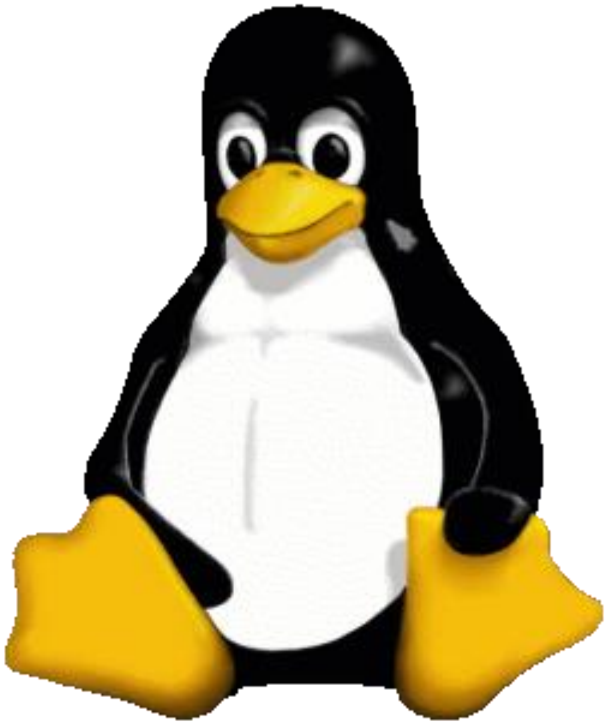
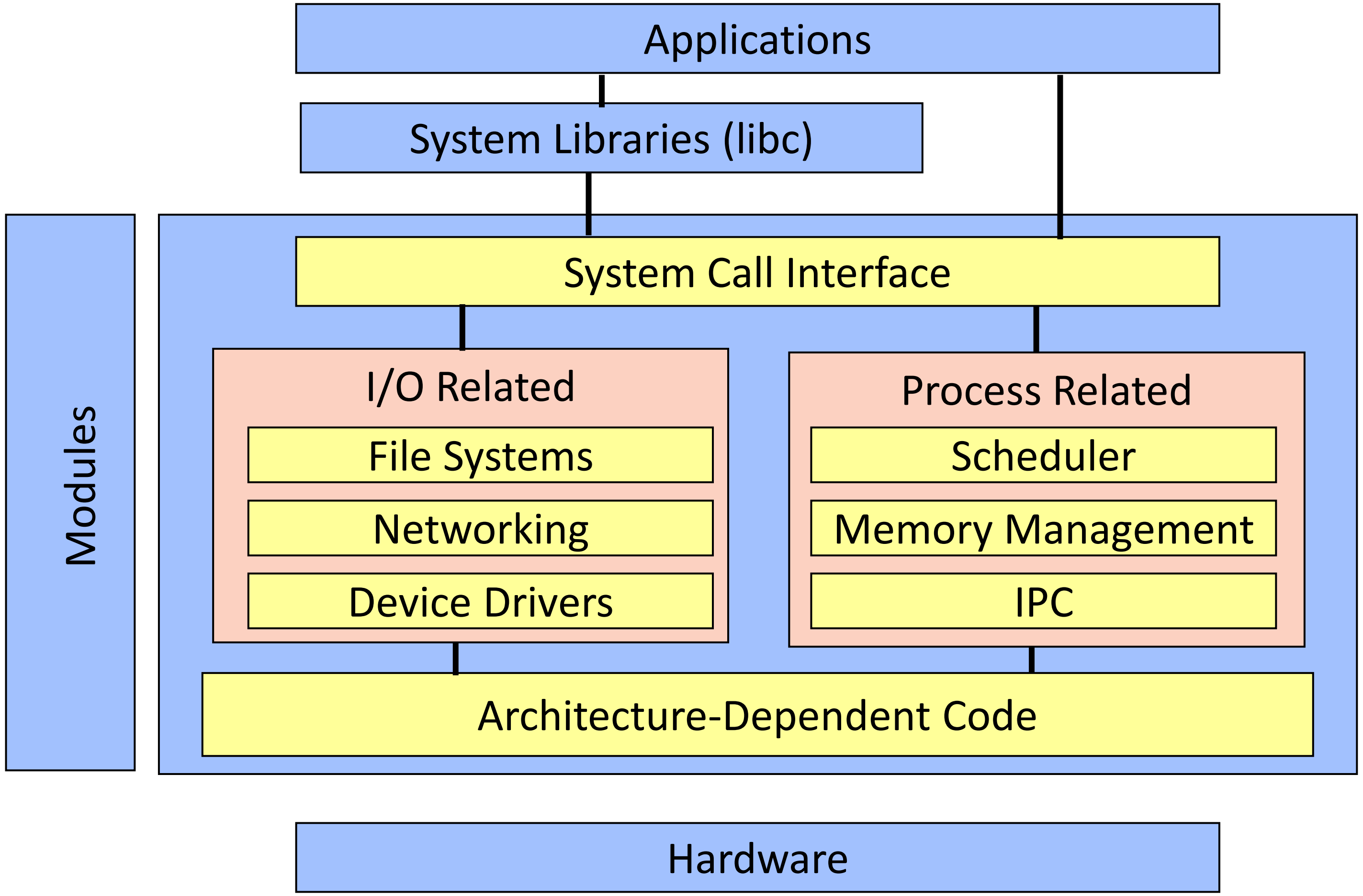
The Linux Kernel: Introduction



