode *inode, struct file *filp)
int (*release)(struct inode *inode, struct file *filp)
```

Called whenever the device is opened and closed. A block driver might spin up the device, lock the door (for removable media) in the open operation

```
int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,
             unsigned long arg)
int (*media_changed)(struct gendisk *gd)
```

Check if the user has changed the media in the drive, returning a nonzero value if so

```
int (*revalidate_disk)(struct gendisk *gd)
```

This function is called in response to a media change. It gives the driver a chance to perform whatever work is required to make the new media ready for use

* Request function, register elsewhere, handles the actual read or write of data.

## The `gendisk` structure

- the kernel's representation of an indivisual disk device
  - The kernel also uses `gendisk` structures to represent partitions

```
int major; int first_minor; int minors;
```

The first field is the major number of the device driver. A drive must use at least one minor number. A partitionable drive has one minor number for each possible partition. If minors = 16, it allows for the "full disk" device and 15 partitions

```
char disk_name[32];
```

The name of the disk device. It shows up in `/proc/partitions` and sysfs

```
struct block_device_operations *fops;
Struct request_queue *queue;
```

Structure used by the kernel to manage I/O requests for this device

## The `gendisk` structure (cont.)

```
sector_t capacity;
```

The capacity of this drive, in 512-byte sectors

```
void *private_data;
```

Block drivers may use this field for a pointer to their own internal data

# The `gendisk` API

```
struct gendisk *alloc_disk(int minors);
```

Allocation of the `struct gendisk` can only be done through this function. `Minors` is the number of minor numbers this disk uses

```
void del_gendisk(struct gendisk *gd);
```

Invalidates stuff in the gendisk and normally removes the final reference to the gendisk

```
void add_disk(struct gendisk *gd);
```

This function makes the disk available to the system. As soon as you call `add_disk`, the disk is "live" and its methods can be called at any time. So you should not call `add_disk` until your driver is completely initialized and ready to respond to requests on that disk.

# *Sbull* – A real example

- The *sbull* driver implements a set of in-memory virtual disk drives
- You can download the example from O'Reilly's website

Sbull allows a major number to be specified at compile or module load time. If no number is specified, one is allocated dynamically.

```
sbull_major = register_blkdev(sbull_major, "sbull");
if (sbull_major <= 0) {
        printk(KERN_WARNING "sbull: unable to get major number\n");
        return -EBUSY;
}
```

# Describing the sbull device

- The sbull device is described by an internal structure

```
struct sbull_dev {
        int size;                       /* Device size in bytes */
        u8 *data;                       /* The data array */
        short users;                    /* How many users */
        short media_change;             /* Flag a media change? */
        spinlock_t lock;                /* For mutual exclusion */
        struct request_queue *queue;    /* The device request queue */
        struct gendisk *gd;             /* The gendisk structure */
        struct timer_list timer;        /* For simulated media changes */
};
```

# Initialization of the `sbull_dev`

Basic initialization and allocation of the underlying memory

```
memset (dev, 0, sizeof (struct sbull_dev));
dev->size = nsectors*hardsect_size;
dev->data = vmalloc(dev->size);
if (dev->data == NULL) {
        printk (KERN_NOTICE "vmalloc failure.\n");
        return;
}
spin_lock_init(&dev->lock);

/* ... */

dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

Allocation of the request queue, `sbull_request` is our request function – the function that actually performs block read and write requests. The spinlock is provided by the driver because, often, the request queue and other driver data structures fall within the same critical section.

# Initialization of the `sbull_dev` (cont.)

Allocate, initialize and install the corresponding `gendisk` structure. `SBULL_MINORS` is the number of minor numbers each *sbull* device supports. The name of the disk is set such that the first one is *sbulla*, the second *sbullb*, and so on. Once everything is set up, we finish with a call to `add_disk`. Chances are that several of our methods will have been called for that disk by the time `add_disk` returns, so we take care to make that call the very last step in the initialization of our device.

```
dev->gd = alloc_disk(SBULL_MINORS);
if (! dev->gd) {
        printk (KERN_NOTICE "alloc_disk failure\n");
        goto out_vfree;
}
dev->gd->major = sbull_major;
dev->gd->first_minor = which*SBULL_MINORS;
dev->gd->fops = &sbull_ops;
dev->gd->queue = dev->queue;
dev->gd->private_data = dev;
snprintf (dev->gd->disk_name, 32, "sbull%c", which + 'a');
set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
add_disk(dev->gd);
```

# A note on sector sizes

- The kernel always expresses itself in 512-byte sectors, but not all hardware uses that sector size. Thus, it is necessary to translate all sector numbers accordingly.

```
blk_queue_hardsect_size(dev->queue, hardsect_size);
```

Use this function to inform the kernel of the sector size your device supports. The hardware sector size is a parameter in the request queue. The *sbull* device exports a `hardsect_size` parameter that can be used to change the "hardware" sector size of the device.

## A feature of the *sbull* device

- *Sbull* pretends to be a removable device
  - Whenever the last user closes the device, a 30-second timer is set; if the device is not opened during that time, the contents of the device are cleared, and the kernel will be told that the media has been changed

## *Sbull's* block device operations -- open()

The function maintains a count of users and calls **del_timer_sync** to remove the "media removal" timer. **check_disk_change** is a kernel function, which calls driver's **media_changed** function to check if a removable media has been changed. In that case, it invalidates all buffer cache entries and calls driver's **revalidate_disk** function.

```
static int sbull_open(struct inode *inode, struct file *filp)
{
        struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;

        del_timer_sync(&dev->timer);
        filp->private_data = dev;
        spin_lock(&dev->lock);
        if (! dev->users)
                check_disk_change(inode->i_bdev);
        dev->users++;
        spin_unlock(&dev->lock);
        return 0;
}
```

## *Sbull's* block device operations -- `release()`

The function, in contrast, decrement the user count and, if indicated, start the media removal timer. In a driver that handles a real hardware device, the `open` and `release` methods would set the state of the driver and hardware accordingly. A block device is opened when user space programs access the device directly (`mkfs`, `fdisk`, `fsck`) or when a partition on it is mounted.

```
static int sbull_release(struct inode *inode, struct file *filp)
{
        struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;

        spin_lock(&dev->lock);
        dev->users--;
        if (!dev->users) {
                dev->timer.expires = jiffies + INVALIDATE_DELAY;
                add_timer(&dev->timer);
        }
        spin_unlock(&dev->lock);
        return 0;
}
```

## *Sbull's* block device operations -- `media_changed() & revalidate_disk()`

If you are writing a driver for a nonremovable device, you can safely omit these methods. Both of these functions are called by `check_disk_change`. When a device is opened and the removable media has changed, the kernel will reread the partition table and start over with the device.

```
int sbull_media_changed(struct gendisk *gd)
{
        struct sbull_dev *dev = gd->private_data;
        return dev->media_change;
}

int sbull_revalidate(struct gendisk *gd)
{
        struct sbull_dev *dev = gd->private_data;

        if (dev->media_change) {
                dev->media_change = 0;
                memset (dev->data, 0, dev->size);
        }
        return 0;
}
```

