Example: Linux and UNIX

1

UNIX History

- First version, a stripped-down MULTICS by Ken Thompson for a discarded PDP-7 minicomputer in assembly language
- Moved to PDP-11 and rewritten using language, C, designed for that purpose
- Evolved to two different, partially incompatible versions
  - System V (AT&T, original designers)
  - BSD (Berkeley, based on an early UNIX Version 6)
- Standardization by IEEE created POSIX 1003.1
  - Intersection *(leikkaus)* of the features

2

UNIX

- Popularity based on
  - Portability
    - C code, some device-dependent parts in assembly code
  - Multiprogramming
  - Supported interactive users (at the era of batch systems)
  - Hierarchical file and directory structure
  - Just one file format: sequence of bytes
  - (Universities had the suitable machines in the beginning)
- UNIX philosophy: *small is beautiful*
  - Combining small utility programs (called filters)
    - pipe output of one to be an input to another
  - Minimum number of system calls

3

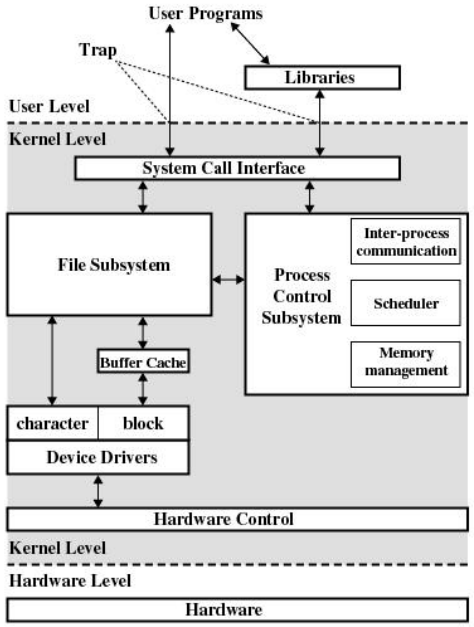


Figure 2.15 Traditional UNIX Kernel [BACH86]

UNIX History, continues

- MINIX (released in 1987)
  - Designed by A. Tanenbaum for educational purpose
  - Microkernel design
  - Today focus also on reliability and dependability issues
- LINUX (released in 1991)
  - Designed by Linus Torvalds for personal use
    - Model from MINIX, but monolithic kernel
  - Uses special features of gcc
  - Free software based on GPL (Gnu Public License)

5

Linux

- Joint Internet-based development
  - Experts all over the world
  - 1991 ->
- HY/TKTL: *"Linux was invented here"*
  - Linus Torvalds studied and worked at the department. He started the work after *this course.*

- Free Software Foundation & GNU Public License
  - Free distribution of the kernel code (C & assembler)
  - Free distribution of some (most) system software
    - GNU C, Gnome, KDE, Apache, Samba, …
- Several commercial distributions
  - Ubuntu, RedHat, SuSe, Debian, Mandrake, TurboLinux, etc...
  - Varies over time

