Shell 命令

1. 目录和文件操作：
   - pwd/cd/cat
   - rmdir/mkdir[-p, --parents]
   - rm/cp[-f, --force][-i, --interactive][-R,-r, --recursive]
   - mv[-f, --force][-i, --interactive]
   - ln[-f, --force][-i, --interactive][-s, --symbolic]
   - chmod[-R, --recursive]

The format of a symbolic mode is `[ugoa...][[+-=][rwxXstugo...]...][,...] `read (r), write (w), execute (or access for directories) (x),execute only if the file is a directory or already has execute permission for some user (X), set user or group ID on execution (s), sticky (t), the permissions granted to the user who owns the file (u), the permissions granted to other users who are members of the file's group (g), and the permissions granted to users that are in neither of the two preceding categories (o)

2. 目录下操作命令：
   - ls [–l][-a, --all][-F, --classify]
   - more
   - find [path...] [expression]

default path is the current directory; default expression is –print expression may consist of: operators, options, tests, and actions:

normal options:
-depth/-maxdepth LEVELS/-mindepth LEVELS

tests (N can be +N or -N or N): -empty/-gid N/-group NAME/-links N/-name PATTERN/ -perm [+-]MODE/-type [bcdpflsD]/-uid N/-user NAME

actions: -delete/-printf FORMAT/-print/-fprint0 FILE/-fprint FILE/-ls/-quit/-exec COMMAND ; -exec COMMAND {} + -ok COMMAND ;

3. 文本文件操作命令：
   - grep/sort/diff
   - wc[–l][-w][-c]

系统调用

- **pid_t fork(void);**

fork() creates a child process that differs from the parent process only in its PID and PPID On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and errno will be set appropriately.

int execl(const char *path, const char *arg, ...);

- **exec**

int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);

- **wait, waitpid**

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

The call wait(&status) is equivalent to: waitpid(-1, &status, 0);

The value of pid can be:

< -1 meaning wait for any child process whose process group ID is equal to the absolute value of pid.

-1 meaning wait for any child process.

0 meaning wait for any child process whose process group ID is equal to that of the calling process.

>0 meaning wait for the child whose process ID is equal to the value of pid.

wait(): on success, returns the process ID of the terminated child; on error, -1 is returned.

- **int kill(pid_t pid, int sig);**
- **signal**

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);

System Calls

Invoking System Calls

- System call invocation in an application program such as fork();
- Wrapper routine in libc standard library fork() { ... " int ox80" };
- System call handler

```
system_call
    sys_fork()
        ret_from_sys_call()
        iret;
```

- System call service routine: sys_fork() { }

Initializing System calls

```
start_kernel
//init/main.c
  trap_init
  //arch/i386/kernel/traps.c
    // sets up the IDT entry corresponding to vector 128
    set_system_gate(SYSCALL_VECTOR = 0x80,
      &system_call);
        _set_gate( gate_addr = idt_table+n,
                   type =15,   dpl = 3,   addr);
```

Initializing System calls

```
#define _set_gate(gate_addr,type,dpl,addr) \
do { \
  int __d0, __d1; \
  __asm__ __volatile__ ("movw %%dx,%%ax\n\t" \
        "movw %4,%%dx\n\t" \
        "movl %%eax,%0\n\t" \
        "movl %%edx,%1" \
        :"=m" (*((long *) (gate_addr))), \
         "=m" (*(1+(long *) (gate_addr))), "=&a" (__d0), "=&d" (__d1) \
        :"i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
         "3" ((char *) (addr)),"2" (__KERNEL_CS << 16)); \
} while (0)
```

System Call Handler

```
pushl %eax                          # save orig_eax
SAVE_ALL                    # save the registers
GET_CURRENT(%ebx)           # get the current ID
cmpl $(NR_syscalls),%eax # check system call
jae badsys
testb $0x02,ptrace(%ebx)          # PT_TRACESYS
jne tracesys
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
movl %eax,EAX(%esp)               # save the return value
ALIGN
.globl ret_from_sys_call          # return
.globl ret_from_intr
```

Parameter Passing

- System call parameters are usually passed to the system call handler in the CPU registers, then copied onto the Kernel Mode stack.
- The length of each parameter cannot exceed the length of a register, that is 32 bits.
- The number of parameters must not exceed size since the Intel Pentium has a very limited number of registers. (eax, ebx, ecx, edx, esi, edi).