## *Sbull's* block device operations -- `ioctl()`

- The higher-level block subsystem code intercepts a number of `ioctl` commands before your driver ever gets to see them
- *Sbull* `ioctl` method handles only one command – a request for the device's geometry
  - The kernel is not concerned with a block device's geometry; it sees it simply as a linear array of sectors
  - But certain user-space utilities still expect to be able to query a disk's geometry
    - Eg. the *fdisk* tool depends on cylinder information and does not function properly if that information is not available

## *Sbull's* block device operations -- `ioctl()` (cont.)

```
int sbull_ioctl (struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg)
{
   long size; struct hd_geometry geo;
   struct sbull_dev *dev = filp->private_data;

   switch(cmd) {
      case HDIO_GETGEO:
         /* Get geometry: since we are a virtual device, we have to make
          * up something plausible.  So we claim 16 sectors, four heads,
          * and calculate the corresponding number of cylinders.  We set the
          * start of data at sector four.
          */
         size = dev->size/KERNEL_SECTOR_SIZE;
         geo.cylinders = (size & ~0x3f) >> 6;
         geo.heads = 4; geo.sectors = 16; geo.start = 4;
         if (copy_to_user((void __user *) arg, &geo, sizeof(geo)))
            return -EFAULT;
         return 0;
   }
   return -ENOTTY; /* unknown command */
}
```

## Request processing
## -- request function

```
void request(request_queue_t *queue);
```

- The place where the real work gets done
- Does not need to complete all of the requests on the queue before it returns
  - But it must make a start on these requests and ensure that they are all, eventually, processed by the driver
- Invocation of the request function is (usually) entirely asynchronous with respect to the actions of any user-space process

## Request processing
## -- request queue

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

- Every device (usually) needs a request queue because:
  - Actual transfers to and from a disk can take place far away from the time the kernel requests them
  - Kernel needs the flexibility to schedule each transfer at the most propitious moment, grouping together requests that affect sectors close together on the disk (I/O scheduling)
- Whenever the request function is called, the queue lock is held by the kernel.
  - It prevents the kernel from queueing any other requests for your device
  - You may want to consider dropping the lock while the request function runs

# *Sbull's* request function
# -- a simple request method

This represents a block I/O request

It obtains the first incomplete request on the queue and returns NULL when there are no requests. It does not remove the request from the queue.

Exclude non-filesystem request because we don't know how to handle it

The index of the beginning sector on our device (in 512-byte sector)

The number of (512-byte) sectors to be transferred

```
static void sbull_request(request_queue_t *q)
{
        struct request *req;

        while ((req = elv_next_request(q)) != NULL) {
                struct sbull_dev *dev = req->rq_disk->private_data;
                if (! blk_fs_request(req)) {
                        printk (KERN_NOTICE "Skip non-fs request\n");
                        end_request(req, 0);
                        continue;
                }
                sbull_transfer(dev, req->sector, req->current_nr_sectors,
                               req->buffer, rq_data_dir(req));
                end_request(req, 1);
        }
}
```

The request has been processed sucessfully

A pointer to the buffer to or from which the data should be transferred

The direction of the transfer from the request (0 = read)

# *Sbull's* request function
# -- `sbull_transfer()`

```
static void sbull_transfer(struct sbull_dev *dev, unsigned long sector,
                unsigned long nsect, char *buffer, int write)
{
        unsigned long offset = sector*KERNEL_SECTOR_SIZE;
        unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;

        if ((offset + nbytes) > dev->size) {
                printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n",
offset, nbytes);
                return;
        }
        if (write)
                memcpy(dev->data + offset, buffer, nbytes);
        else
                memcpy(buffer, dev->data + offset, nbytes);
}
```

## Problems with the simple request function

- Executes requests synchronously, only 1 requests at a time
  - Some devices are capable of having numerous requests outstanding at the same time
- The largest single transfer never exceed the size of a single page

## Request queue

- A queue for keeping block I/O requests
- Stores parameters that describe what kinds of requests the device is able to service
  - Maximum size
  - Maximum number of segments per request
  - Hardware sector size, alignment requirements
- A plug-in interface allowing the use of multiple I/O schedulers
  - Improve I/O performance by accumulating and sorting requests
  - Merge of adjacent requests

# Queue creation and deletion functions

Accusys The RAID Architects

```
request_queue_t *blk_init_queue(request_fn_proc *request, spinlock_t *lock);
```

Create and initialize a request queue. The arguments are the request function for this queue and a spinlock that controls access to the queue. This function allocates memory and can fail because of this; you should always check the return value before attempting to use the queue

```
void blk_cleanup_queue(request_queue_t *);
```

Return a request queue to the system. After this call, your driver sees no more requests from the given queue and should not reference it again

Accusys The RAID Architects

# Queueing functions

The queue lock must be hold before calling these functions

```
struct request *elv_next_request(request_queue_t *queue);
```

This function returns the next request to process or NULL if no more requests remain to be processed. The request returned is left on the queue but marked as being active; this mark prevents the I/O scheduler from attempting to merge other requests with this one

```
void blk_dequeue_request(struct request *req);
```

Remove a request from a queue. If your driver operates on multiple requests from the same queue simultaneously, it must dequeue them in this manner

```
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

Put a dequeued request back on the queue

## Queue control functions

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

If your device has reached a state where it can handle no more outstanding commands, you can call `blk_stop_queue` to prevent the request function from being called until you call `blk_start_queue` to restart queue operations

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
```

This function tells the kernel the highest physical address to which your device can perform DMA. If a request comes in containing a reference to memory above the limit, a bounce buffer will be used for the operation. You can use these predefined symbols:

`BLK_BOUNCE_HIGH` : bounce all highmem pages.
`BLK_BOUNCE_ANY` : don't bounce anything
`BLK_BOUNCE_ISA` : bounce pages above ISA DMA boundary

## Queue control functions (cont.)

```
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
```

These functions set parameters describing the requests that can be satisfied by this device. `blk_queue_max_sectors` set the maximum size of any request in (512-byte) sectors; the default is 255. `blk_queue_max_phys_segments` and `blk_queue_max_hw_segments` both control how many physical segments (nonadjacent areas in system memory) may be contained within a single request. The first limit would be the largest sized scatter list the driver could handle, and the second limit would be the largest number of address/length pairs the host adapter can actually give as once to the device.

## Queue control functions (cont.)

```
void blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);
```

Some devices cannot handle requests that cross a particular size memory boundary. For example, if your device cannot handle requests that cross a 4-MB boundary, pass in a mask of `0x3fffff`. The default mask is `0xffffffff`

```
void blk_queue_dma_alignment(request_queue_t *queue, int mask);
```

Tells the kernel the memory alignment constraints your device imposes on DMA transfers. All requests are created with the given alignment, and the length of the request also matches the alignment. The default mask is `0x1ff`, which causes all requests to be aligned on 512-byte boundaries

```
void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);
```

Tells the kernel about your device's hardware sector size. All requests generated by the kernel are a multiple of this size and are properly aligned. All communications between the block layer and the driver continues to be expressed in 512-byte sectors, however.