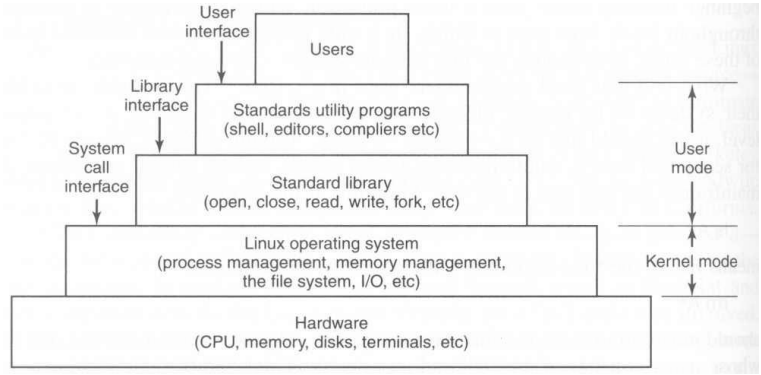
6

Linux

- Combining POSIX + SysV and BSD
  - LSB: Linux Standard Base
    - must be in all distributions
- Running on multiple architectures
  - i386, IA64, Alpha, MIPS
- Also in embedded systems (*upotetuissa järjestelmissä)*
  - PDAs digital-TV, refrigerator, ”wrist computer"...
- Configurable
  - During compilation include only necessary parts to kernel
  - Even code can be modified
- Optimizable
  - Different needs on different platforms
  - You are allowed to modify the code
    - Copyleft, GNU General Public License (GNU GPL)

7

UNIX / LINUX interfaces



- The library interface actually hides the system calls.
- User programs just call procedures (that then internally do the ‘trap’ command in assembly code)

8

Shell: command-line interface

- For programmers and sophisticated users
- Faster to use, more powerful than GUI
- Ordinary user program that assumes that each line is a command and first word is the program name
  - Arguments and flags -f, -h
  - Magic characters (wild cards) *? [abcABC]
  - Redirection < >
  - Pipe |
- Examples
  - sort <in | head -30
  - grep ter*.t | sort | head -15 | tail -5 >foo

9

Some POSIX Utility Programs

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
ps	List running processes
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

All of these and much more is available from shell interface

10

UNIX Kernel

<table>
<tr><td colspan="4">System calls</td><td colspan="3">Interrupts and traps</td></tr>
<tr><td colspan="2">Terminal handing</td><td>Sockets</td><td>File naming</td><td>Map-ping</td><td>Page faults</td><td>Signal handling</td><td>Process creation and termination</td></tr>
<tr><td rowspan="2">Raw tty</td><td>Cooked tty</td><td>Network protocols</td><td>File systems</td><td colspan="2">Virtual memory</td><td rowspan="2">Process scheduling</td></tr>
<tr><td>Line disciplines</td><td>Routing</td><td>Buffer cache</td><td colspan="2">Page cache</td></tr>
<tr><td colspan="2">Character devices</td><td>Network device drivers</td><td colspan="2">Disk device drivers</td><td colspan="2">Process dispatching</td></tr>
<tr><td colspan="7">Hardware</td></tr>
</table>

Approximate structure of generic UNIX kernel

11

Linux kernel

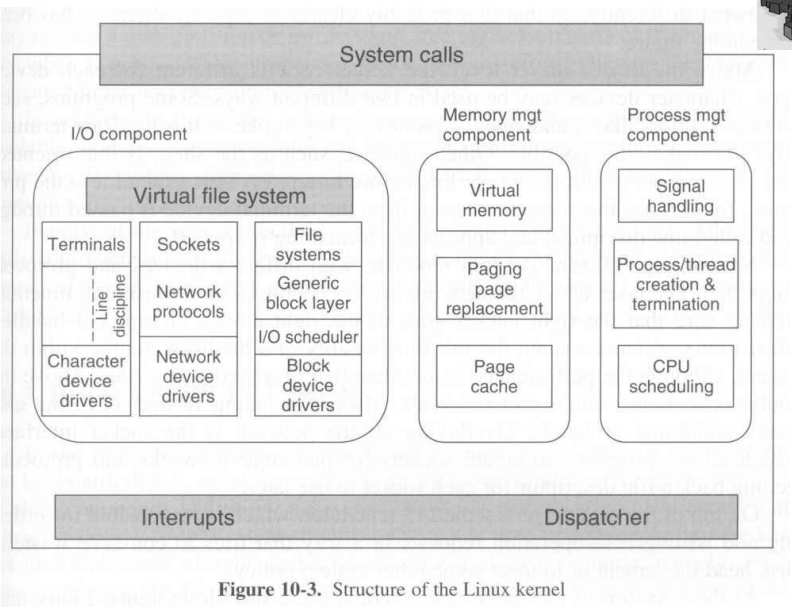


Figure 10-3. Structure of the Linux kernel

12

Process management

13

Linux processes

- Called tasks
- Each runs a single program, initially has a single thread of control
- Kernel treats processes and threads similarly
- Daemons, background processes
  - Started by a shell script when system is booted
  - Run periodically
- Parents can start, fork, child processes
- Process communication: pipe, signal