Kernel Design Goals

- Performance: efficiency, speed.
    - Utilize resources to capacity with low overhead.
- Stability: robustness, resilience.
    - Uptime, graceful degradation.
- Capability: features, flexibility, compatibility.
- Security, protection.
    - Protect users from each other & system from bad users.
- Portability.
- Extensibility.

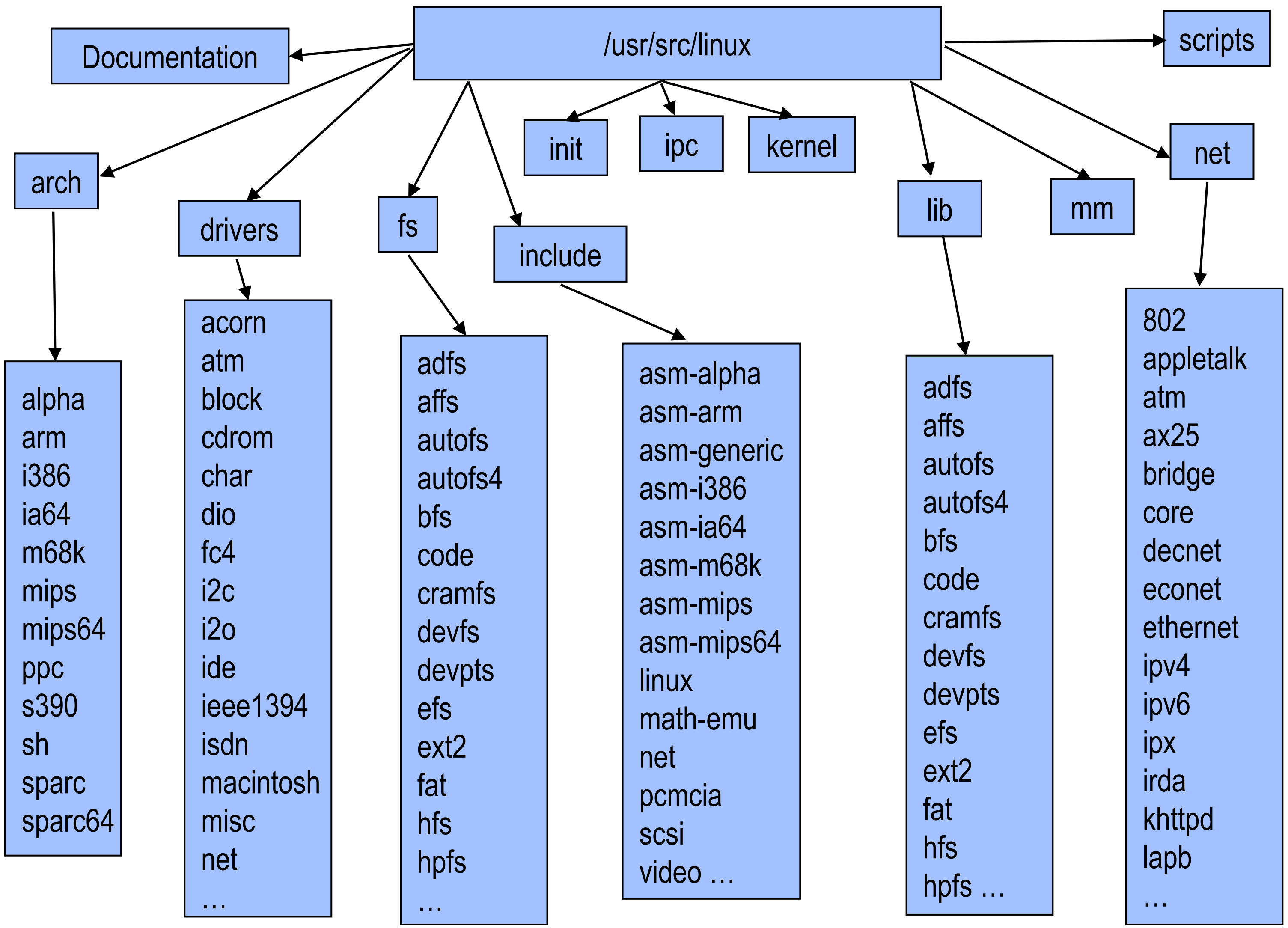
Example "Core" Kernel



Architectural Approaches

- Monolithic.
- Layered.
- Modularized.
- Micro-kernel.
- Virtual machine.

Linux Source Tree Layout



linux/arch

- Subdirectories for each current port.
- Each contains **kernel**, **lib**, **mm**, **boot** and other directories whose contents override code stubs in architecture independent code.
- **lib** contains highly-optimized common utility routines such as memcpy, checksums, etc.
- **arch** as of 2.4:
  - alpha, arm, i386, ia64, m68k, mips, mips64.
  - ppc, s390, sh, sparc, sparc64.

linux/drivers

- Largest amount of code in the kernel tree (~1.5M).
- device, bus, platform and general directories.
- drivers/char – n_tty.c is the default line discipline.
- drivers/block – elevator.c, genhd.c, linear.c, ll_rw_blk.c, raidN.c.
- drivers/net –specific drivers and general routines Space.c and net_init.c.
- drivers/scsi – scsi_*.c files are generic; sd.c (disk), sr.c (CD-ROM), st.c (tape), sg.c (generic).
- General:
  - cdrom, ide, isdn, parport, pcmcia, pnp, sound, telephony, video.
- Buses – fc4, i2c, nubus, pci, sbus, tc, usb.
- Platforms – acorn, macintosh, s390, sgi.

linux/fs

- Contains:
  - virtual filesystem (VFS) framework.
  - subdirectories for actual filesystems.
- vfs-related files:
  - exec.c, binfmt_*.c - files for mapping new process images.
  - devices.c, blk_dev.c – device registration, block device support.