Verifying Parameters

- All system call parameters must be carefully checked before the kernel attempts to satisfy a user request.
- Whenever a parameter specifies an address, the kernel must check whether it is inside the process address space. (verify_area, access_ok)
- Accessing the process address space get_user(x,ptr) // include/asm-i386/uaccess.h __get_user_x

Memory Addressing

Memory Addressing: 3 addresses

- Logical address
  - Consists of a segment and an offset.
  - Included in the machine language instruction to specify the address of operand or of an instruction.
- Linear address
  - A single 32-bit unsigned integer
  - Can be used to address up to 4GB
- Physical address
  - Used to address memory cells included in memory chips.
  - Corresponding to the electrical signals sent along the address pins of the microprocessor to the memory bus.
- Address translation
  - Logical address → SEGMENTATION → Linear address
  - Linear address → PAGING → Physical address

Memory Addressing: I386 segmentation

- Real-mode
  - Segment selectors
- Protected-mode
  - Segment selectors
  - Segment descriptor table registers
  - Segment descriptors

Memory Addressing: I386 segmentation

- Segment registers
  - cs: code segment register
  - ds: data segment register
  - ss: stack segment register
  - es, fs, gs: additional data segment registers.

Memory Addressing: I386 segmentation

- IDT, GDT, LDT registers
  - IDTR: Interrupt descriptor table register
    - GDT maintains a list of most segments and may contain special “system” descriptors.
  - GDTR: Global descriptor table register
    - IDT maintains a list of interrupt service routines.
  - LDTR: Local descriptor table register
    - LDT is optional, can extends range of GDT, is allocated to each task when multitasking is enabled.

Memory Addressing: I386 segmentation

Segment descriptors define

- Base address (32 bits)
- segment limit(20 bits)
- Type of segment (4 bits)
- Privilege level of segment (2 bits)
- Whether segment is physically present (1 bit)
- Whether segment has been accessed before (1 bit)
- Granularity of limit field (1 bit)
- Size of operands within segment (1 bit)
- Intel reserved flag (1 bit)
- User-defined flag (1 bit)

Memory Addressing: Linux segmentation

```
lgdt gdt_descr
 // arch/i386/kernel/head.S
  gdt_descr:
    // 16 bit for limit
    // 32 bit for base
    .word GDT_ENTRIES*8-1
    SYMBOL_NAME(gdt):
    .long SYMBOL_NAME(gdt_table)
```

Memory Addressing: Linux segmentation

```
ENTRY(gdt_table)  [arch/i386/kernel/head.S]
  // 8 bits for BASE 31:24
  // 8 for G--D/B--0¡ªAVL   LIMIT
  // 8 for P--DPL--S        TYPE--A
  // 24 for BASE 23:0  and 16 for LIMIT 15:0
  .quad 0x0000000000000000 /* NULL descriptor */
  .quad 0x0000000000000000 /* not used */
  .quad 0x00cf9a000000ffff /* 0x10 k                */
                           /* cf9a = 1100-1111-1001-1010 */
  .quad 0x00cf92000000ffff /* 0x18 k                */
                           /* cf92 = 1100-1111-1001-0010 */
  .quad 0x00cffa000000ffff /* 0x23 u                */
                           /* cffa = 1100-1111-1111-1010 */
  .quad 0x00cff2000000ffff /* 0x2b u                */
                           /* cff2 = 1100-1111-1111-1010 */
```

Memory Addressing: Linux segmentation

```
// include/asm-i386/segment.h
// 13 bits for index
```

```
// 1 bit for GDT or LDT
// 2 bits for RPL (Requestor Privilege Level)
#define __KERNEL_CS     0x10  // 2-0-0
#define __KERNEL_DS     0x18  // 3-0-0
#define __USER_CS       0x23  // 4-0-3
#define __USER_DS       0x2B  // 5-0-3
```

Memory Addressing: I386 Paging

- Address bits 31 to 22 select one of 1024 PDEs in the page directory
- Address bits 21 to 12 select one of 1024 PTS in the page table
- Address bits 11 to 0 select one of 4096 bytes in the page
- Registers CR3 locates the base address of the page directory
- PDE locates the base address of the page table
- PTE locates the base address of the page

Memory Addressing: I386 Paging

Memory Addressing: I386 Paging

Memory Addressing: I386 Paging

Memory Addressing: Linux Paging

- The Linux paging model:

  Linear addresses
  → Page global directory (PGD)
  → Page middle directory (PMD)
  → Page table
  → Physical addresses

Memory Addressing: Linux Paging

```
// include/asm-i386/pgtable-2level.h
#define PGDIR_SHIFT     22
#define PTRS_PER_PGD    1024
/*
 * the i386 is two-level, so we don't really have any
 * PMD directory physically.
 */
#define PMD_SHIFT       22
#define PTRS_PER_PMD    1
#define PTRS_PER_PTE    1024
```

Memory Addressing: Linux Paging

// include/asm-i386/page.h

pgt_t

pmd_t

pte_t

// include/asm-i386/pgalloc.h

pgd_alloc

  get_pgd_fast

    get_pgd_slow

MEMORY MANAGEMENT

Page Frame Management

```
// node NUMA
typedef struct pglist_data {
  //include/linux/mmzone.h
} pg_data_t;
// zone descriptor
typedef struct zone_struct{
  //include/linux/mmzone.h
} zone_t;
// page descriptor
typedef struct page {
  //include/linux/mm.h
} mem_map_t;
```