14

Linux: process states

- TASK_RUNNING
  - On CPU or waiting for CPU (ready-queue)
- TASK_INTERRUPTIBLE
  - Waiting, that can be interrupted using for example signal
- TASK_UNINTERRUPTIBLE
  - Waiting for device related operation, that must be finished
  - Does not accept signals (for example “kill -9 1234”)
- TASK_STOPPED
  - Task that is waiting for user operation. For example, background task that needs to read from keyboard or write to screen
  - Got a signal SIGSTOP, SIGTTIN tai SIGTTOU
  - Now waits for dignal SIGCONT
- TASK_ZOMBIE
  - Finised task, waiting for parent to recognize and clean up

15

Linux Process / Thread states

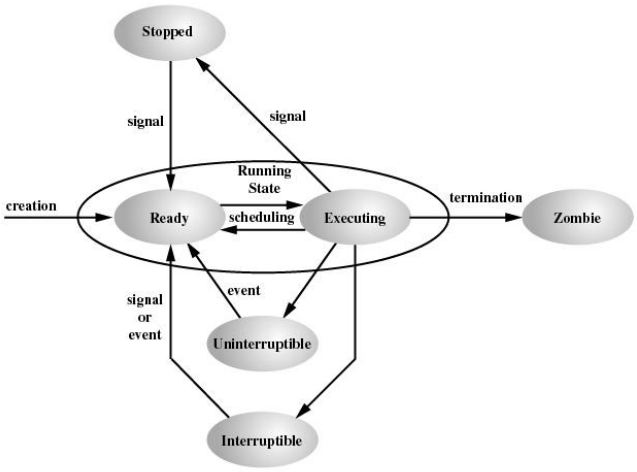


Figure 4.18 Linux Process/Thread Model

Fig 4.18 [Stal05]

16

POSIX signals

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

- The signals required by POSIX.
- Using other signals may violate portability.

17

System Calls for Process Management

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

s is an error code
pid is a process ID
residual is the remaining time from the previous alarm

18

POSIX Shell

```
while (TRUE) {                              /* repeat forever /*/
     type_prompt( );                        /* display prompt on the screen */
     read_command(command, params);         /* read input line from keyboard */

     pid = fork( );                         /* fork off a child process */
     if (pid < 0) {
          printf("Unable to fork0);         /* error condition */
          continue;                         /* repeat the loop */
     }

     if (pid != 0) {
          waitpid (–1, &status, 0);         /* parent waits for child */
     } else {
          execve(command, params, 0);       /* child does the work */
     }
}
```

A highly simplified shell

19

Process descriptor - task_struct

- Scheduling parameters
  - Priority, CPU time consumed, time spent sleeping
- Memory image
  - Page table pointer or text, data, and stack segment pointers
- Signal masks
- Machine registers
- System call state
- File descriptor table
- Kernel stack
- Accounting
- Miscellaneous