  - super.c, filesystems.c.
  - inode.c, dcache.c, namei.c, buffer.c, file_table.c.
  - open.c, read_write.c, select.c, pipe.c, fifo.c.
  - fcntl.c, ioctl.c, locks.c, dquot.c, stat.c.

linux/include

- include/asm-*:
  - Architecture-dependent include subdirectories.
- include/linux:
  - Header info needed both by the kernel and user apps.
  - Usually linked to /usr/include/linux.
  - Kernel-only portions guarded by #ifdefs
    - #ifdef __KERNEL__
    - /* kernel stuff */
    - #endif
- Other directories:
  - math-emu, net, pcmcia, scsi, video.

linux/init

- Just two files: version.c, main.c.
- version.c – contains the version banner that prints at boot.
- main.c – architecture-independent boot code.
- start_kernel is the primary entry point.

linux/ipc

- System V IPC facilities.
- If disabled at compile-time, util.c exports stubs that simply return –ENOSYS.
- One file for each facility:
  - sem.c – semaphores.
  - shm.c – shared memory.
  - msg.c – message queues.

linux/kernel

- The core kernel code.
- sched.c – “the main kernel file”:
  - scheduler, wait queues, timers, alarms, task queues.
- Process control:
  - fork.c, exec.c, signal.c, exit.c etc…
- Kernel module support:
  - kmod.c, ksyms.c, module.c.
- Other operations:
  - time.c, resource.c, dma.c, softirq.c, itimer.c.
  - printk.c, info.c, panic.c, sysctl.c, sys.c.

linux/lib

- kernel code cannot call standard C library routines.
- Files:
  - brlock.c – “Big Reader” spinlocks.
  - cmdline.c – kernel command line parsing routines.
  - errno.c – global definition of errno.
  - inflate.c – “gunzip” part of gzip.c used during boot.
  - string.c – portable string code.
    - Usually replaced by optimized, architecture-dependent routines.
  - vsprintf.c – libc replacement.