## The anatomy of a request

- Each `request` structure represents one block I/O request; it is:
  - A set of segments, each of which corresponds to one in-memory buffer
  - A set of consecutive sectors on the block device
  - Implemented as a linked list of `bio` structures with some information for the driver to keep track of its position as it works through the request

# The bio -- uppermost interface used by filesystems

- `bio` is issued by filesystems, virtual memory, or a system call to read or write a block device
- It may be merged into an existing `request` structure or put into a newly created one

# The bio structure

`struct block_device *bi_bdev; sector_t bi_sector;`

The block device to be read/write; The first (512-byte) sector to be transferred for this `bio`

`unsigned int bi_size;`

The size of the data to be transferred, in bytes. This macro `bio_sectors(bio)` returns the size of a `bio` in sectors

`unsigned long bi_flags; unsigned long bi_rw;`

Sets of flags describing the `bio`. The least significant bit of `bi_rw` is set if this is a write request. Use `bio_data_dir(bio)` to query the read/write flag

`unsigned short bio_phys_segments; unsigned short bio_hw_segments;`

The number of physical segments contained within this BIO and the number of segments seen by the hardware after DMA mapping is done, respectively

# The `bio` structure (cont.)

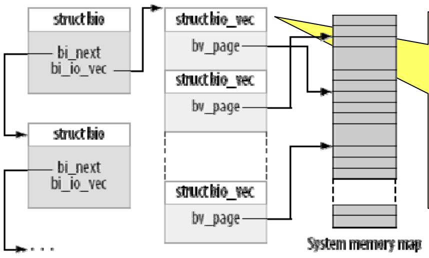
```
struct bio_vec {
  struct page *bv_page;
  unsigned int bv_len;    // bytes to be transferred
  unsigned int bv_offset; // starting at bv_offset
}
```

`struct bio_vec *bi_io_vec;`

An array of data structures indicating memory locations from which data is read or write

`unsigned short bi_vcnt; unsigned short bi_idx;`

The number of I/O vectors in `bi_io_vec`; Current I/O position in `bi_io_vec`




**To loops through every unprocessed entry in the `bi_io_vec` array, use the macro:**

```
int segno;
struct bio_vec *bvec;

bio_for_each_segment(bvec, bio, segno) {
   /* do something with this segment */
}
```

# Mappping the buffer of a `bio`

- To access the pages in a `bio` directly, make sure that they have a proper kernel virtual address
  - Pages in high memory are not addressable

map the i-th buffer in `bi_io_vec` array

atomic kmap slot

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
void __bio_kunmap_atomic(char *buffer, enum km_type type);
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
```

Use these functions to ensure that the buffer in a given `bio` is addressable. An atomic kmap is created and a kernel virtual address is returned. The caller cannot sleep while the mapping is in used.

map current buffer as indicated in `bio->bi_idx`

# Macros to read the current state of a `bio`

```
struct page *bio_page(struct bio *bio);
```

Returns a pointer to the `page` structure representing the page to be transferred next

```
int bio_offset(struct bio *bio);
```

Return the offset within the page for the data to be transferred

```
int bio_cur_sectors(struct bio *bio);
```

Returns the number of sectors to be transferred out of the current page

```
char *bio_data(struct bio *bio);
```

Returns a kernel logical address pointing to the data to be transferred. Note that if the page in question is in high memory, calling this function is a bug. By default, the block subsystem does not pass high-memory buffers to your driver, but if you have changed that setting with `blk_queue_bounce_limit`, you probably should not be using `bio_data`

# The `request` structure

```
sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;
```

These fields are for use only within the block subsystem; drivers should not make use of them

`hard_sector` is the first sector that has not been transferred. `hard_nr_sectors` is the total number of sectors yet to transfer. `hard_cur_sectors` is the number of sectors remaining in the current `bio`

```
struct bio *bio;
```

The linked list of `bio` structures for this request. Use `rq_for_each_bio` to traverse the list

```
char *buffer;
```

The simple driver example earlier use this field to find the buffer for the transfer. It equals to the result of calling `bio_data` on the current `bio`

# The `request` structure (cont.)

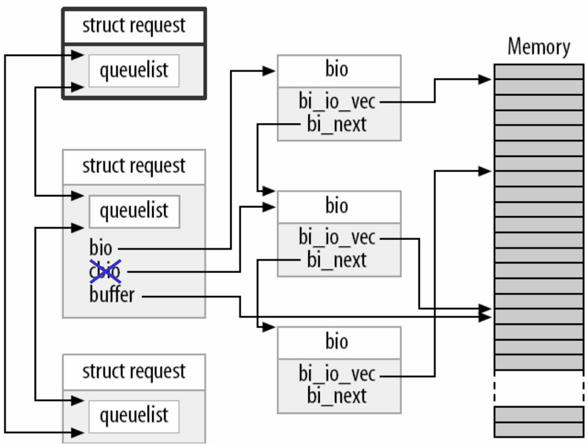
`unsigned short nr_phys_segments;`

Number of discinct segments after adjacent pages have been merged

`struct list_head queuelist;`

The linked list structure that links the request into the request queue. if the request is removed from the queue with `blkdev_dequeue_request`, you may use this list head for other purpose

# A request queue with a partially processed request



# Barrier requests

- Block layer reorders requests before submitting them to the device drivers to improve I/O performance
- But some applications require that certain I/O operations complete before the others
  - Journaling filesystems, relational databases
- The solution is barrier request. if a request is marked with `REQ_HARDBARRER` flag, it must be written to the drive before any following request is initiated

# Barrier request control functions

```
void blk_queue_ordered(request_queue_t *queue, int flag);
```

Inform the block layer that your driver implements barrier requests. In case a power failure occurs when the critical data is still sitting in the drive's cache, your driver must take steps to force the drive to actually write the data to the media

```
int blk_barrier_rq(struct request *req);
```

If this macro returns a nonzero value, the request is a barrier request

## Nonretryable requests

- If the macro returns a nonzero value on a failed request, your driver should simply abort the request instead of retrying it

```
int blk_noretry_request(struct request *req);
```

## Request completion functions

```
int end_that_request_first(struct request *req, int success, int count);
```

Tell the block code that your driver has completed transferring some or all of the sectors in an I/O request. `count` is the number of sectors transferred starting from where you last left off. If the I/O was successful, pass `success` as 1. The return value indicates if all sectors in this request have been transferred or not

```
void end_that_request_last(struct request *req);
```

wakeup whoever is waiting for the completion of the request and recycles the `request` structure

# `end_request` function

This function is called in ***sbull's*** request function

```
void end_request(struct request *req, int uptodate)
{
   if (!end_that_request_first(req, uptodate, req->hard_cur_sectors)) {
      add_disk_randomness(req->rq_disk);
      blkdev_dequeue_request(req);
      end_that_request_last(req);
   }
}
```

When all sectors in the request have been transferred, we dequeue the request from request queue and recycle it

contribute entropy to the system's random number pool. It should be called only if the disk's I/O completion time is truly random

# Work directly with the `bio` – Replace `sbull_request` function

```
static void sbull_full_request(request_queue_t *q)
{
        struct request *req;
        int sectors_xferred;
        struct sbull_dev *dev = q->queuedata;

        while ((req = elv_next_request(q)) != NULL) {
                if (! blk_fs_request(req)) {
                        printk (KERN_NOTICE "Skip non-fs request\n");
                        end_request(req, 0);
                        continue;
                }
                sectors_xferred = sbull_xfer_request(dev, req);
                if (! end_that_request_first(req, 1, sectors_xferred)) {
                        blkdev_dequeue_request(req);
                        end_that_request_last(req);
                }
        }
}
```

# Work directly with the bio (cont.)