`include/linux/sched.h`

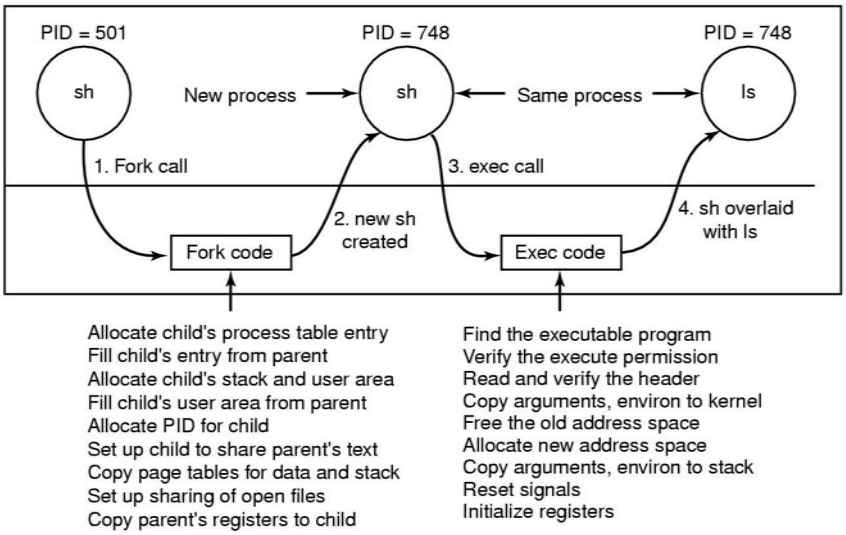
20

Process creation

- Create process descriptor and user area
- Fill descriptor mostly from parent
- Own PID, memory map, kernel mode stack
- Link with previous/next process on task array
- Memory allocation
  - Full copy too expensive
  - Copy-on-write
    - Point to parent's pages as read-only
    - Make own copy when trying to write to the page
- Child quite often change the code using exec system call

21

Process creation: the *ls* command



Steps in executing the command *ls* typed to the shell

22

Linux thread support

- Unique approach: clone system call
  - Blurr the distinction between process and thread
  - Create new thread in current process (shared address space) or in new process (not shared address space) – CLONE_VM
- Fine grain control of sharing

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PID	New thread gets old PID	New thread gets own PID

Bits in the sharing_flags bitmap

23

Scheduling (2.6 kernel)

sched_setscheduler()

- Classes of tasks
  - Real-time FIFO : non-preemptable, except by higher-priority task, SCHED_FIFO
  - Real-time round robin: processes have priority and time quanta, SCHED_RR
  - Timesharing: normal processes, SCHED_OTHER
  - NOTE: ”real-time processes” do not have deadlines, or execution guarantees
- Priorities
  - 0..99 for real-time threads, 0 highest
  - 100..139 for timesharing threads
  - Different levels get different time quantums

kernel 2.6 [DDC04]

24

Linux: task priority

- *Nice* (value) system call
  - Basic value 0
  - 0 .. +19 all users, user can lower the priority
  - -20 .. -1 for root (and OS) only
  - Static priority value
- Dynamic (effective) priority *epri* = *nice* ±5
- Reward interactive or I/O-bound tasks
  - Maximum bonus available -5
  - Gain when 'voluntarily' give up CPU in the middle of quantum
- Punish CPU-hogging tasks
  - Maximum penalty available +5
  - Increment when task is preempted or quantum expires
- Result: CPU-bound processes get the service that is left over by all I/O and interactive processes

25

Scheduling conventional threads

- Quantum is number of clock ticks called jiffies
- Runqueue
  - Two arrays: active, expired
- Select a task from highest-priority active arrays list
- Task's quantum expires -> move to expired array
- No more tasks in the active array -> swap arrays

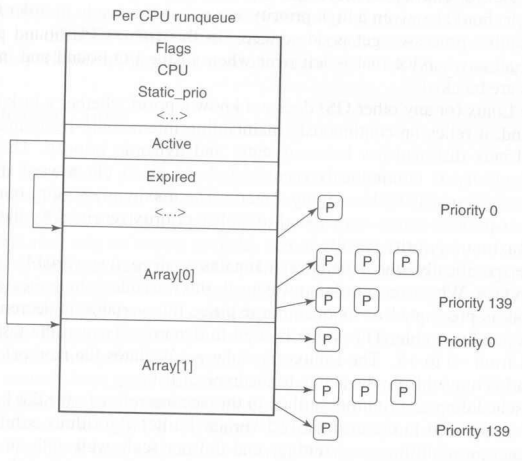


Figure 10-10. Illustration of Linux runqueue and priority arrays.