linux/mm

- Paging and swapping:
  - swap.c, swapfile.c (paging devices), swap_state.c (cache).
  - vmscan.c – paging policies, kswapd.
  - page_io.c – low-level page transfer.
- Allocation and deallocation:
  - slab.c – slab allocator.
  - page_alloc.c – page-based allocator.
  - vmalloc.c – kernel virtual-memory allocator.
- Memory mapping:
  - memory.c – paging, fault-handling, page table code.
  - filemap.c – file mapping.
  - mmap.c, mremap.c, mlock.c, mprotect.c.

linux/scripts

- Scripts for:
  - Menu-based kernel configuration.
  - Kernel patching.
  - Generating kernel documentation.

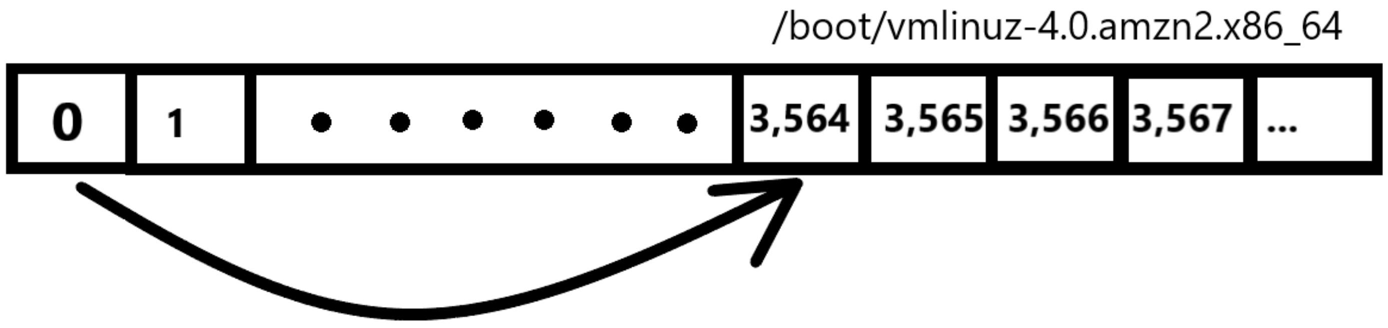
Summary

- Linux is a modular, UNIX-like monolithic kernel.
- Kernel is the heart of the OS that executes with special hardware permission (kernel mode).
- “Core kernel” provides framework, data structures, support for drivers, modules, subsystems.
- Architecture dependent source sub-trees live in /arch.

The Bootloader

The Bootloader

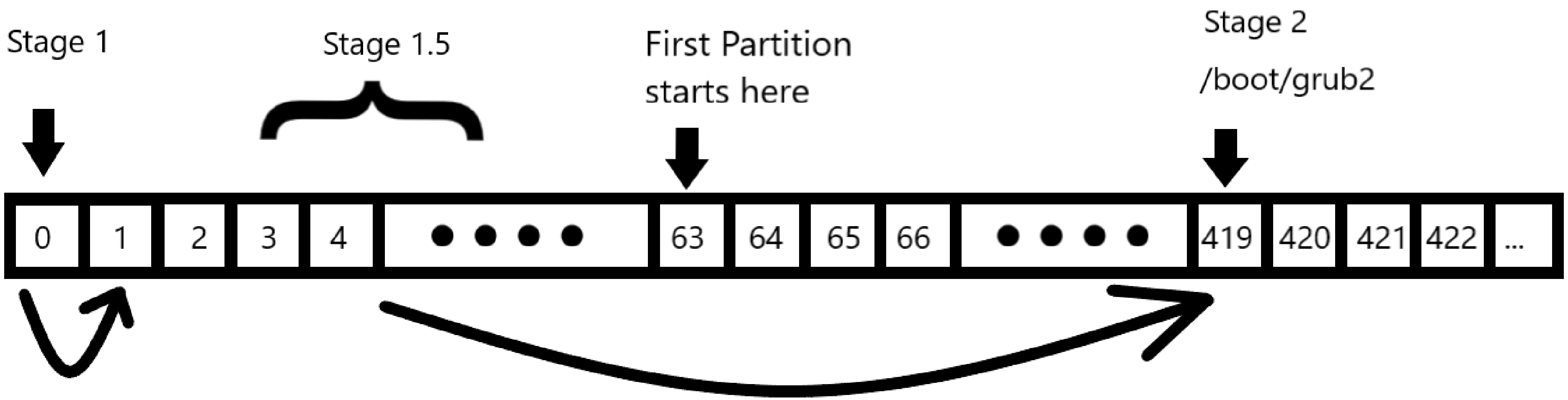
- Responsible for loading the operating system into memory.
- LILO (Linux Loader) was the default for a long time
- It works by pointing to the first sector where the kernel is stored (simplified).