```
static int sbull_xfer_request(struct sbull_dev *dev, struct request *req)
{
	struct bio *bio;
	int nsect = 0;

	rq_for_each_bio(bio, req) {
		sbull_xfer_bio(dev, bio);
		nsect += bio->bi_size/KERNEL_SECTOR_SIZE;
	}
	return nsect;
}
```

# Work directly with the bio (cont.)

```
static int sbull_xfer_bio(struct sbull_dev *dev, struct bio *bio)
{
	int i;
	struct bio_vec *bvec;
	sector_t sector = bio->bi_sector;

	/* Do each segment independently. */
	bio_for_each_segment(bvec, bio, i) {
		char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
		sbull_transfer(dev, sector, bio_cur_sectors(bio),
				buffer, bio_data_dir(bio) == WRITE);
		sector += bio_cur_sectors(bio);
		__bio_kunmap_atomic(bio, KM_USER0);
	}
	return 0; /* Always "succeed" */
}
```

# Prepare a scatterlist for DMA transfer

```
int blk_rq_map_sg(request_queue_t *q, struct request *rq,
                  struct scatterlist *sg);
```

Map a request to scatterlist, return number of `sg` entries setup. The returned scatterlist can then be passed to `dma_map_sg`. Caller must make sure `sg` can hold `rq->nr_phys_segments` entries. Segments that are adjacent in memory will be coalesced prior to insertion into the scatterlist. If you do not want to coalesce adjacent segments, clear the bit `QUEUE_FLAG_CLUSTER` in `q->queue_flags`

# The problem of queueing requests

- The purpose having request queue
  - Optimizing the order of requests
  - Stalling requests to allow an anticipated request to arrive
- Some devices does not benefit from these optimizations
  - Memory-based device like RAM disks, flash drives
  - Virtual disks created by RAID or LVM

# Overriding the default make request function

- Every request queue keeps a function pointer to its make request function, which is invoked when the kernel submit a `bio` to the request queue
- Override the default make request function `__make_request` to avoid reordering and stalling of requests

# Designing your make request function

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

The prototype of the make request function. In this function, we can put the `bio` into a request in the request queue, transfer the `bio` directly by walking through the `bio_vec`, or redirect it to another device. Returns a nonzero value when you want to redirect the `bio` to other device. It will cause the `bio` to be submitted again. So a "stacking" driver can modify the `bi_dev` to point to a difference device, change the starting sector value, and return.

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

Signal completion directly to the creator of the `bio`. `bytes` is the number of bytes you have transferred so far. It can be less than the number of bytes represented by the `bio` as a whole. If an error is encountered and the request cannot be completed, you can signal an error by providing a nonzero value like `-EIO` for `error` parameter

# *Sbull* without queuing requests

```
static int sbull_make_request(request_queue_t *q, struct bio *bio)
{
        struct sbull_dev *dev = q->queuedata;
        int status;

        status = sbull_xfer_bio(dev, bio);
        bio_endio(bio, bio->bi_size, status);
        return 0;
}
```

Never call `bio_endio` from a regular request function; that job is handled by `end_that_request_first` instead.

# *Sbull* without queuing requests (cont.)

This differs from `blk_init_queue` in that it does not actually set up the queue to hold requests

```
dev->queue = blk_alloc_queue(GFP_KERNEL);
if (dev->queue == NULL)
        goto out_vfree;
blk_queue_make_request(dev->queue, sbull_make_request);
```

Change the make request function of a request queue

# Reference

- For detailed information, refer to
  - Linux Device Drivers, 3rd edition, Chapter 16, Block Drivers

# Linux I/O Schedulers

Hao-Ran Liu

# Why I/O scheduler?

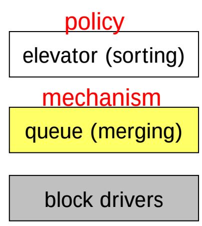
- Disk seek is the slowest operation in a computer
  - A system would perform horribly without a suitable I/O scheduler
- I/O scheduler arranges the disk head to move in a single direction to minimize seeks
  - Like the way elevators moves between floors
  - Achieve greater global throughput at the expense of fairness to some requests

## What do I/O schedulers do?

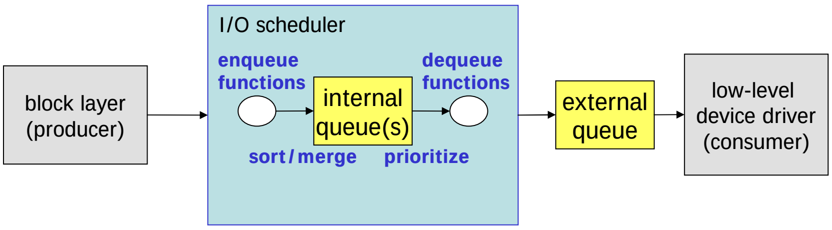
- Improve overall disk throughput by
  - Reorder requests to reduce the disk seek time
  - Merge requests to reduce the number of requests
- Prevent starvation
  - submit requests before deadline
  - Avoid read starvation by write
- Provide fairness among different processes

## Linux I/O scheduling framework

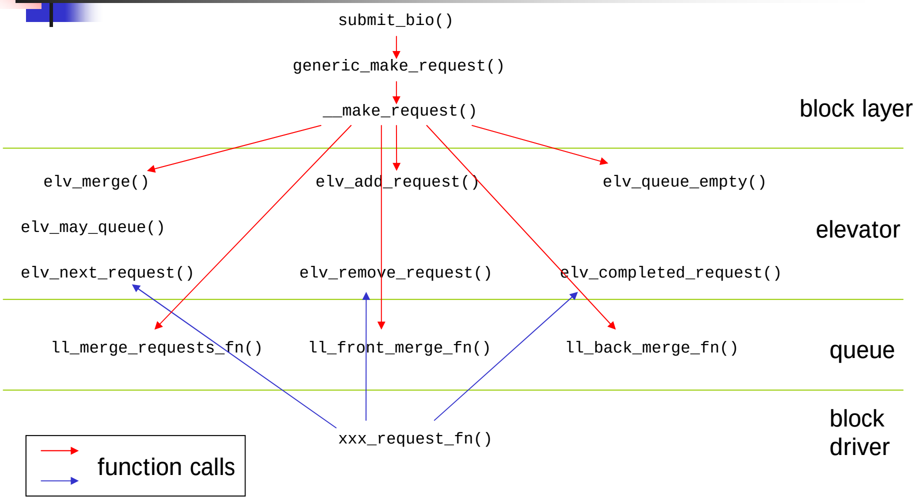
- Linux elevator is an abstract layer to which different I/O scheduler can attach
- Merging mechanisms are provided by request queues
  - Front or back merge of a request and a bio
  - Merge two requests
- Sorting policy and merge decision are done in elevators
  - Pick up a request to be merged with a bio
  - Add a new request to the request queue
  - Select next request to be processed by block drivers



## Abstraction of Linux I/O scheduler framework




## The relationship of I/O scheduler functions

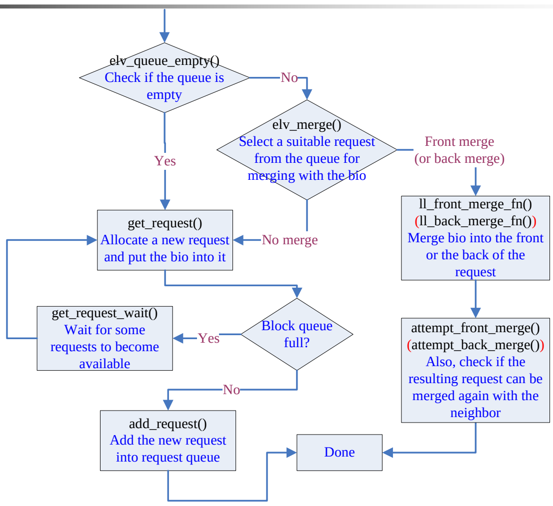


# Description of elevator functions

**Most functions are just wrappers. The actual implementation are elevator-specific**

| Type | Description |
|---|---|
| `elv_merge` | Find a request in the request queue to be merged with a bio. The function's return value indicate front merge, back merge or no merge. |
| `elv_add_request` | Add a new request to the request queue |
| `elv_may_queue` | Ask if the elevator allows enqueuing of a new request |
| `elv_remove_request` | Remove a request from the request queue |
| `elv_queue_empty` | Check if the request queue is empty |
| `elv_next_request` | Called by the device drivers to get next request from the request queue |
| `elv_completed_request` | Called when a request is completed |
| `elv_set_request` | When a new request is allocated, this function is called to initialize elevator-specific variables |
| `elv_put_request` | When a request is to be freed, this function is called to free memory allocated for some elevator. |

# Flowchart of `__make_request()`



Accusys

# Merge functions at request queue