26

Booting

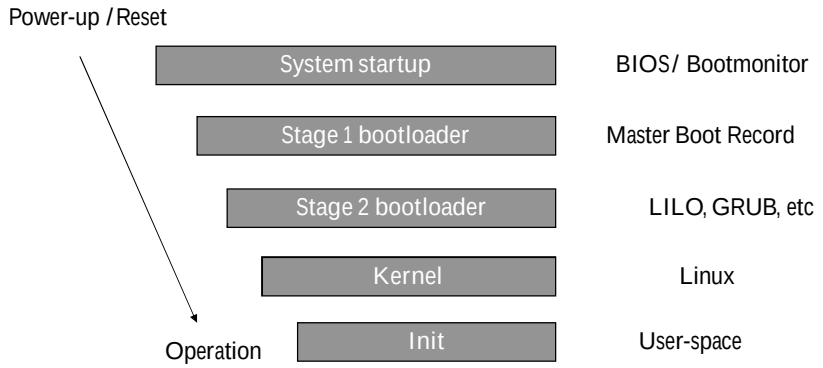
27

Booting

- Key problem: How do you initiate a system using only itself?
- Booting <- Bootstraping
- Bootstraping - why this term?
  - Baron Munchausen pulled himself out of swamp using his boot straps

28

View of the Linux boot process



Source: http://www-128.ibm.com/developerworks/library/l-linuxboot/index.html

29

BIOS tasks

- BIOS – Basic Input / Output System
  - BIOS refers to the firmware code run by a personal computer when first powered on.
  - BIOS can also be said to be a coded program embedded on a chip that recognizes and controls various devices that make up x86 personal computers. (source: wikipedia)
- Execution starts from a fixed location (on x386 that location is 0xFFFF0000)
- Check hardware, locate the boot device
  - Disk, CD, ...
- Load the first 'loader' from Master Boot Record

30

BIOS example sequence

1. Check the CMOS Setup for custom settings
2. Load the interrupt handlers and device drivers
3. Initialize registers and power management
4. Perform the power-on self-test (POST)
5. Display system settings
6. Determine which devices are bootable
7. Initiate the bootstrap sequence

31

BIOS loading the boot loader – pseudocode example

0: set the P register to 8
1: check paper tape reader ready
2: if not ready, jump to 1
3: read a byte from paper tape reader to accumulator
4: if end of tape, jump to 8
5: store accumulator to address in P register
6: increment the P register
7: jump to 1