- Has limited functionality due to the 446 bytes available in MBR

The Bootloader - GRUB

- GRUB replaced LILO as the default Linux bootloader (we'll focus on version 2 but version 1 is similar).
- Split into 3 stages.



The Bootloader - GRUB

- Stage 1 loads Stage 1.5
- Stage 1.5 loads the grub kernel and filesystem modules needed to locate Stage 2 (*/boot/grub2*)
- Stage 2 parses */boot/grub2/grub.cfg* and displays GRUB Menu
- When a (Linux) OS is selected, corresponding kernel image along with the appropriate initramfs images are loaded into memory.
- If Windows is selected, GRUB chainloads the Windows Bootloader.

The Chicken-and-Egg problem

The Chicken-and-Egg Problem

- The kernel needs to mount the root filesystem which can have one or more properties:
  - NFS
  - RAID
  - LVM
  - Encryption
- The necessary modules are stored in */lib/modules*
- In order to access them, the filesystem must be mounted.

The Chicken-and-Egg Problem

Solution?

Initial Ram Disk (initrd)

- A temporary filesystem that has all the stuff needed to mount the root filesystem.
- Initial Ram Disk (initrd) is an image of a block device.
- Grub makes it available as a special block device in memory (/dev/ram)
- The kernel mounts this device and uses it as it’s root filesystem.
- Kernel must be compiled with the drivers needed to mount initrd.

Initrd vs Initramfs

initrd vs initramfs

- Initrd is made available as a block device that is mounted in memory.
- Initramfs is an archive that contains the directory structure of a typical filesystem.
- Archive is extracted to created a tmpfs (temporary filesystem)
- Mounted in memory to create initramfs
- Initramfs introduced in Linux Kernel v2.6.13

The Kernel

The Kernel

- Initializes and configures memory and various hardware.
- Extract initramfs into a tempfs and mount it in memory
- Mount the root filesystem as read-only.
- Start the first program.

SysV Init

SysV-Init

- The first program started by the kernel, thus has a PID of 1.
- A daemon that runs throughout the lifetime of the system until it shuts down. Manages all other daemons and programs.
- Program is located at “/sbin/init”

The rc.sysinit script

- Located at “/etc/rc.d/rc.sysinit”, this is the first script that is executed by init and it will perform the following tasks:
  - Set the system’s hostname
  - Unmount initramfs
  - Sets kernel parameters as defined in /etc/sysctl.conf.
  - Start devfs
  - Mount procfs and sysfs
  - Dumps the current contents of the kernel ring buffer into /var/log/dmesg
  - Process /etc/fstab (mounting and running fsck)
  - Enable RAID and LVM
- After the rc.sysinit script is executed, the relevant runlevel scripts are executed.

Runlevels

- A runlevel describes the state of a system with regards to the services and functionality that is available.
- There are a total of 7 runlevels which are defined as follows:
  - 0: Halt or shutdown the system
  - 1: Single user mode
  - 2: Multi-user mode, without networking
  - 3: Full multi user mode, with networking
  - 4: Officially not defined; Unused
  - 5: Full multi user with NFS and graphics (typical for desktops)
  - 6: Reboot
- Default runlevel is configured in the config file `/etc/inittab`

