8 - Linux (not given as a separate lecture during 2014)

TSEA81 - Computer Engineering and Real-time Systems

This document is released - 2013-12-16 - first version (new homepage)

Author - Ola Dahl

Lecture - 8 - Linux (not given as a separate lecture during 2014)

This Lecture gives an overview of Linux. Note that for 2014, this lecture was not given. The most important parts of the lecture (e.g. information about pthreads) were given during many lectures. Its main sources of inspiration has been the books Understanding the Linux Kernel[1], which is also available at the LiU Library[2], and Linux Kernel Development (3rd Edition)[3].

For additional information regarding the history and the open source aspects of Linux, see e.g. Just For Fun[4] by Linus Torvalds, or Rebel Code: Linux And The Open Source Revolution[5] by Glyn Moody.

Additional references are found inside the text.

[1] http://www.amazon.com/Understanding-Linux-Kernel-Third-Daniel/dp/0596005652

[2] http://www.bibl.liu.se/?l=en

[3] http://www.amazon.com/Linux-Kernel-Development-Robert-Love/dp/0672329468/

[4] http://www.bokus.com/bok/9789150100235/just-for-fun-mannen-bakom-linux/

[5] http://www.amazon.com/Rebel-Code-Linux-Source-Revolution/dp/0738206709

Background

According to Wikipedia[6], Linux is a Unix-like operating system. It is named after its inventor, Linus Torvalds[7], who created the Linux kernel[8].

[6] http://en.wikipedia.org/wiki/Linux

[7] http://en.wikipedia.org/wiki/Linus_Torvalds

[8] http://en.wikipedia.org/wiki/Linux_kernel

The word Linux can also mean an *operating system distribution*, containing an operating system, file system(s), many applications, tools, and software for graphics and communication. Examples are *Ubuntu* and Ångström[9].

[9] http://www.angstrom-distribution.org/

Linux is used in personal computers and in servers, but also in embedded systems, then often referred to as Embedded Linux[10], e.g. in communication systems and in industrial systems.

[10] http://elinux.org/Main_Page

The Linux Kernel is licensed using a GNU[11] license.

[11] http://www.gnu.org/

The Linux kernel is a *monolithic* operating system kernel. It allows *loadable kernel modules*, it provides preemptive multitasking of processes (and threads), and it has memory management for virtual memory with multiple address spaces.

Linux was first announced on August 26, 1991, by Linus Torvalds, in a newsgroup message[12] with the text

[12] https://groups.google.com/forum/?fromgroups=#!msg/comp.os.minix/dlNtH7RRrGA/SwRavCzVE7gJ

- *"Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready"*

In 1994, Linux 1.0 was released[13]. The current stable version of Linux[14] is 3.6.9.

The source code of the Linux kernel can be downloaded from The Linux Kernel Archives[15].

The Linux kernel can be used together with GNU software, as is done when creating Linux distributions. The resulting system may be called a *GNU/Linux operating system*. This is in contrast to a pure GNU operating system[16], which instead would have used the GNU Kernel, which is called Hurd[17].

The Linux Kernel is implemented mostly in C[18].

[13] http://en.wikipedia.org/wiki/History_of_Linux
[14] http://kernel.org/
[15] http://kernel.org/
[16] http://www.gnu.org/
[17] http://www.gnu.org/software/hurd/hurd.html
[18] http://en.wikipedia.org/wiki/C_%28programming_language%29

Usage and programming

The Linux file system structure[19] is hierarchical. It is organized in a standardized way[20]. Starting from the root level, which has the directory name /, one often finds a recognizable set of directories, such as

- */bin* - containing programs implementing commands, to be executed by system administrators and users. Example commands are *ls*, *pwd*, and *cat*.
- */sbin* - with programs implementing system-oriented commands, e.g. *insmod*
- */boot* - with boot loader files, e.g. files related a boot loader called GRUB[21].
- */etc* - with configuration files, e.g. configuration files for *X11*, and the file */etc/passwd*, with information required during login.
- */home* - with user directories
- */edu* - with student directories
- */usr/bin* with programs implementing commands. Example commands could be *ssh* and *which*.
- */usr/include* - with standard include files, e.g. the file *stdio.h*
- */var* - files that change during run-time

[19] http://www.tldp.org/LDP/intro-linux/html/sect_03_01.html
[20] http://www.pathname.com/fhs/pub/fhs-2.3.html
[21] http://www.gnu.org/software/grub/

Linux supports programming in a variety of languages, such as C, Python, Perl, C++, and Java.