Page Frame Management

```
// numa.c
static bootmem_data_t contig_bootmem_data;
pg_data_t contig_page_data = { bdata: &contig_bootmem_data };
// mm/memory.c
mem_map_t* mem_map;
```

Page Frame Management

```
start_kernel //init/main.c
 setup_arch //arch/i386/kernel/setup.c
  paging_init //arch/i386/kernel/init.c
   zone_sizes_init //arch/i386/kernel/init.c
    free_area_init       //mm/page_alloc.c
     free_area_init_core //mm/page_alloc.c
       build_zonelists    //mm/page_alloc.c
start_kernel //init/main.c
 mem_init //arch/i386/kernel/setup.c
```

Page Frame Management

```
__get_free_pages
// mm/page_alloc.c
  alloc_pages
  // include/linux/mm.h
    _alloc_pages
    //mm/page_alloc.c
      alloc_pages_pgdat
      // NUMA
         __alloc_pages // buddy
           //mm/page_alloc.c
             rmqueue
```

Page Frame Management

```
free_pages
// mm/page_alloc.c
  __free_pages
    __free_pages_ok
```

Memory Area Management

- The memory is organized in caches, one cache for each object type. (e.g. inode_cache, dentry_cache, buffer_head, vm_area_struct) Each cache consists of many slabs (they are small (usually one page long) and always contiguous), and each slab contains multiple initialized objects.
- Each cache can only support one memory type (GFP_DMA, GFP_HIGHMEM, normal). If you need a special memory type, then must create a new cache for that memory type.
- In order to reduce fragmentation, the slabs are sorted in 3 groups: full slabs with 0 free objects, partial slabs, empty slabs with no allocated objects
- If partial slabs exist, then new allocations come from these slabs, otherwise from empty slabs or new slabs are allocated.

Memory Area Management

```
typedef struct kmem_cache_s kmem_cache_t;
// include/linux/slab.h
typedef struct slab_s slab_t
// mm/slab.c
```

Memory Area Management

- cache_cache: The first cache contains the cache descriptors of the remaining caches used by the kernel.
- Twenty-six additional caches contain geometrically distributed memory areas. The table, called cache_sizes (whose elements are of type cache_sizes_t), points to the 26 cache descriptors associated with memory areas of size 32, 64, 128, 256, 512, 1,024, 2,048, 4,096, 8,192, 16,384, 32,768, 65,536, and 131,072 bytes, respectively.

Memory Area Management

```
// Interfacing the Slab Allocator
// with the Buddy System
kmem_getpages
 __get_free_pages
kmem_freepages
  free_pages
kmem_cache_grow
kmem_slab_destroy
```

Memory Area Management

```
kmalloc
//mm/slab.c
  __kmem_cache_alloc
  //mm/slab.c
    __kmem_cache_alloc
    //mm/slab.c
```

Noncontiguous Memory Management

```
struct vm_struct{ //include/linux/vmalloc.h
  unsigned long     flags;
  void*             addr; //the linear address
  unsigned long     size;
  struct vm_struct* next;
};
struct vm_struct* vmlist; // mm/vmalloc.c
```

Noncontiguous Memory Management

```
get_vm_area // mm/vmalloc.c
vmalloc //include/linux/vmalloc.h
```

Noncontiguous Memory Management

```
__vmalloc
  get_vm_area
  vmalloc_area_pages
    pgd_offset_k
      pgd_index
     pmd_alloc
      pte_alloc
      alloc_area_pte
    alloc_area_pmd
```

Noncontiguous Memory Management

```
vfree
  vmfree_area_pages
    pgd_offset_k
    free_area_pmd
      free_area_pte
```

Virtual File System

VFS: Introduction

- The *Virtual File System* (also known as Virtual Filesystem Switch or VFS) is a kernel software layer that handles all system calls related to a standard Unix file system.
- Its main strength is providing a common interface to several kinds of filesystems.
- An example

  ```
  $ cp  /floppy/TEST  /tmp/test
  ```

  # where /floppy is the mount point of an MS-DOS diskette # /tmp is a normal Second Extended Filesystem (Ext2) # directory.

VFS: File model

VFS supports three main classes of filesystems:

- Disk-based filesystems (ext2, HPFS, NTFS, VFAT, iso9660)
- Network filesystems (NFS, SMB)
- Special filesystems (/proc, /dev/pts)

VFS File Model

- superblock: to store information concerning a mounted filesystem. (Filesystem control block)
- inode: to store general information about a specific file. (File control block)
- dentry: to store information about the linking of a directory entry with the corresponding file.
- file: to store information about the interaction between an open file and a process.