Runlevel Scripts

- Runlevel scripts are responsible for starting and stopping services that will provide the functionality for the respective runlevel.
- Located under “/etc/rc.d”:

```
$ ls -l /etc/rc.d
total 60
drwxr-xr-x 2 root root  4096 Aug 26 09:19 init.d
-rwxr-xr-x 1 root root  2617 Aug 17 2017 rc
drwxr-xr-x 2 root root  4096 Aug 26 09:19 rc0.d
drwxr-xr-x 2 root root  4096 Aug 26 09:19 rc1.d
drwxr-xr-x 2 root root  4096 Aug 26 09:19 rc2.d
drwxr-xr-x 2 root root  4096 Aug 26 09:19 rc3.d
drwxr-xr-x 2 root root  4096 Aug 26 09:19 rc4.d
drwxr-xr-x 2 root root  4096 Aug 26 09:19 rc5.d
drwxr-xr-x 2 root root  4096 Aug 26 09:19 rc6.d
-rwxr-xr-x 1 root root   220 Jul 2 06:56 rc.local
-rwxr-xr-x 1 root root 20108 Aug 17  2017 rc.sysinit
```

Runlevel Scripts

- Naming convention of the scripts is as follows:
- Starts with “S” or “K”. “S” scripts are used to start the service and “K” scripts are used to stop the service.
- Followed by a number that specifies the order in which the scripts are to be executed.
- Ends with the name of the service.

rc.local sript

- After the runlevel scripts have been executed, the last script to run is the `/etc/rc.d/rc.local` script.
- You can add custom Bash commands you want to be executed at boot.
- The login prompt will show up after the `rc.local` script has been executed.

SysV-Init Important Commands

- `$ service httpd start`
- `$ service httpd status`
- `$ chkconfig httpd on`
- `$ runlevel`
- `$ telinit 3`
- `$ sed -i “s/^id:.*:initdefault:/id:3:initdefault/” /etc/inittab`

SystemD Boot Process

SystemD

- SystemD is a system and service manager that was designed to replace SysV-Init which has the following limitations:
- Services are started sequentially.
- Longer boot times.
- No easy and straightforward way to monitor running services.
- Dependencies have to be handled manually.

SystemD Units

- Every resource that is managed by SystemD is called a unit.
- A unit is a plain-text file that stores information about any one of the following:
  - a service
  - a socket
  - a device
  - a mount point,
  - an automount point
  - a swap file or partition
  - a start-up target
  - a watched file system path
  - a timer controlled and supervised by systemd
  - a resource management slice or a group of externally created processes.

SystemD Target Units

- Runlevels were replaced by “target” units,
- Target files are used to group together units that are needed for that specific target (services, sockets, devices, etc.)

SysV-Init Runlevel	**SystemD Start-up Target**
0: Halt or shutdown the system	poweroff.target
1: Single User mode	rescue.target
2: Multi-user mode, without networking	multi-user.target
3: Full multi user mode, with Networking	multi-user.target
4: Undefined	multi-user.target
5: Full multi-user mode with networking and graphical desktop.	graphical.target
6: Reboot	reboot.target