```
struct request_queue
{
        struct list_head queue_head;
        struct elevator_queue *elevator;
        ...
        merge_request_fn *back_merge_fn;
        merge_request_fn *front_merge_fn;
        merge_requests_fn*merge_requests_fn;
        ...
}
```

A list of requests (external queue)

Elevator queue of this request queue

Pointers to merge functions:

```
ll_back_merge_fn()     : back merge a request and a bio
ll_front_merge_fn()    : front merge a request and a bio
ll_merge_requests_fn() : merge two requests
```

**`ll_xxx_fn()` is the default set of functions for merge**

Accusys

# The structure of elevator type

**Each request queue is associated with its own elevator queue of certain type**

```
struct elevator_queue
{
        struct elevator_ops *ops;
        void *elevator_data;
        struct kobject kobj;
        struct elevator_type *elevator_type;
};
```

the private data of the elevator

```
struct elevator_type
{
        struct list_head list;
        struct elevator_ops ops;
        struct kobj_type *elevator_ktype;
        char elevator_name[ELV_NAME_MAX];
        struct module *elevator_owner;
};
```

A list of all available elevator types

elevator functions

the name of the elevator

Accusys

# The structure of elevator operations

**These pointers point to the functions of a specific elevator**

```
struct elevator_ops {
        elevator_merge_fn *elevator_merge_fn;
        elevator_merged_fn *elevator_merged_fn;
        elevator_merge_req_fn *elevator_merge_req_fn;
        elevator_next_req_fn *elevator_next_req_fn;
        elevator_add_req_fn *elevator_add_req_fn;
        elevator_remove_req_fn *elevator_remove_req_fn;
        elevator_requeue_req_fn *elevator_requeue_req_fn;
        elevator_deactivate_req_fn *elevator_deactivate_req_fn;
        elevator_queue_empty_fn *elevator_queue_empty_fn;
        elevator_completed_req_fn *elevator_completed_req_fn;
        elevator_request_list_fn *elevator_former_req_fn;
        elevator_request_list_fn *elevator_latter_req_fn;
        elevator_set_req_fn *elevator_set_req_fn;
        elevator_put_req_fn *elevator_put_req_fn;
        elevator_may_queue_fn *elevator_may_queue_fn;
        elevator_init_fn *elevator_init_fn;
        elevator_exit_fn *elevator_exit_fn;
};
```

Accusys

# Elevators in Linux 2.6

- All elevator types are registered in a global linked list `elv_list`
- Request queues can change to a different type of elevator online
  - This allows for adaptive I/O scheduling based on current workloads
- I/O schedulers available
  - noop, deadline, CFQ, anticipatory

Accusys

# NOOP I/O scheduler

- Suitable for truly random-access device, like flash memory card
- Requests in the queue are kept in FIFO order
- Only the last request added to the request queue will be tested for the possibility of a merge

Accusys

# NOOP: Registration

```
static struct elevator_type elevator_noop = {
        .ops = {
                .elevator_merge_fn        = elevator_noop_merge,
                .elevator_merge_req_fn    = elevator_noop_merge_requests,
                .elevator_next_req_fn     = elevator_noop_next_request,
                .elevator_add_req_fn      = elevator_noop_add_request,
        },
        .elevator_name = "noop",
        .elevator_owner = THIS_MODULE,
};

static int __init noop_init(void) {
        return elv_register(&elevator_noop);
}

static void __exit noop_exit(void) {
        elv_unregister(&elevator_noop);
}

module_init(noop_init);
module_exit(noop_exit);
```

This structure stores the name of the noop elevator and pointers to `noop` functions. Use `elv_register()` function to register the structure with the plugin interfaces of the elevator

# NOOP: add request and get next request

```
static void elevator_noop_add_request(request_queue_t *q, struct request *rq,
                                      int where) {
        if (where == ELEVATOR_INSERT_FRONT)
                list_add(&rq->queuelist, &q->queue_head);
        else
                list_add_tail(&rq->queuelist, &q->queue_head);

        /*
         * new merges must not precede this barrier
         */
        if (rq->flags & REQ_HARDBARRIER)
                q->last_merge = NULL;
        else if (!q->last_merge)
                q->last_merge = rq;
}

static struct request *elevator_noop_next_request(request_queue_t *q) {
        if (!list_empty(&q->queue_head))
                return list_entry_rq(q->queue_head.next);

        return NULL;
}
```

Add a new request to the request queue

Called by the device driver to get the next request to be submitted. If the request queue is not empty, return the request at the head of the queue

# NOOP: request merge

