Maemo Diablo The Linux System Model
Training Material

February 9, 2009

Contents

Chapter 1

The Linux System Model

Linux is a free, multi-threading, multiuser operating system that has been ported to several different platforms and processor architectures. This chapter gives an overview of the common System Model of Linux, which is also a base for the maemo platform. The concepts described in this chapter include kernel, processes, memory, filesystem, libraries and linking.

1.1 The Kernel

The kernel is the very heart of a Linux system. It controls the resources, memory, schedules processes and their access to CPU and is responsible of communication between software and hardware components. The kernel provides the lowest-level abstraction layer for the resources like memory, CPU and I/O devices. Applications that want to perform any function with these resources communicate with the kernel using `system calls`. `System calls` are generic functions (such as `write`) and they will handle the actual work with different devices, process management and memory management. The advantage in `system calls` is that the actual call stays the same regardless of the device or resource being used. Porting the software for different versions of the operating system also becomes easier when the `system calls` are persistent between versions.

Kernel memory protection divides the virtual memory into `kernel space` and `user space`. `Kernel space` is reserved for the kernel, its' extensions and the device drivers. `User space` is the area in memory where all the user mode applications work. User mode application can access hardware devices, virtual memory, file management and other kernel services running in `kernel space` only by using `system calls`. There are over 100 `system calls` in Linux, documentation for those can be found from the Linux kernel system calls manual pages (`man 2 syscalls`).

Kernel architecture, where the kernel is run in `kernel space` in supervisor mode and provides `system calls` to implement the operating system services is called `monolithic` kernel. Most modern monolithic kernels, such as Linux, provides a way to dynamically load and unload executable kernel modules at runtime. The modules allow extending the kernel's capabilities (for example adding a device driver) as required without rebooting or rebuilding the whole

kernel image. In contrast, microkernel is an architecture where device drivers and other code are loaded and executed on demand and are not necessarily always in memory.

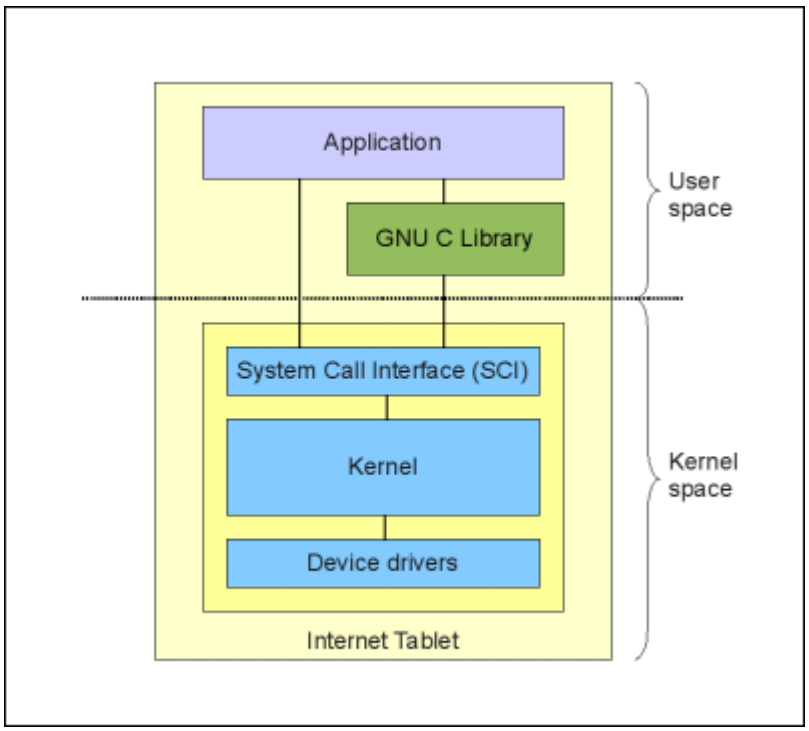

Figure 1.1: Kernel, hardware and software relations

1.2 What are processes

A process is a program that is being executed. It consists of the executable program code, a bunch of resources (for example open files), an address space, internal data for kernel, possibly threads (one or more) and a data section.

Each process in Linux has a series of characteristics associated with it, below is a (far from complete) list of available data:

PID Process ID, numeric identifier of the process

State State of the process (running, stopped etc.)