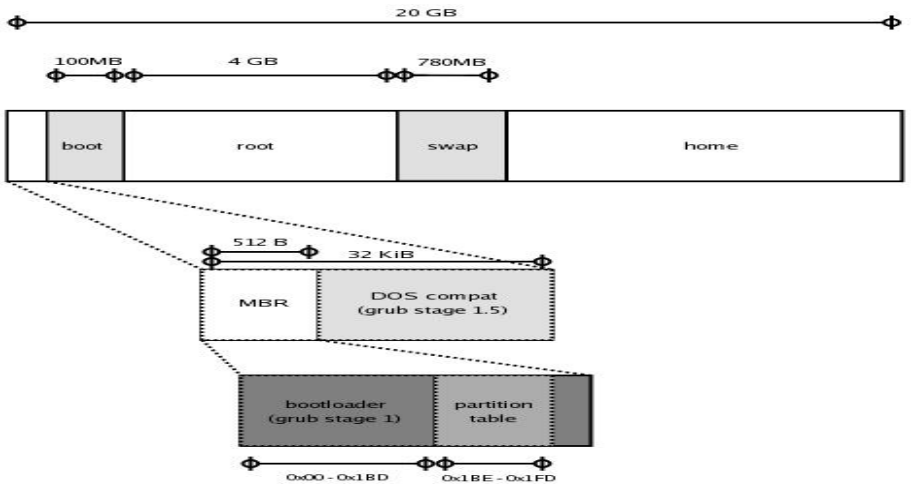
32

Linux boot loader GRUB - GRand Unified Bootloader

- GRUB is independent of any particular operating system and may be thought of as a tiny, function-specific OS.
- It's primary task is to load the actual operating system and pass control to it.
- GRUB is large service:
  - It does not fit to MBR: needs to be handled in two phases
  - Phase 1 (loaded from MBR by BIOS) load phases 1.5 and 2
- A lot of features: file systems, command line interface, chain-loading other boot loaders, ...

33

GRUB on disk



http://www.pixelbeat.org/docs/disk/

34

Loading Linux

- Bootsector loads setup, decompression routines and compressed kernel image.
- The kernel is uncompressed in protected mode.
- Low-level initialisation is performed by asm code arch/i386/kernel/head.S:
  - Initialise segment values and page tables.
  - Enable paging by setting PG bit in %cr0.
  - Copy the first 2k of bootup parameters (kernel commandline).
  - The first CPU calls start_kernel(), all others call initialize_secondary().
- High-level C initialisation

Source: http://www.faqs.org/docs/kernel_2_4/lki-1.html

35

OS Initialising: start_kernel()

- Arch-specific setup (memory layout analysis, copying boot command line again, etc.).
- Initialise traps, irqs, data required for scheduler, time keeping data, softirq subsystem.
- Initialise console.
- Enable interrupts.
- Set a flag to indicate that a schedule should be invoked at "next opportunity"
- Create a kernel thread init()
- Go into the idle loop, this is an idle thread with pid=0.

Important thing to note here that the init() kernel thread calls do_basic_setup() which in turn calls do_initcalls()

More detailed version available from: http://www.faqs.org/docs/kernel_2_4/lki-1.html

36

Process Init

- Init is the father of all processes. Its primary role is to create processes from a script stored in the file /etc/inittab (man init)
- Establishes and operates the entirety of user space.
  - checking and mounting file systems,
  - starting up necessary user services, and
  - switching to a user-based environment when system startup is completed
- Uses /etc/rc directory hierarchy

37

/etc/rc.d

- debian: run /etc/init.d/rcS which runs:
- /etc/rcS.d/S* scripts
- /etc/rc.boot/* (depreciated)
- run programs specified in /etc/inittab

- Scripts in /etc/rc*.d/* are symlinks to /etc/init.d
- Scripts prefixed with S will be started when the runlevel is entered, eg /etc/rc5.d/S99xdm
- Scripts prefixed with K will be killed when the runlevel is entered, eg /etc/rc6.d/K20apache
- Executed in numerical order

38

http://www.yolinux.com/TUTORIALS/LinuxTutorialInitProcess.html

39

First processes

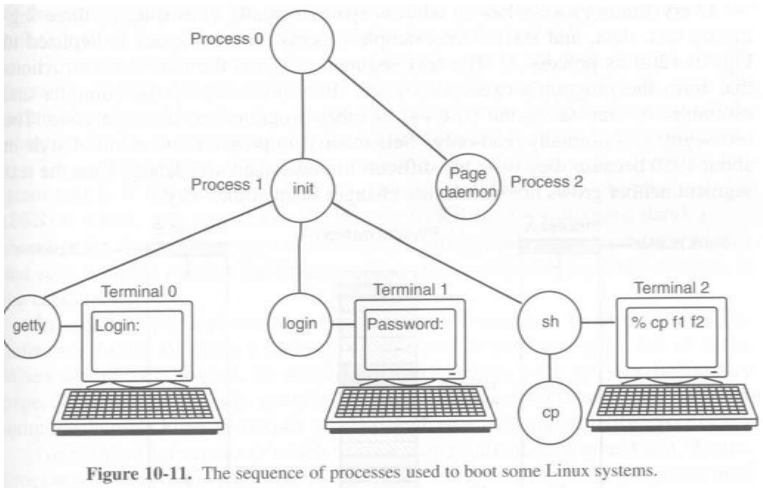


Figure 10-11. The sequence of processes used to boot some Linux systems.