```
/*
 * See if we can find a request that this buffer can be coalesced with.
 */
static int elevator_noop_merge(request_queue_t *q, struct request **req,
                               struct bio *bio) {
        int ret;

        ret = elv_try_last_merge(q, bio);
        if (ret != ELEVATOR_NO_MERGE)
                *req = q->last_merge;

        return ret;
}

static void elevator_noop_merge_requests(request_queue_t *q,
                          struct request *req, struct request *next) {
        list_del_init(&next->queuelist);
}
```

Given a bio, find a adjacent request in the request queue to be merged with.

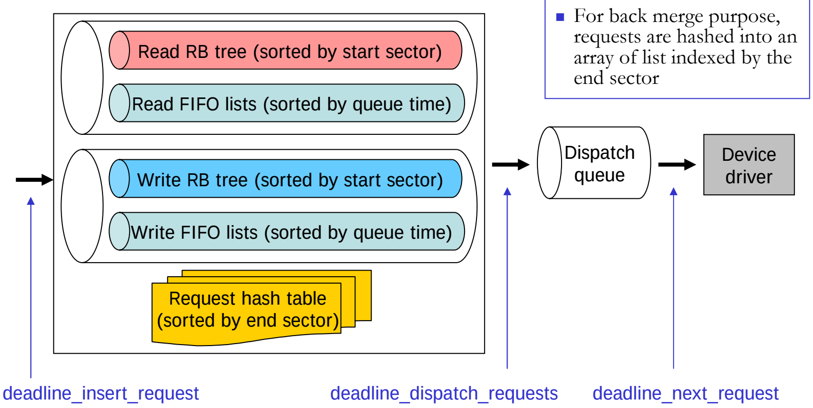
This function simply remove `next` request from the request queue. It is called after `next` are merged into `req`.

# Deadline I/O scheduler

- Goal
  - Reorder requests to improve I/O performance while simultaneously ensuring that no I/O request is being starved
  - Favor reads over writes
- Each requests is associated with a expire time
  - Read: 500ms, write 5sec
- Requests are inserted into
  - A sorted-by-start-sector queue (two queues! for read and write)
  - A FIFO list (two lists too!) sorted by expire time
- Normally, requests are pulled from sorted queues. However, if the request at the head of either FIFO queue expires, requests are still processed in sorted order but started from the first request in the FIFO queue

# Architecture view of Deadline I/O scheduler

- The sorted queues are built on red-black trees
- For back merge purpose, requests are hashed into an array of list indexed by the end sector



## Deadline: dispatching requests

1. If [next_req] is in the batch (adjacent to previous request and batch count < 16), set it as [dispatch_req] and jump to step 5
2. Here, we are not in a batch. If there are read reqs and write is not starved, select read dir and jump to step 4
3. If there are write reqs, select write dir. Otherwise, return 0
4. If the first req in the fifo of the selected data direction expired, set it as [dispatch_req] and set batch count = 0. Otherwise, set [next_req] as [dispatch_req]
5. Increase batch count and dispatch the [dispatch_req].
6. Search forward from the end sector of [dispatch_req] in the RB tree of selected dir. Set the next request as [next_req]

## Anticipatory scheduling Background

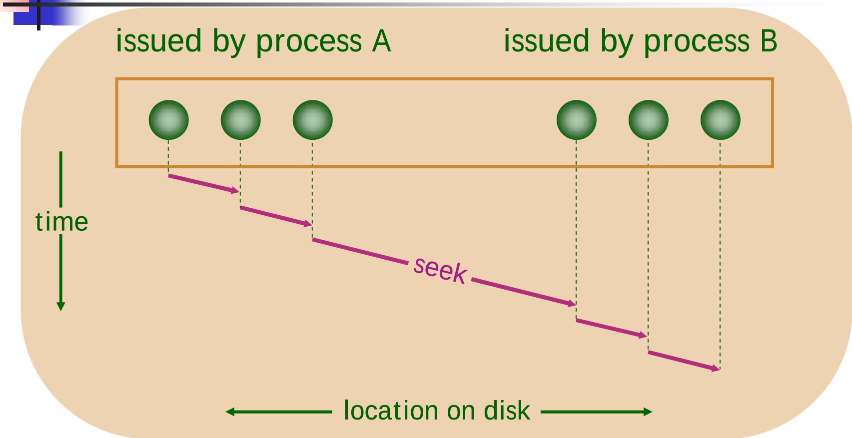
Disk schedulers reorder available disk requests for

- performance by seek optimization,
- proportional resource allocation, etc.

Any policy needs multiple outstanding requests to make good decisions!

from http://www.cs.rice.edu/~ssiyer/r/antsched/

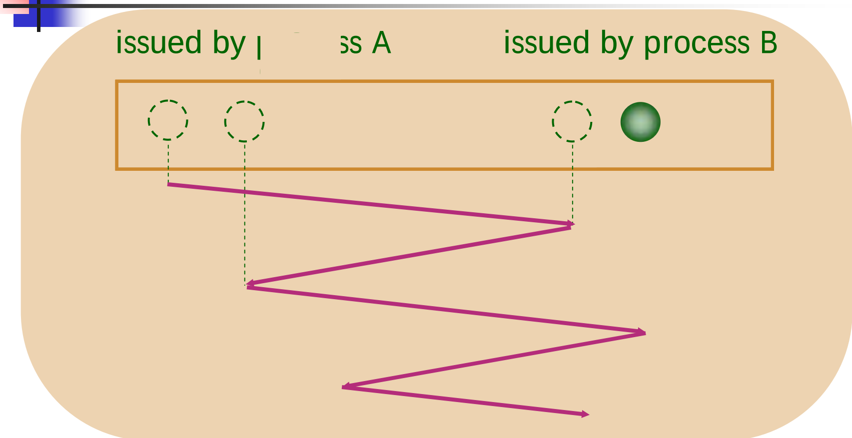
# With enough requests...




E.g., Throughput = 21 MB/s (IBM Deskstar disk)

from http://www.cs.rice.edu/~ssiyer/r/antsched/

# With synchronous I/O...




E.g., Throughput = 5 MB/s

from http://www.cs.rice.edu/~ssiyer/r/antsched/

# Deceptive idleness

Process A is about to issue next request.

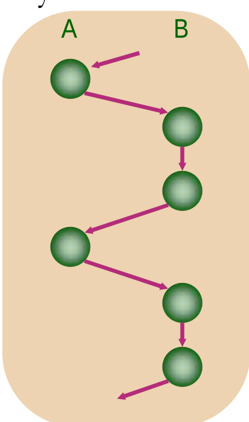
but

Scheduler hastily assumes that process A has no further requests!

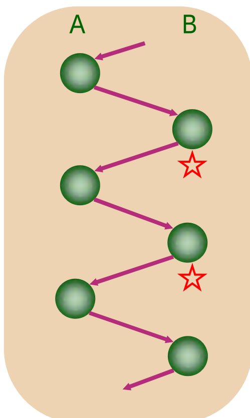
from http://www.cs.rice.edu/~ssiyer/r/antsched/

# Proportional scheduler

Allocate disk service in say 1:2 ratio:




**Deceptive idleness causes 1:1 allocation:**




from http://www.cs.rice.edu/~ssiyer/r/antsched/

# Anticipatory scheduling

Key idea: Sometimes wait for process whose request was last serviced.

Keeps disk idle for short intervals.

But with informed decisions, this:

- Improves throughput
- Achieves desired proportions

from http://www.cs.rice.edu/~ssiyer/r/antsched/