SystemD Service Units

- Runlevel Scripts were replaced by systemd unit files (mostly service unit files)
- A service unit file will define how that service is started and stopped.

```
[Unit]
Description=OpenSSH server daemon
Documentation=man:sshd(8) man:sshd_config(5)
After=network.target sshd-keygen.service
Wants=sshd-keygen.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/sshd
ExecStart=/usr/sbin/sshd -D $OPTIONS
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartSec=42s

[Install]
WantedBy=multi-user.target
```

```
[Unit]
Description=The Apache HTTP Server
Wants=httpd-init.service
After=network.target remote-fs.target nss-lookup.target httpd-init.service
Documentation=man:httpd.service(8)

[Service]
Type=notify
Environment=LANG=C

ExecStart=/usr/sbin/httpd $OPTIONS -DFOREGROUND
ExecReload=/usr/sbin/httpd $OPTIONS -k graceful
# Send SIGWINCH for graceful stop
KillSignal=SIGWINCH
KillMode=mixed
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

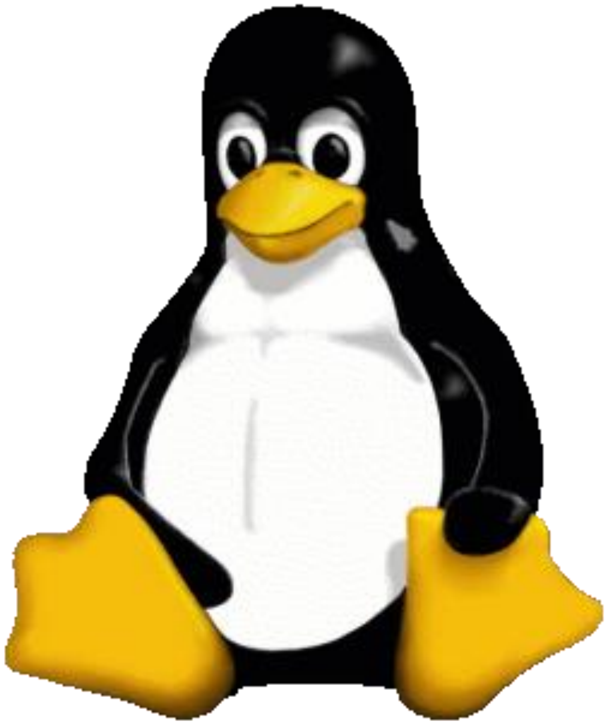
SystemD Dependencies

- There are 3 main ways of defining the dependencies of unit (target or service):
  - “Wants=” statements inside the unit files.
  - “Requires=” statements inside the unit files.
  - Special “.wants” directories associated with each target unit file found under the directory `/etc/systemd/system`
- When you enable a service using systemctl, systemd creates a symbolic link in the default target’s “.wants” directory that points to the service unit file.

SystemD Boot Process

- When a system boots into a specific target, all the dependent units are activated (services, mounts, sockets, etc.)
- All the tasks that are performed by the rc.sysinit script are replaced by “sysinit.target”.
- basic.target, multi-user.target, graphical.target and rescue.target all have sysinit.target as one of their dependencies.
- By default, rc.local script will not run by default unless if you make it executable:
  - `chmod +x /etc/rc.d/rc.local`

System Calls



System Calls

- Interface between user-level processes and hardware devices.
  - CPU, memory, disks etc.
- Make programming easier:
  - Let kernel take care of hardware-specific issues.
- Increase system security:
  - Let kernel check requested service via syscall.
- Provide portability:
  - Maintain interface but change functional implementation.

POSIX APIs

- API = Application Programmer Interface.
  - Function defn specifying how to obtain service.
  - By contrast, a system call is an explicit request to kernel made via a software interrupt.
- Standard C library (**libc**) contains wrapper routines that make system calls.
  - e.g., malloc, free are libc routines that use the brk system call.
- POSIX-compliant = having a standard set of APIs.
- Non-UNIX systems can be POSIX-compliant if they offer the required set of APIs.

Linux System Calls (1)

Invoked by executing **`int $0x80`**.

- Programmed exception vector number 128.
- CPU switches to kernel mode & executes a kernel function.

- Calling process passes **`syscall number`** identifying system call in **`eax`** register (on Intel processors).
- Syscall handler responsible for:
  - Saving registers on kernel mode stack.
  - Invoking syscall service routine.
  - Exiting by calling **`ret_from_sys_call()`**.

Linux System Calls (2)

- System call dispatch table:
  - Associates syscall number with corresponding service routine.
  - Stored in **`sys_call_table`** array having up to **`NR_syscall`** entries (usually 256 maximum).
  - nth entry contains service routine address of syscall n.

Initializing System Calls

- **`trap_init()`** called during kernel initialization sets up the IDT (interrupt descriptor table) entry corresponding to vector 128:
  - **`set_system_gate(0x80, &system_call);`**
- A system gate descriptor is placed in the IDT, identifying address of **`system_call`** routine.
  - Does not disable maskable interrupts.
  - Sets the descriptor privilege level (DPL) to 3:
    - Allows User Mode processes to invoke exception handlers (i.e. syscall routines).

The system_call() Function

- Saves syscall number & CPU registers used by exception handler on the stack, except those automatically saved by control unit.
- Checks for valid system call.
- Invokes specific service routine associated with syscall number (contained in **`eax`**):
  - **`call *sys_call_table(0, %eax, 4)`**
- Return code of system call is stored in **`eax`**.

Parameter Passing

- On the 32-bit Intel 80x86:
  - 6 registers are used to store syscall parameters.
    - **`eax`** (syscall number).
    - **`ebx`**, **`ecx`**, **`edx`**, **`esi`**, **`edi`** store parameters to syscall service routine, identified by syscall number.

Wrapper Routines

- Kernel code (e.g., kernel threads) cannot use library routines.
- **`_syscall0`** … **`_syscall5`** macros define wrapper routines for system calls with up to 5 parameters.
- e.g., **`_syscall3(int,write,int,fd,`**

```
const char *,buf,unsigned int,count)
```

Example: “Hello, world!”