PPID Parent process ID

Children

Open files List of open files for the process

TTY The terminal to which the process is connected

RUID Real user ID, the owner of the process

Every process has a PID, process ID. This is a unique identification number used to refer to the process and a way to the system to tell processes apart from each other. When user starts the program, the process itself and all processes started by that process will be owned by that user (process RUID), thus processes' permissions to access files and system resources are determined by using permissions for that user.

1.3 Creating a process

To understand the creation of a process in Linux, we must first introduce few necessary subjects, `fork`, `exec`, `parent` and `child`. A process can create an exact clone of itself, this is called `forking`. After the process has forked itself the new process that has born gets a new PID and becomes a child of the forking process, and the forking process becomes a parent to a child. But, we wanted to create a new process, not just a copy of the parent, right? This is where `exec` comes into action, by issuing an `exec` call to the system the child process can overwrite the address space of itself with the new process data (executable), which will do the trick for us.

This is the only way to create new processes in Linux and every running process on the system has been created exactly the same way, even the first process, called `init` (PID 1) is forked during the boot procedure (this is called `bootstrapping`).

If the parent process dies before the child process does (and parent process does not explicitly handle the killing of the child), `init` will also become a parent of `orphaned` child process, so its' PPID will be set to 1 (PID of `init`), see Figure 1.2.

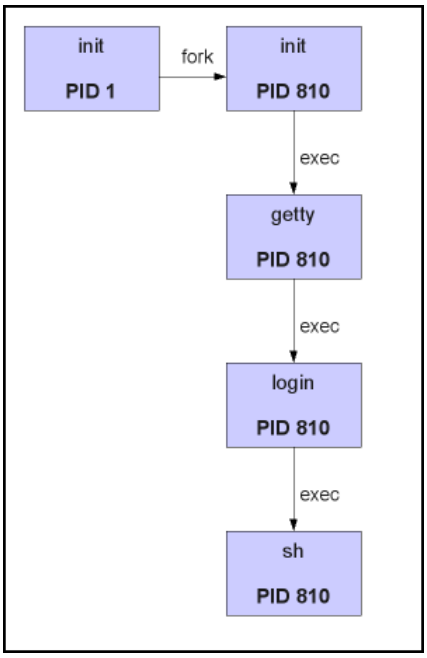

Figure 1.2: Example of the fork-and-exec mechanism

Parent-child relations between processes can be visualized as a hierarchical tree:

```
init-+-gconfd-2
     |-avahi-daemon---avahi-daemon
     |-2*[dbus-daemon]
     |-firefox---run-mozilla.sh---firefox-bin---8*[{firefox-bin}]
     |-udevd

...listing cut for brewity...
```

The tree structure above also states difference between `programs` and `processes`. All processes are executable programs, but one program can start multiple processes. From the tree structure you can see that running `Firefox` web browser has created 8 child processes for various tasks, still being one program. In this case, the preferred word to use from Firefox would be an application.

1.4 Ending a process

Whenever a process terminates normally (program finishes without intervention from outside), the program returns numeric `exit status` (return code) to the parent process. The value of the return code is program-specific, there is no standard for it. Usually `exit status 0`, means that process terminated normally (no error). Processes can also be ended by sending them a `signal`. In Linux there are over 60 different signals to send to processes, most commonly used are listed in Table 1.1.

SIGNAL	Signal number	Explanation
SIGTERM	15	Terminate the process nicely.
SIGINT	2	Interrupt the process. Can be ignored by process.
SIGKILL	9	Interrupt the process. Can NOT be ignored by process.
SIGHUP	1	Used by daemon-processes, usually to inform daemon to re-read the configuration.

Table 1.1: Most commonly used process signals in Linux

Only the user owning the process (or `root`) can send these signals to the process. Parent process can handle the `exit status` code of the terminating child process, but is not required to do so, in which case the `exit status` will be lost.

1.5 Filesystem hierarchy

Linux follows the UNIX-like operating systems concept called unified hierarchical namespace. All devices and filesystem partitions (they can be local or even accessible over the network) apper to exist in a single hierarchy. In filesystem namespace all resources can be referenced from the **root** directory, indicated by a forward slash (**/**), and every file or device existing on the system is located under it somewhere. You can access multiple filesystems and resources within the same namespace: you just tell the operating system the location in the filesystem namespace where you want the specific resource to appear. This action is called `mounting`, and the namespace location where you attach the filesystem or resource is called a `mount point`.

The mounting mechanism allows establishing a coherent namespace where different resources can be overlaid nicely and transparently. In contrast, the filesystem namespace found in Microsoft operating systems is split into parts and each physical storage is presented as a separate entity, e.g. **C:** is the first hard drive, **E:** might be the CD-ROM device.