# Cost-benefit analysis

Balance expected benefits of waiting against cost of keeping disk idle.

Tradeoffs sensitive to scheduling policy

e.g., 1. seek optimizing scheduler

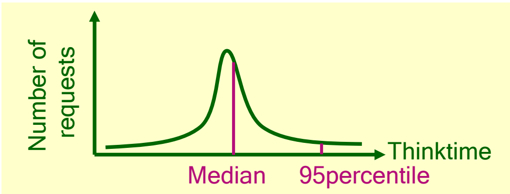
2. proportional scheduler

from http://www.cs.rice.edu/~ssiyer/r/antsched/

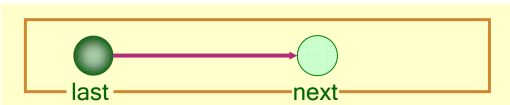
# Statistics

For each process, measure:

1. Expected median and 95percentile thinktime




2. Expected positioning time




from http://www.cs.rice.edu/~ssiyer/r/antsched/

# Cost-benefit analysis for seek optimizing scheduler

best := best available request chosen by scheduler

next := expected forthcoming request from process whose request was last serviced

Benefit =
best.positioning_time — next.positioning_time

Cost = next.median_thinktime

Waiting_duration =
(Benefit > Cost) ? next.95percentile_thinktime : 0

from http://www.cs.rice.edu/~ssiyer/r/antsched/

## Proportional scheduler

Costs and benefits are different.

e.g., proportional scheduler:

Wait for process whose request was last serviced,
1. if it has received less than its allocation, and
2. if it has thinktime below a threshold (e.g., 3ms)

Waiting_duration = next.95percentile_thinktime

from http://www.cs.rice.edu/~ssiyer/r/antsched/

## Prefetch

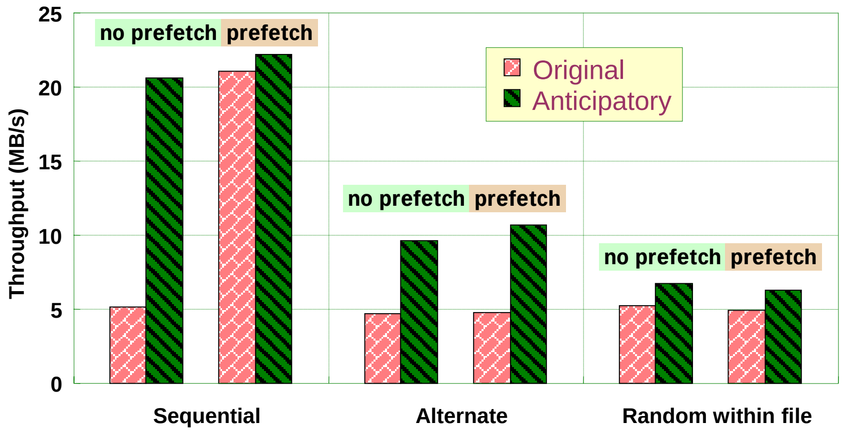
Overlaps computation with I/O.

Side-effect:
avoids deceptive idleness!

- Application-driven
- Kernel-driven

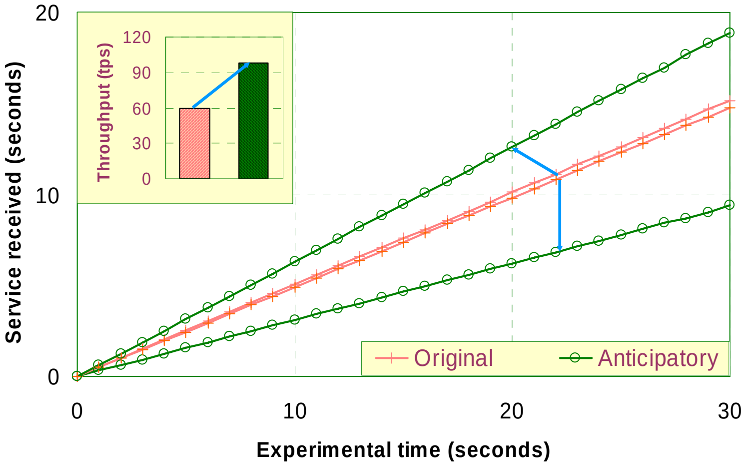
from http://www.cs.rice.edu/~ssiyer/r/antsched/

# Microbenchmark




from http://www.cs.rice.edu/~ssiyer/r/antsched/

# Proportional scheduler




**Database benchmark: two databases, select queries**

from http://www.cs.rice.edu/~ssiyer/r/antsched/

## Work-conserving vs. non-work-conserving

- Work-conserving scheduler
  - If the disk is idle or a request is completed, next request in the queue is scheduled immediately
- Non-work-conserving scheduler
  - the disk stands idle in the face of nonempty queue
- Anticipatory scheduler are non-work-conserving

## Anticipatory I/O scheduler in Linux

- Based on deadline I/O scheduler
- Suitable for desktop, good interactive performance
- Design shortcomings
  - Assume only 1 physical seeking head
    - Bad for RAID devices
  - Only 1 read request are dispatched to the disk controller at a time
    - Bad for controller that supports TCQ
  - Read anticipation assumes synchronous requests are issued by individual processes
    - Bad for requests issued cooperatively by multiple processes
- Rough benefit-cost analysis
  - Anticipate a better request if mean thinktime of the process < 6ms and mean seek distance of the process < seek distance of next request

# Anticipatory IO scheduler policy

- One-way elevator algorithm
  - Limited backward seeks
- FIFO expiration times for reads and for writes
  - When a requests expire, interrupt the current elevator sweep
- Read and write request batching
  - Scheduler alternates dispatching read and write batches to the driver. The read (write) FIFO timeout values are tested only during read (write) batches.
- Read Anticipation
  - At the end of each read request, the I/O scheduler examines its next candidate read request from its sorted read list and decide whether to wait for a "better request"

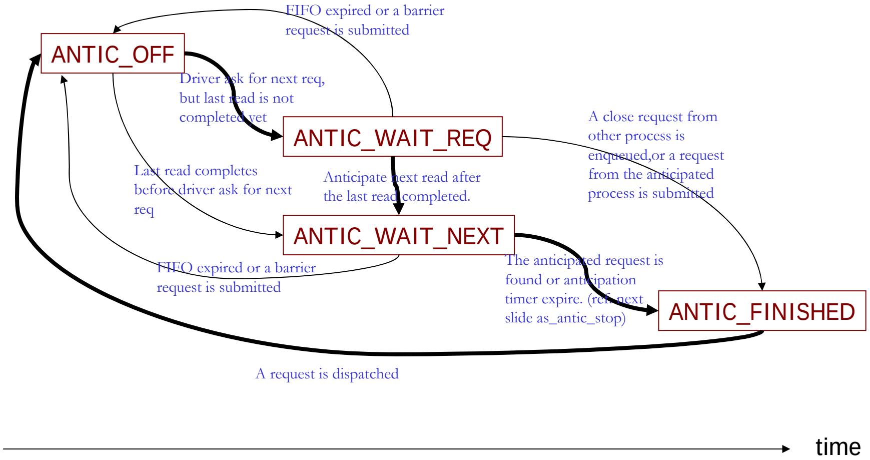
# I/O statistics for anticipatory scheduler

- Per request queue (as_data)
  - The last sector of the last request
  - Exit probability
    - Probability a task will exit while being waited on
- Per process (as_io_context)
  - Last request completion time
  - Last request position
  - Mean think time
  - Mean seek distance