40

Memory management

41

Main memory representation

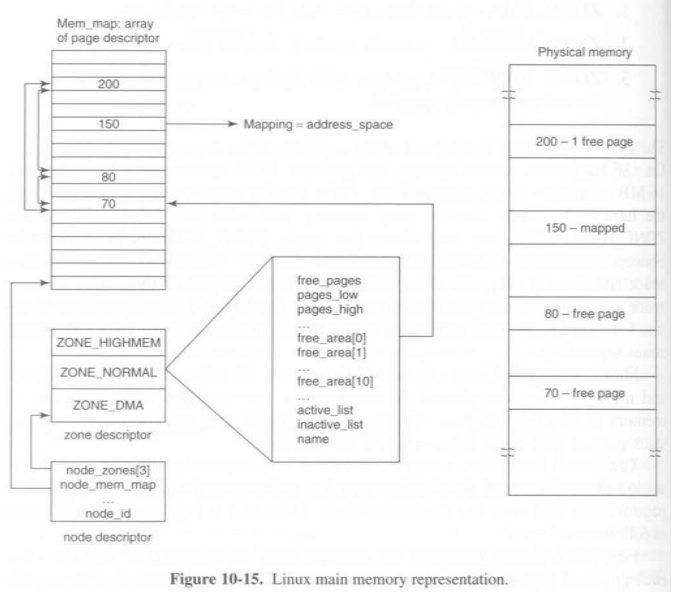


Figure 10-15. Linux main memory representation.

42

Linux: memory management

- 32-bit architectures
  - 3 GB reserved for user processes
  - 1 GB for kernel (including the page tables)
    - at the beginning of the physical memory
  - Kernel mode allowed to make references to the full 4 GB
- Kernel's memory allocation based on slabs
  - Only for the kernel, efficient allocation of small memory areas, avoids allocating full pages

43

Linux page tables

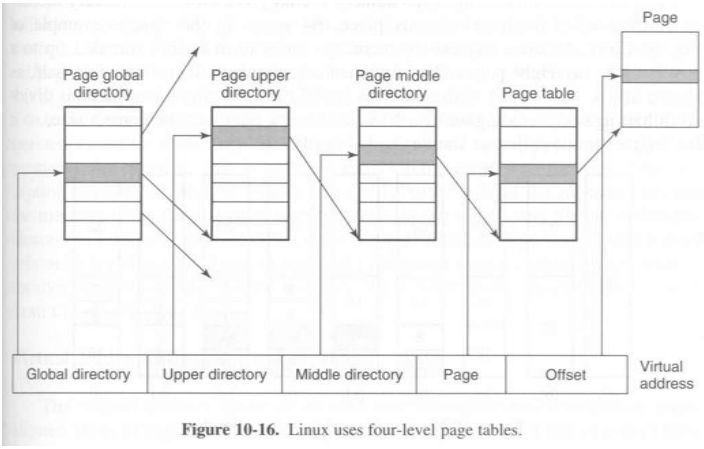


Figure 10-16. Linux uses four-level page tables.