Example of mounting: Let us assume we have a memory card (MMC) containing three directories, named **first**, **second** and **third**. We want the contents of the MMC card to appear under directory **/media/mmc2**. Let us also assume that our `device file` of the MMC card is **/dev/mmcblk0p1**. We issue the `mount` command and tell where in the filesystem namespace we would like to `mount` the MMC card.

```
/ $ sudo mount /dev/mmcblk0p1 /media/mmc2
/ $ ls -l /media/mmc2
total 0
drwxr-xr-x 2 user group 1 2007-11-19 04:17 first
drwxr-xr-x 2 user group 1 2007-11-19 04:17 second
drwxr-xr-x 2 user group 1 2007-11-19 04:17 third
/ $
```

In addition to physical devices (local or networked), Linux supports several `pseudo` filesystems (virtual filesystems) that behave like normal filesystems,

but do not represent actual persistent data, but rather provide access to system information, configuration and devices. By using pseudo filesystems, operating system can provide more resources and services as part of the filesystem namespace. There is a saying that nicely describes the advantages: "In UNIX, everything is a file".

Examples of `pseudo` filesystems in Linux and their contents:

procfs Process filesystem, used to access process information from the kernel. Usually mounted under **/proc**

devfs Used for presenting `device files`, e.g. devices as files. An abstraction for accessing I/O an peripherals. Usually mounted under **/dev**

sysfs Information about devices and drivers from the kernel, also used for configuration. Usually mounted under **/sys**

Using pseudo filesystems provides a nice way to access kernel data and several devices from userspace processes using same API and functions than with regular files.

Most of the Linux distributions (as well as maemo platform) also follow the Filesystem Hierarchy Standard (FHS) quite well. FHS is a standard which consists of a set of requirements and guidelines for file and directory placement under UNIX-like operating systems 1.3.

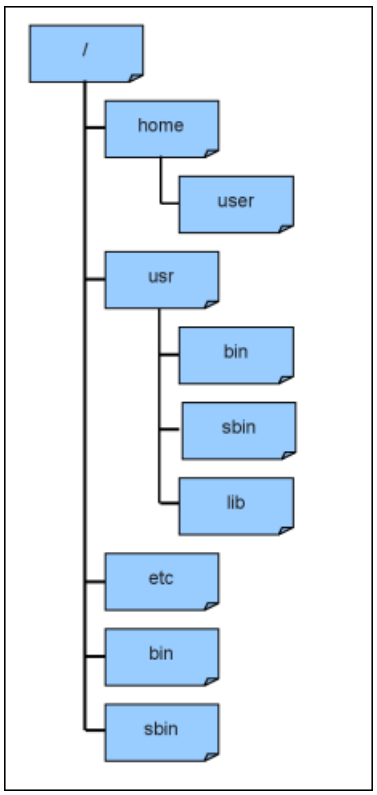

Figure 1.3: Example of filesystem hierarchy

The most important directories and their contents:

/bin Essential user command binaries that need to be available also in `single user mode`.

/sbin Essential system binaries (e.g. `init`, `insmod`, `ifup`)

/lib Libraries for the binaries in **/bin** and **/sbin**

/usr/bin Non-essential user command binaries that are not needed in `single user mode`

/usr/sbin Non-essential system binaries (e.g. daemons for network-services)

/usr/lib Libraries for the binaries in **/usr/bin** and **/usr/sbin**

/etc Host-specific system-wide configuration files

/dev Device files

/home User home directories (optional)

/proc Virtual file system documenting kernel and process status as text files

More information about FHS can be found from its' homepage at pathname.com/fhs/.

1.6 Files and inodes

For most users understanding the tree-like structure of the filesystem namespace is enough. In reality, things get more compilated than that. When a physical storage device is taken into use for the first time, it must be partitioned. Every partition has its own filesystem, which must be initialized before first usage. By mounting the initialized filesystems, we can form the tree-structure of the entire system.

When a filesystem is created to partition, a data structures containing information about files are written into it. These structures are called `inodes`. Each file in filesystem has an `inode`, identified by inode serial number.

Inode contains the following information of the file:

Owner Owner of the file

Group Group owner of the file

Permissions File permissions, more on these to follow

Last read time Time when file was last accessed (atime)

Last change time Time when file was last modified (mtime)

Inode change time Time when inode itself was last modified (ctime)

Hard links Number of hard links to this file

Size The length of the file in bytes

Address Pointer to the actual content data of the file

The only information not included in an inode is the file name and directory. These are stored in the special directory files, each of which contains one filename and one inode number. The kernel can search a directory, looking for a particular filename, and by using the inode number the actual content of the inode can be found, thus allowing the content of the file to be found.