VFS: VFS system calls

- mount, umount
- sysfs, statfs, fstatfs, ustat, stat, fstat, lstat, access
- chroot, chdir, fchdir, chown, fchown, lchown, chmod, fcmod,
- getcwd, mkdir, rmdir, readdir, getdents
- link, unlink, rename, readlink, symlink
- open, close, creat, umask, dup, dup2, fcntl, select, poll, truncate, ftruncate, lseek, _llseek, read, write, readv, writev, sendfile, pread, pwrite
- mmap, munmap
- fdatasync, fsync, sync, msync
- flock
- mknod
- socket, coonect, bind, protocols, ...

VFS: Data structures

```
struct super_block          //include/linux/fs.h
 struct file_system_type    //include/linux/fs.h
 struct super_operations    //include/linux/fs.h
struct inode                //include/linux/fs.h
 struct inode_operations    //include/linux/fs.h
struct file                 //include/linux/fs.h
 struct file_operations     //include/linux/fs.h
struct dentry               //include/linux/dcache.h
 struct dentry_operations   //include/linux/dcache.h
struct task_struct          //include/linux/sched.h
  struct fs_struct          //include/linux/fs_struct.h
  struct files_struct       //include/linux/sched.h
    struct file             //include/linux/fs.h
```

VFS: Filesystem registering

```
start_kernel // init/main.c
  vfs_caches_init
    bdev_cache_init
      register_filesystem
register_filesystem  // fs/super.c
  find_filesystem
    static struct file_system_type *file_systems;
```

VFS: Filesystem mounting

Mounting the root filesystem

```
start_kernel
  vfs_caches_init
    mnt_init
      init_mount_tree
        do_kern_mount(
           const char *fstype = "rootfs",
           int flags  = 0,
           char *name = "rootfs",
           void *data = NULL);
```

VFS: Filesystem mounting

Mounting a generic filesystem

```
sys_mount        //fs/namespace.c
  copy_mount_options
  do_mount
    do_remount
    do_loopback
    do_move_mount
    do_add_mount
      do_kern_mount
sys_umount        //fs/namespace.c
  do_umount
```

VFS: sys_open

```
sys_open (fs/open.c)
  getname (fs/namei)
  // to read the file pathname
  // from the process address space
    do_getname
      strncpy_from_user
  get_unused_fd (fs/open.c)
  // to find an empty slot in current-files-fd.
  filp_open
    open_namei
    dentry_open
  fd_install
```

VFS: sys_read

```
sys_read (fs/read_write.c)
  fget
  // to derive from fd the address file of
  // the corresponding file object and
  // increments the usage counter file->f_count
  locks_verify_area
  // to check whether there are mandatory locks
  // for the file portion to be accessed.
  // invokes file->f_op->read to do the job
  fput
  // to decrement the usage counter file->f_count
```

VFS: sys_write

```
sys_write (fs/read_write.c)
  fget
  // to derive from fd the address file of
  //  the corresponding file object and
  //  increments the usage counter file->f_count
  locks_verify_area
  // to check whether there are mandatory locks
  // for the file portion to be accessed.
  // invokes file->f_op->write to do the job
  fput
  // to decrement the usage counter file->f_count
```

VFS: sys_close

```
sys_close (fs/open.c)
  FD_CLR
  __put_unused_fd
  filp_close
    flush
    fcntl_dirnotify
    locks_remove_posix
    fput
```

The ext2 Filesystem

Introduction

- minix Filesystem: Linux was inspired by minix.
- Extended Filesystem (Ext FS): It included several significant extensions, but offered unsatisfactory performance.
- Ext2: Besides including several new features, it is quite efficient and robust and has become the most widely used Linux filesystem.
- Ext3: compatible with the old Ext2 filesystem and a journaling filesystem

Disk data structures

Disk data structures: MBR

OFFSET	LENGTH	NOTE
0x000	0x1BE	Booting the kernel
0x1BE	0x010	Partition 1
0x1CE	0x010	Partition 2
0x1DE	0x010	Partition 3
0x1EE	0x010	Partition 4
0x1FE	0x002	0xAA55

Disk data structures: Partition entry

OFFSET	LENGTH	NOTE
00h	1	80h = active partition / 00h = not active
01h	1	begin of partition (head number)
02h	1	begin of partition (sector number) [*]
03h	1	begin of partition (cylinder number) [*]
04h	1	partition ID
05h	1	end of partition (head number)
06h	1	end of partition (sector number) [*]
07h	1	end of partition (cylinder number) [*]
08h	4	rel. sectors (# sec. to begin of partition)
0Ch	4	number of sectors in partition

Note: CHS/LBA

Disk data structures

- To read in the raw data
  `dd if=/dev/hda  bs=512 count=1      >/tmp/dump_hda`
- To view binary data
  `od -tx1 -Ax /tmp/dump_hda`
- To disassemble binary code.
  `ndisasm /tmp/dump_hda`
  `# see GRUB source code`

Disk data structures

An ext2 partition consists of

- one boot block
- and many block groups

A block group consists of

- A copy of the filesystem's superblock
- A copy of the group of block group descriptors
- A data block bitmap
- A group of inodes
- An inode bitmap
- A chunk of data that belongs to a file; i.e., a data block