44

Linux: kernel memory allocation

- Kernel memory locked to the memory
- Dynamic allocation based on pages
- Dynamically loadable modules fully to the kernel area
  - buddy system on the pages kmalloc()
- Additional need for small, short term memory allocations -> use slab
  - One page 'cut' to smaller areas (slabs)
  - buddy system within each page
    - x86: 32, 64, 128, 252, 508, 2040, 4080 bytes (page size 4KB)

45

Slap allocation

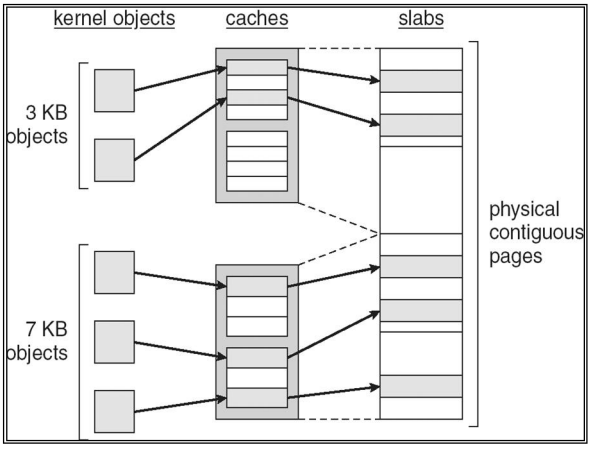


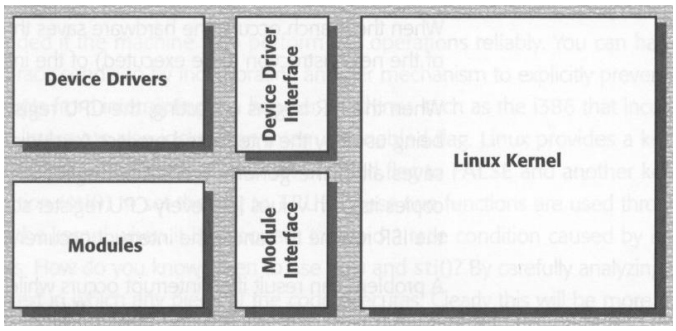
Fig 9.27 – Silberschatz, Galvin & Gagne: Operating system concepts with Java

46

Loadable modules

47

Linux: Loadable modules



- Modules must be registered in kernel
  - init_module(), delete_module(), …
  - register_blkdev(), unregister_blkdev(), …
  - register_filesystem(), unregister_filesystem(), …

48

Linux: Loadable modules

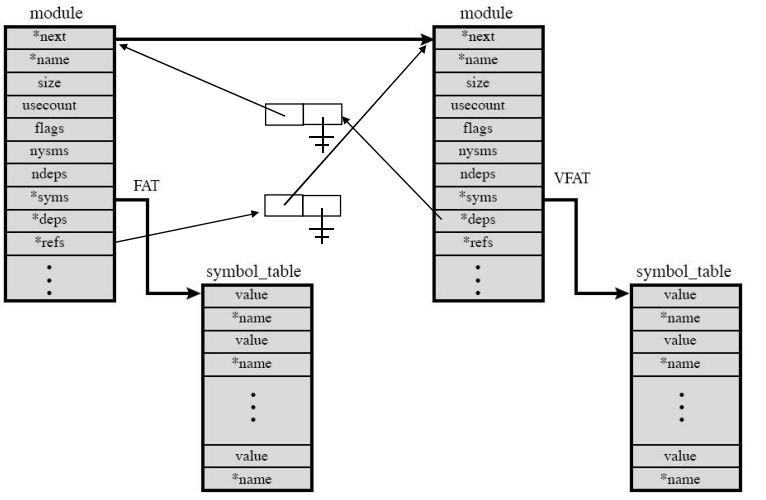


Fig 2.17 [Stal05]

49

Linux: More information

Kernel implementation principles

- Bovet D.P., Cesati M.: *Understanding the Linux Kernel*. O'Reilly, 2nd ed., 2003.
- Beck M., Böhme H. & al. : *Linux Kernel Programming*. Addison-Wesley, 3rd ed., 2002
- Rubini A., Corbet J.: *Linux Device Drivers*. O'Reilly. 2001. 2nd ed.

Cross referenced source code

- http:// lxr.linux.no

50