Inode information can be queried from the file using `stat` command:

```
user@system:~$ stat /etc/passwd
  File: '/etc/passwd'
  Size: 1347            Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d      Inode: 209619      Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2007-11-23 06:30:49.199768116 +0200
Modify: 2007-11-17 21:27:02.271803959 +0200
Change: 2007-11-17 21:27:02.771832454 +0200
```

Notice that the `stat` command shown above is from desktop Linux, as maemo platform does not have `stat` -command by default.

Some advantages of `inodes`:

- Multiple filenames can be associated with the same `inode`. This is called `hard linking` or just linking. Notice that `hard links` only work within one partition as the inode numbers are only unique within a given partition.
- As processes open files, the kernel converts the filename to an inode number at the first possible chance, thus moving the file (within same partition) does not prevent a process from accessing the file once it is opened (as long as the process knows the inode number).

In addition to hard links, Linux filesystem also supports `soft links`, or more commonly called, `symbolic links` (or for short: `symlinks`). A symbolic link contains the path to the target file instead of a physical location on the hard disk. Since `symlinks` do not use inodes, `symlinks` can span across the partitions. Symlinks are transparent to processes which read or write to files pointed by a symlink, process behaves as if operating directly on the target file.

1.7 File access permissions

The Linux security model is based on the UNIX security model. Every user on the system has an user ID (`UID`) and every user belongs to one or more groups (identified by group ID, `GID`). Every file (or directory) is owned by a `user` and a `group user`. There is also a third category of users, `others`, those are the users that are not the owners of the file and do not belong to the group owning the file.

For each of the three user categories, there are three permissions that can either be granted or denied:

Read (r) Whether the file may be read. In the case of a directory, the ability to list the contents of the directory.

Write (w) Whether the file may be written to or modified (includes deleting and renaming).

Execute (x) Whether the file may be executed. In the case of a directory, the ability to enter to or execute program from that directory.

The file permissions can be checked simply by issuing a `ls -l` command:

```
/ $ ls -l /bin/ls
-rwxr-xr-x 1 root root 78004 2007-09-29 15:51 /bin/ls
/ $ ls -l /tmp/test.sh
-rwxrw-r-- 1 user users 67 2007-11-19 07:13 /tmp/test.sh
```

The first 10 characters in output describe the file type and `permission flags` for all three user categories:

- First character tells the file type (- means regular file, `d` means directory, `l` means symlink)
- Characters 2-4 display the access rights for the `actual owner` of the file (- means denied)
- Characters 5-7 display the access rights for the `group owner` of the file (- means denied
- Character 8-10 display the access rights for `other users` (- means denied)

The output also lists the `owner` and the `group owner` of the file, in this order.

Let us look closer what this all means. For the first file, **/bin/ls**, the owner is root and group owner is also root. First three characters (rwx) indicate that `owner` (root) has read, write and execute permissions to the file. Next three characters (r-x) indicate that the `group owner` has read and execute permissions. Last three characters (r-x) indicate that all other users have read and execute permissions.

The second file, **/tmp/test.sh**, is owned by `user` and group owned by users belonging to `users` group. For `user` (owner) the permissions (rwx) are read, write and execute. For users in `users` group the permissions (rw-) are read and write. For all other users the permissions (r–) are only read.

File permissions can also be presented as octal values, where every user category is simply identified with one octal number. Octal values of the permission flags for one category are just added together using following table:

```
                  Octal value       Binary presentation
Read access             4                  100
Write access            2                  010
Execute access          1                  001

Example: converting "rwxr-xr--" to octal:

First group  "rwx" = 4 + 2 + 1 = 7
Second group "r-x" = 4 + 1     = 5
Third group  "r--" = 4         = 4

So "rwxr-xr--" becomes 754
```

As processes run effectively with permissions of the user who started the process, the process can only access the same files as the user.

Root-account is a special case: Root user of the system can override any permission flags of the files and folders, so it is very adviseable to **not** run unneeded processes or programs as `root` user. Using `root` account for anything else than system administration is not recommended.