Disk data structures: ext2_super_block

```
struct super_block
  struct ext2_super_block
  struct ext2_group_desc
struct inode
  struct ext2_inode_info
struct ext2_dir_entry_2
```

Disk data structures

To Dump filesystem information

`dumpe2fs  /dev/had*`

To dump filesystem data and view it

`dd`

`cc`

To compare the data

Memory data structures

Type	Disk data structure	Memory data structure	Caching
Superblock	ext2_super_block	ext2_sb_info	Cached
Group descriptor	ext2_group_desc	ext2_group_desc	Cached
Block bitmap	Bit array in block	Bit array in buffer	Fixed
Inode bitmap	Bit array in block	Bit array in buffer	Fixed
Inode	ext2_inode	ext2_inode_info	Dynamic
Data block	Unspecified	Buffer page	Dynamic
Free inode	ext2_inode	None	Never
Free block	Unspecified	None	Never

The Ext2 Filesystem Initialization

1. Initializes the superblock and the group descriptors.
2. Optionally, checks whether the partition contains defective blocks; if so, it creates a list of defective blocks.
3. For each block group, reserves all the disk blocks needed to store the superblock, the group descriptors, the inode table, and the two itmaps.
4. Initializes the inode bitmap and the data map bitmap of each block group to 0.
5. Initializes the inode table of each block group.
6. Creates the /root directory.
7. Creates the lost+found directory, which is used by e2fsck to link the lost and found defective blocks.
8. Updates the inode bitmap and the data block bitmap of the block group in which the two previous directories have been created.
9. Groups the defective blocks (if any) in the lost+found directory.

The Ext2 Filesystem Operation

Ext2 superblock operation

```
    struct super_block
      struct super_operations
        static struct super_operations
```

Ext2 Inode Operations:

```
  struct inode
  struct inode_operations
    ext2_file_inode_operations

  struct file_operation
    ext2_file_operations
```

Process Management

PCB

- struct task_struct (include/linux/sched.h)
- union task_union (include/linux/sched.h)

- free_task_struct() // include/asm-i386/process.h
- alloc_task_struct() // include/asm-i386/process.h
- init_task_union (arch/i386/kernel/init_task.c)
  - init_task
- current (include/asm-i386/current.h)

PCB
// kernel/sched.c
struct task_struct * init_tasks[NR_CPUS] = {&init_task, };
// (include/linux/sched.h)
#define for_each_task(p) for (p = &init_task ; (p = p->next_task) != &init_task ;)
// SET_LINKS
// REMOVE_LINKS
PCB: Running list
struct task_struct;
  struct list_head run_list;
// (kernel/sched.c)
add_to_runqueue
move_last_runqueue
move_first_runqueue
PCB: PidHash
task_struct
    struct task_struct *pidhash_next;
    struct task_struct **pidhash_pprev;
struct task_struct *pidhash[PIDHASH_SZ];
// kernel/fork.c
// (include/linux/sched.h)
hash_pid
unhash_pid
find_task_by_pid
PCB: Wait queue
// include/linux/wait.h
   __add_wait_queue
   __add_wait_queue_tail __remove_wait_queue
PCB: Parenthood relatinoships
Parenthood relationships among processes
struct task_struct {
  .....
  struct task_struct *p_opptr, // original parent
      *p_pptr, // current parent
      *p_cptr, // youngest child
      *p_ysptr, // yonger sibling
      *p_osptr; // older sibling
  .....
}
sys_fork, sys_clone, sys_vfork
// arch/i386/kernel/process.c
sys_fork
  do_fork
sys_clone
  do_fork
sys_vfork
  do_fork
    // kernel/fork.c
    do_fork
do_fork
alloc_task_struct // get memory for the task_struct
get_exec_domain
// current
// get_current (include/asm-i386/current.h)
copy_flags, get_pid
init_waitqueue_head, init_completion, init_sigpending, init_timer
copy_files, copy_fs, copy_sighand, copy_mm, copy_thread
SET_LINKS, hash_pid
sys_exit
// kernel/exit.c
sys_exit
  do_exit
    __exit_mm
    __exit_files;
    __exit_fs;
    exit_sighand;
    exit_thread;

    put_exec_domain
    exit_notify
    schedule
Process Switching
asmlinkage void schedule(void)
// kernel/sched.c
  switch_to(prev,next,last)
  // include/asm-i386/system.h
    void __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
    // arch/i386/kernel/process.c

Booting

The CPU
- After receiving an active level on its RESET input pin
- Optional built-in self-test (BIST): The EAX register should be 0; otherwise, you may have a faulty processor
- The EDX register contains processor type (DH) and revision infos (DL).
- The physical address at which the first instruction must be placed is FFFFFFF0 (BIOS). This is exactly 16 bytes before the absolute high end of the 4GB address space.
  - CS is F000 and IP is FFF0 → 000FFFF0 for 286
  - Other CPUs boost all CS-relative addresses after reset.