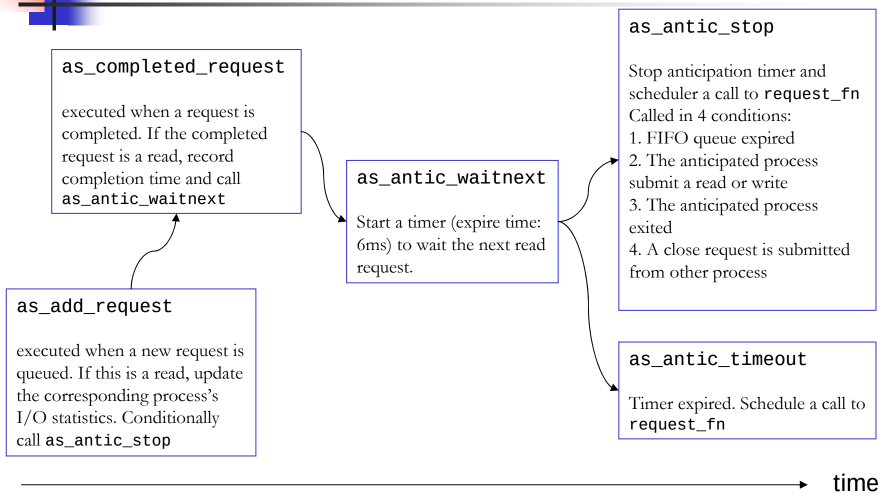
# Anticipation States

- ANTIC_OFF
  - Not anticipating (normal operation)
- ANTIC_WAIT_REQ
  - The last read has not yet completed
- ANTIC_WAIT_NEXT
  - Currently anticipating a request vs last read (which has completed)
- ANTIC_FINISHED
  - Anticipating but have found a candidate or timed out

# State transitions of request anticipation



# Functions executed during the anticipation of requests




# I/O statistics – thinktime & seek distance

- These statistics are associated with each process, but not with a specific I/O device
  - The statistics will be a combination of I/O behavior from all actively-use devices (It seems bad!)
- Thinktime
  - At enqueuing of a new read request, thinktime = current jiffies – completion time of last read request
- seek distance
  - At enqueuing of a new read request, seek distance = |start sector of the new request – last request end sector|

## I/O statistics – average thinktime and seek distance

Accusys The RAID Architects

- Previous I/O history decays as new request are enqueued
- Fixed point arithmetic (1.0 == 1 << 8)

**Mean thinktime of a process**

$$tsamples = \frac{7 \times tsamples + 256}{8}$$

$$ttotal = \frac{7 \times ttotal + 256 \times thinktime}{8}$$

$$tmean = \frac{ttotal + 128}{tsamples}$$

**Mean seek distance of a process**

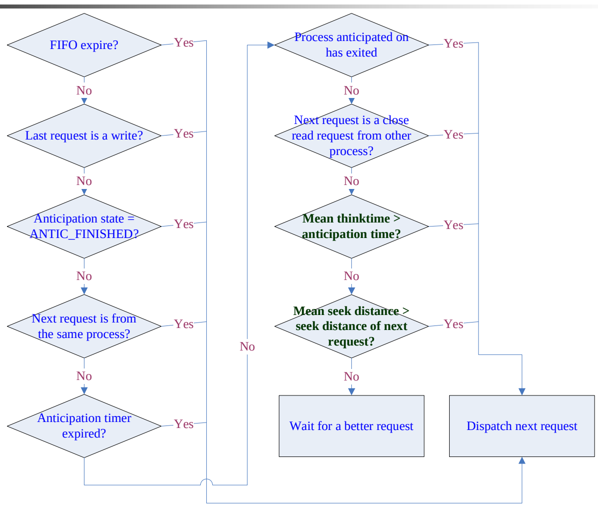
$$ssamples = \frac{7 \times ssamples + 256}{8}$$

$$stotal = \frac{7 \times stotal + 256 \times seekdist}{8}$$

$$smean = \frac{stotal + {}^{ssamples}\!/_{2}}{ssamples}$$

## Make a decision – Shall we anticipate a “better request”?

Accusys The RAID Architects



# Cooperative Anticipatory Scheduler

- Proposed in this paper: Enhancements to Linux I/O scheduler, OLS2005
- The problems of anticipatory scheduler
  - Anticipation works only when requests are issued by the same process
- Solution
  - Keep anticipating even when the anticipated process has exited
  - Cooperative exit probability: existence of cooperative processes related to dead processes

AS failed to anticipate chunk reads

AS works too well for Program 1. Program 2 starved.

# CAS: Performance Evaluation

### Streaming writes and reads

**Program 1:**
```
while true
do
    dd if=/dev/zero of=file \
       count=2048 bs=1M
done
```
**Program 2:**
```
time cat 200mb-file > /dev/null
```

| Scheduler | Execution time (sec) | Throughput (MB/s) |
|---|---|---|
| Deadline | 129 | 25 |
| AS | 10 | 33 |
| CAS | 9 | 33 |

### Streaming and chunk reads

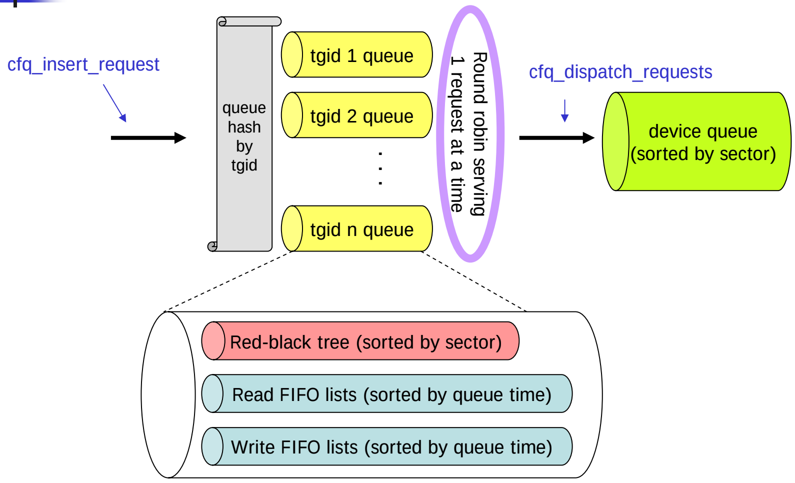
**Program 1:**
```
while true
do
    cat big-file > /dev/null
done
```
**Program 2:**
```
time find . -type f -exec \
    cat '{}' ';' > /dev/null
```

| Scheduler | Execution time (sec) | Throughput (MB/s) |
|---|---|---|
| Deadline | 297 | 9 |
| AS | 4767 | 35 |
| CAS | 255 | 34 |

# CFQv2 (Complete Fair Queuing) I/O scheduler

- Goal
  - Provide fair allocation of I/O bandwidth among all the initiators of I/O requests
- CFQ can be configured to provide fairness at per-process, per-process-group, per-user and per-user-group levels.
- Each initiator has its own request queue and CFQ services these queues round-robin
  - Data writeback is usually performed by the ***pdflush*** kernel threads. That means, all data writes share the alloted I/O bandwidth of the ***pdflush*** threads

# Architecture view of CFQv2



# References


- Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O, Sitaram Iyer, ACM SOSP'01
- Enhancements to Linux I/O scheduling, Seetharami Seelam, OLS'05
- Linux 2.6.12 kernel source
- Linux Kernel Development, 2nd edition, Robert Love, 2005