```
.data                                   # section declaration

msg:
        .string "Hello, world!\n"       # our dear string
        len = . - msg                   # length of our dear string

.text                                   # section declaration

                        # we must export the entry point to the ELF linker or
    .global _start      # loader. They conventionally recognize _start as their
                        # entry point. Use ld -e foo to override the default.

_start:

# write our string to stdout

        movl    $len,%edx       # third argument: message length
        movl    $msg,%ecx       # second argument: pointer to message to write
        movl    $1,%ebx         # first argument: file handle (stdout)
        movl    $4,%eax         # system call number (sys_write)
        int     $0x80           # call kernel

# and exit

        movl    $0,%ebx         # first argument: exit code
        movl    $1,%eax         # system call number (sys_exit)
        int     $0x80           # call kernel
```

Linux Files Relating to Syscalls

- Main files:
  - arch/i386/kernel/entry.S
    - System call and low-level fault handling routines.
  - include/asm-i386/unistd.h
    - System call numbers and macros.
  - kernel/sys.c
    - System call service routines.

arch/i386/kernel/entry.S

```
.data
ENTRY(sys_call_table)
.long SYMBOL_NAME(sys_ni_syscall)  /* 0  -  old "setup()" system
                                   call*/
     .long SYMBOL_NAME(sys_exit)
     .long SYMBOL_NAME(sys_fork)
     .long SYMBOL_NAME(sys_read)
     .long SYMBOL_NAME(sys_write)
```

- Add system calls by appending entry to sys_call_table:

```
.long SYMBOL_NAME(sys_my_system_call)
```

include/asm-i386/unistd.h

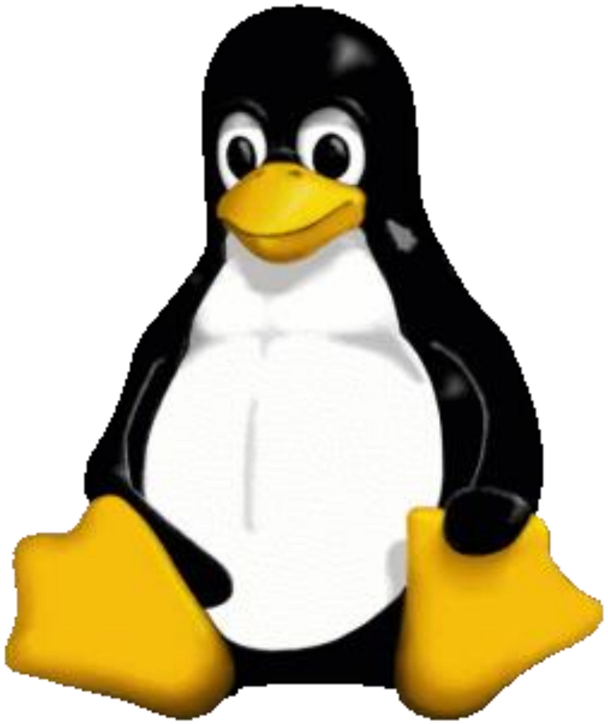
- Each system call needs a number in the system call table:
  - e.g., **`#define __NR_write 4`**
  - **`#define __NR_my_system_call nnn`**, where **`nnn`** is next free entry in system call table.

kernel/sys.c

- Service routine bodies are defined here:
- e.g.,

```
asmlinkage retval
sys_my_system_call (parameters) {
     body of service routine;
     return retval;
}
```

Kernel Modules



Kernel Modules

- Modules can be compiled and dynamically linked into kernel address space.
  - Useful for device drivers that need not always be resident until needed.
    - Keeps core kernel “footprint” small.
  - Can be used to “extend” functionality of kernel too!

Example: “Hello, world!”

```
#define MODULE
#include <linux/module.h>
int init_module(void) {
 printk(“<1>Hello, world!\n”);
 return 0;
}
void cleanup_module(void) {
 printk(“<1>Goodbye cruel world ☹\n”);
}
```

Using Modules

- Module object file is installed in running kernel using **`insmod module_name`**.
  - Loads module into kernel address space and links unresolved symbols in module to symbol table of running kernel.

The Kernel Symbol Table

- Symbols accessible to kernel-loadable modules appear in **`/proc/ksyms`**.
  - **`register_symtab`** registers a symbol table in the kernel's main table.
  - Real hackers export symbols from the kernel by modifying **`kernel/ksyms.c`** ☺