The BIOS
- The BIOS performs the following four operations
- Executes a series of tests on the computer hardware. POST(Power-on Self-Test)
- Initializes the hardware devices (IRQs and I/O ports)
- Searches for an operating system to boot.
- Copies the contents of the boot sector into RAM, starting from physical address 0x00007C00, then jumps into that address and executes the code just loaded. (This code is just the boot loader)
  - The BIOS Bootstrap Loader function is invoked via **int 0x19**, with %dl containing the boot device 'drive number'. This loads track 0, sector 1 at physical address 0x7C00 (0x07C0:0000).

The Boot Loader: Floppy disks
- Move itself from address 0x00007C00 to address 0x00090000
- Set up the Real Mode stack.
- Set up the disk parameter table used by the BIOS to handle the floppy device driver
- Invoke a BIOS procedure to display a "loading" message
- Invoke a BIOS procedure to load the setup() code of the kernel image from the floppy disk and puts it in RAM starting from address 0x00090200.
- Invoke a BIOS procedure to load the rest of the kernel image from the floppy disk and puts the image in RAM starting either low address 0x00010000 or high address 0x00100000
- Jump to the setup() code
- For more, see arch/i386/boot/{bootsect.S, setup.S, video.S}

The Boot Loader: Hard disks
- Booting Linux from hard disks
- Use lili or grub to load the kernel into the RAM
- Jumps to the setup() code

The setup() Function
- setup() (start_of_setup (arch/i386/boot/setup.S))
- Read second hard drive DASD type
- Check that LILO loaded us right
- Check old loader trying to load a big kernel
- Determine system memory size
- Get video adapter modes
- Get Hard Disk parameters
- Check for Micro Channel (MCA) bus
- Check for mouse
- Check for APM BIOS support
- Prepare to move to protected mode (LMSW)
- Jump to the startup_32 assembly language function (linux/arch/i386/kernel/head.S)

The startup_32() Function
- startup_32 (linux/arch/i386/kernel/head.S)
- Set segments to known values
- Initialize page tables
- Enable paging
- Clear BSS first so that there are no surprises...
- Start system 32-bit setup. Initialize eflags.
- Copy bootup parameters out of the way.
- Check CPU type (check_x87)
- Configure for SMP
- Jump to start_kernel

The start_kernel() Function
- Take a global kernel lock (it is needed so that only one CPU goes through initialisation).
- Perform arch-specific setup (memory layout analysis, copying boot command line again, etc.).
- Print Linux kernel "banner" containing the version, compiler used to build it etc. to the kernel ring buffer for messages. This is taken from the variable linux_banner defined in init/version.c and is the same string as displayed by cat /proc/version.
- Initialise traps.
- Initialise irqs.
- Initialise data required for scheduler.
- Initialise time keeping data.
- Initialise softirq subsystem.
- Parse boot commandline options.
- Initialise console.

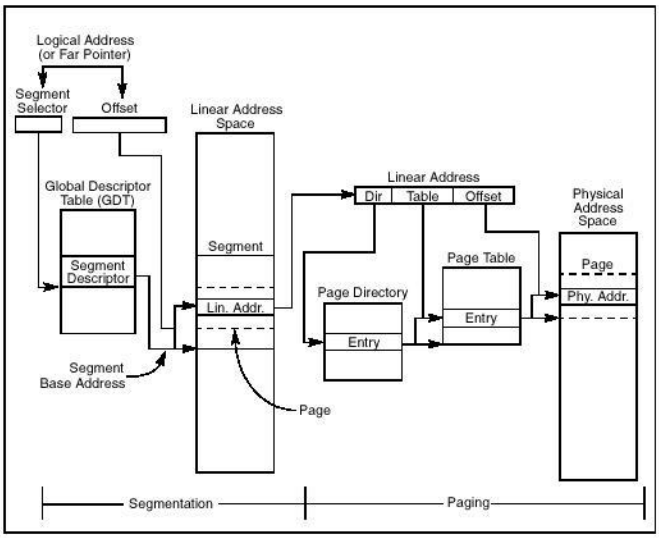
The startup_32() Function
- If module support was compiled into the kernel, initialise dynamical module loading facility.
- If "profile=" command line was supplied, initialise profiling buffers.
- kmem_cache_init(), initialise most of slab allocator.
- Enable interrupts.
- Calculate BogoMips value for this CPU.
- Call mem_init() which calculates max_mapnr, totalram_pages and high_memory and prints out the "Memory: ..." line.
- kmem_cache_sizes_init(), finish slab allocator initialisation.
- Initialise data structures used by procfs.