1.8 Programs, daemons and libraries

As we earlier stated, processes are programs executing in the system. Processes are, however, also a bit more than that: They also include set of resources such as open files, pending signals, internal kernel data, processor state, an address space, one or more threads of execution, and a data section containing global variables. Processes, in effect, are the **result** of running program code. A program itself is not a process. A process is an active program and related resources.

Threads are objects of activity inside the process. Each thread contains a program counter, process stack and processor registers unique to that thread. Threads require less overhead than forking a new process because the system does not initialize a new system virtual memory space and environment for the process. Threads are most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing.

Daemons are processes that run in the background unobtrusively, without any direct control from a user (by disassociating the daemon process from the controlling TTY). Daemons can perform various scheduled and non-scheduled tasks and often they serve the function of responding to the requests from other computers over a network, other programs or hardware activity. Daemons have usually `init` as their `parent process`, as daemons are launched by forking a child and letting the parent process die, in which case `init` adopts the process. Some examples of the daemons are: httpd (web server), lpd (printing service), cron (command scheduler).

`Linking` refers to combining a program and its libraries into a single executable and resolving the symbolic names of variables and functions into their resulting addresses. Libraries are a collection of commonly used functions combined into a package.

There are few types of linking:

Static linking Static linking copies a set of routines to the executable target application, thus expanding the target executable size.

Dynamic linking Dynamic linking means that the subroutines of a library are loaded into an application program at runtime. Dynamic libraries remain on filesystem as separate files from executable. Most of the work of linking is done at the time the application is loaded, thus creating a bit of overhead in application startup time.

Run-time linking Run-time linking (or dynamic loading) is a subset of dynamic linking where a dynamically linked library loads or unloads at run-time on request.

In addition to being loaded statically or dynamically, libraries can also be `shared`. Dynamic libraries are almost always shared, static libraries can not be shared at all. Sharing allows same library to be used by multiple programs at the same time, sharing the code in memory.

Dynamic linking of shared libraries provides multiple benefits:

- Common code of the applications is shared at runtime, which reduces the total memory usage.

- Application executable file size is reduced, which reduced the total storage usage.
- Fixing a bug from shared code fixes the bug from all the applications using the same code.

Most Linux systems use almost entirely dynamic libraries and dynamic linking.

1.9 Decomposition of a simple command-line program

Below is an image which shows the decomposition of a very simple command-line-program in Linux, dynamically linking only to `glibc` (and possibly `Glib`). As Glib is so commonly used in addition to standard C library (`glibc`), those libraries have been drawn to one box, altough they are completely separate libraries. The hardware devices are handled by the kernel and the program only accesses them through the system call (`syscall`) interface.

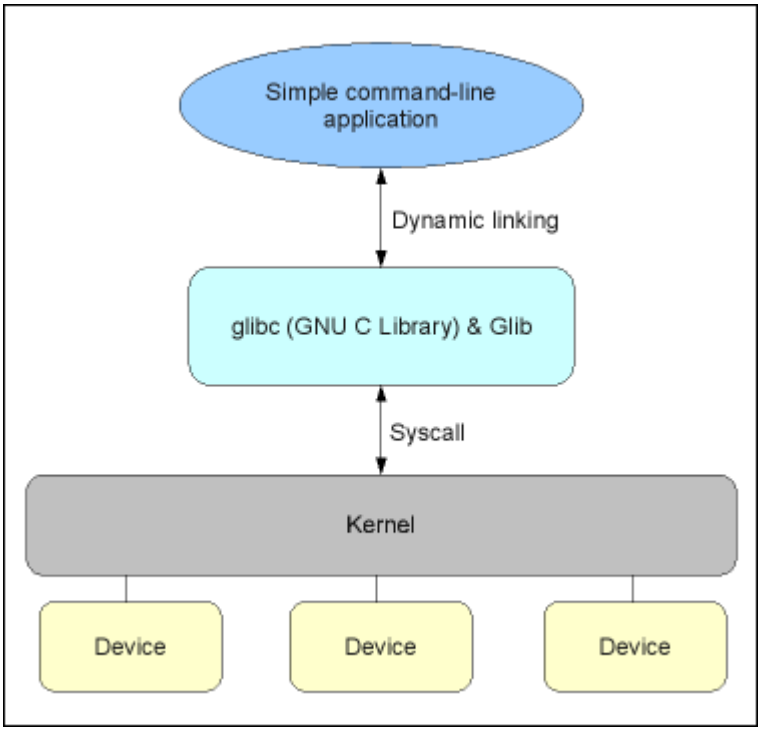

Figure 1.4: Decomposition of a simple command-line program