The startup_32() Function
- fork_init(), create uid_cache, initialise max_threads based on the amount of memory available and configure RLIMIT_NPROC for init_task to be max_threads/2.
- Create various slab caches needed for VFS, VM, buffer cache, etc.
- If System V IPC support is compiled in, initialise the IPC subsystem. Note that for System V shm, this includes mounting an internal (in-kernel) instance of shmfs filesystem.
- If quota support is compiled into the kernel, create and initialise a special slab cache for it.
- Perform arch-specific "check for bugs" and, whenever possible, activate workaround for processor/bus/etc bugs. Comparing various architectures reveals that "ia64 has no bugs" and "ia32 has quite a few bugs", good example is "f00f bug" which is only checked if kernel is compiled for less than 686 and worked around accordingly.

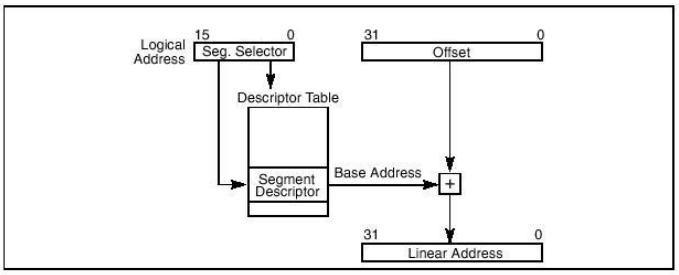
The startup_32() Function
- Set a flag to indicate that a schedule should be invoked at "next opportunity" and create a kernel thread init() which execs execute_command if supplied via "init=" boot parameter, or tries to exec /sbin/init, /etc/init, /bin/init, /bin/sh in this order; if all these fail, panic with "suggestion" to use "init=" parameter.
- Go into the idle loop, this is an idle thread with pid=0.

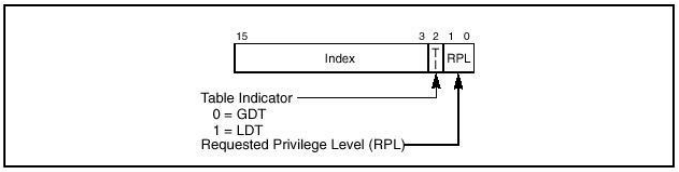
Memory Addressing:

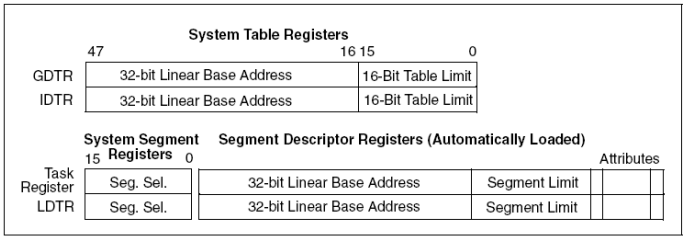
3 addresses



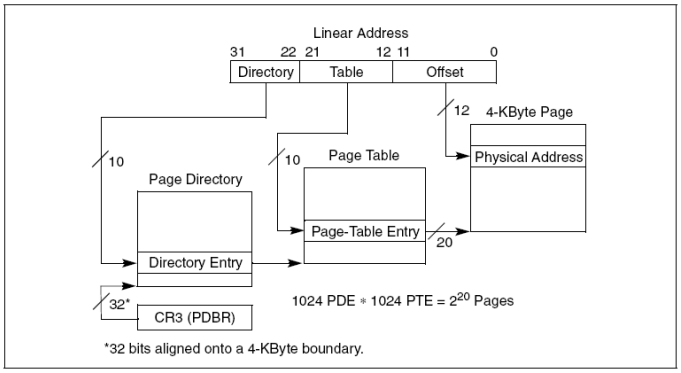
I386 segmentation

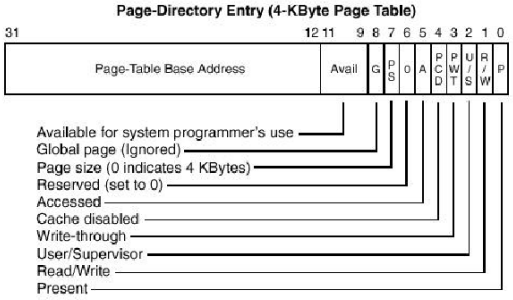


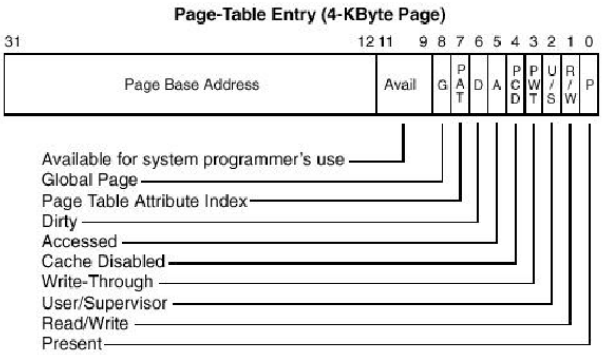




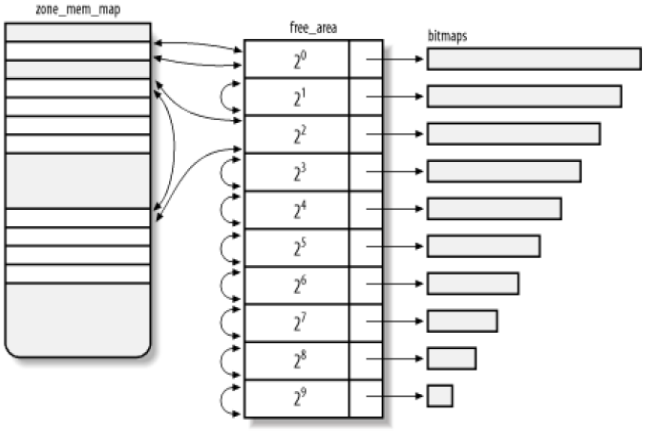
I386 Paging



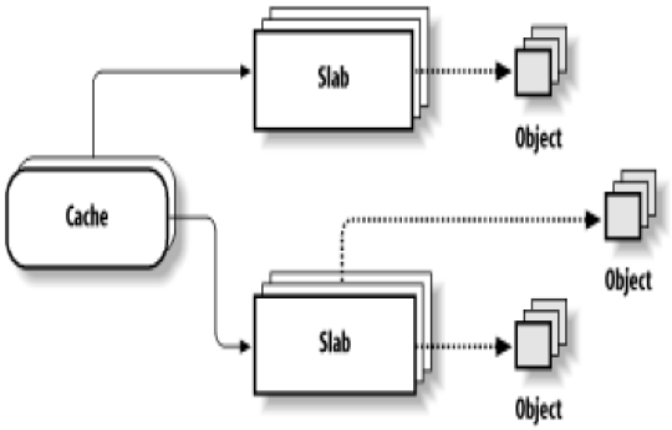




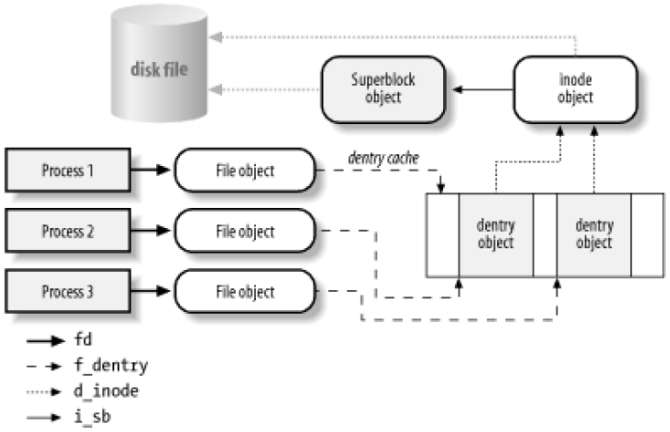
Page Frame Management



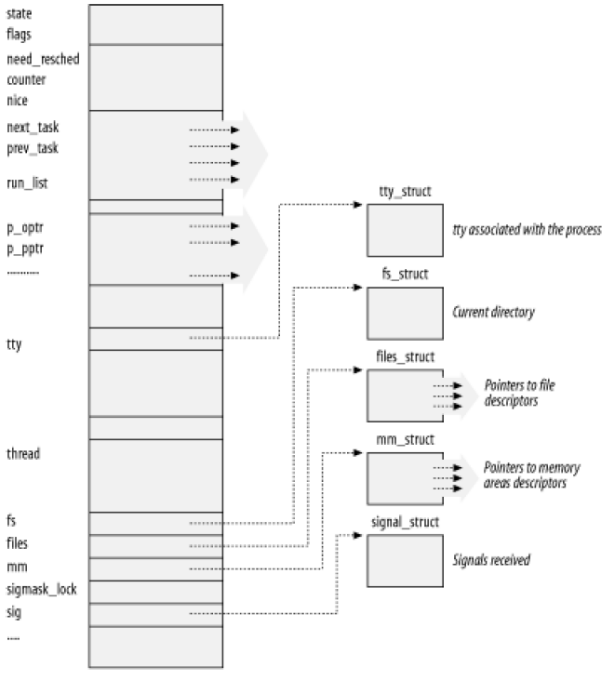
Memory Area Management



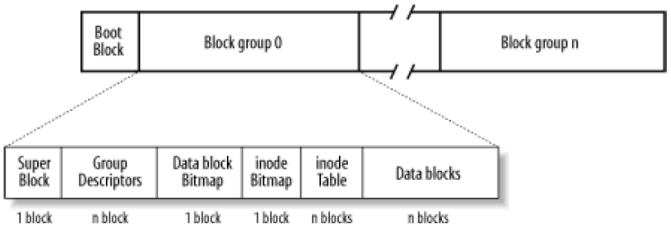
VFS: Interaction with processes



PCB task_struct



Disk data structures



Processes ◊The Ext2