When developing software using Linux it may be of interest to use features that are commonly found in Unix-related operating systems. It is e.g. possible to create programs which utilize features for *process programming*, such as process communication and process synchronization, using mechanisms like *sockets*, *pipes*, or *shared memory*.

Information about programming in Linux can e.g. be found in the book Advanced Linux Programming[22].

[22] http://www.advancedlinuxprogramming.com/

Information about usage of Linux, including information about commands, shell scripting, and system administration tasks, can be found in an on-line book called LINUX: Rute User's Tutorial and Exposition[23].

[23] http://rute.2038bug.com/index.html.gz

A program in Linux can request a service from the Linux kernel. This is done using a system call[24].

[24] http://en.wikipedia.org/wiki/System_call

A system call changes the processor mode, from *user mode* to a *privileged* mode. This mode is often referred to as *kernel mode*.

A system call can be implemented using a special processor instruction (e.g. *software interrupt*).

A library is a set of routines used by a program. A program, executing in user mode, may call a library function, e.g. *printf*. The function printf may issue a system call, e.g. *write()*, which then constitutes the entry point to the operating system.

The Linux man pages[25] are organized in different sections. There are 8 sections. General commands are given in section 1, system calls are given in section 2, and library functions are given in section 3. As an example, one might try, in a Linux shell, to give the commands *man printf* and *man 3 printf*, or the commands *man write* and *man 2 write*.

[25] http://linux.die.net/man/

A program executing in user mode uses addresses assigned to it. These addresses are referreed to as *user space*. The corresponding addresses when executing in kernel mode are referred to as *kernel space*.

A *process* is an instance of a *program* in execution. Processes in Linux have a parent-child-relationship.

Linux provides *multiple address spaces*. This means that, in most cases, each process has its own *address space*. As a consequence, one process cannot directly refer to an address used in another process, e.g. by using a pointer, and there can be no variables which are shared by two processes. By the use of virtual memory this means that one virtual address, e.g. *0x1000*, correspond to *different physical addresses* when used in different processes.

The word *thread* is used to denote a process which shares its address space with another process. When two threads sharing a common address space use a specific virtual address, these references correspond to *the same physical address*. Threads can therefore share variables, and they can access each other's memory using pointers.

Tasks in a real-time operating system, such as *Simple_OS* or *FreeRTOS*, often have a common address space.

In Linux, the word *task* is used to refer to an executing entity, which can be either a process with its own address space, or a thread

sharing its address space with one or more other threads.

The Linux scheduler[26] schedules tasks. Each task is described by a data structure[27]. When a task is created, using the clone system call[28], it is decided if it shall share address space with other tasks or not. It is also decided if it shall share other resources.

A device driver[29] is a piece of software responsible for creating an interface between a hardware unit and a user program. A device driver performs communication with the hardware as well as with a user program. Linux device drivers often execute in kernel mode (using addresses in kernel space), and their interfaces to user programs can sometimes be implemented using system calls such as *read*, *write*, and *ioctl*.

Linux device drivers can contain *interrupt handlers*, e.g. an interrupt handler for a keyboard[30] can be implemented as part of a device driver.

A comprehensive treatment of device drivers in Linux is given in the book Linux Device Drivers, Third Edition[31].

[26] http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler

[27] http://lxr.linux.no/linux+v3.6.6/include/linux/sched.h#L1234

[28] http://linux.die.net/man/2/clone

[29] http://www.linuxforu.com/tag/linux-device-drivers-series/page/2/

[30] http://www.ts.mah.se/utbild/da135a/laborationer/lab6/lab6_da135a.html

[31] http://lwn.net/Kernel/LDD3/

Processes and threads

The Linux processes are identified using a process identity. The first process created, during startup, has the number 0. This is the *idle process*, which is a process internal to the kernel (referred to as a kernel thread). The process with number 1 is a user space program, referred to as the *init* program. All other processes are descendants of this process.

As mentioned above, processes, or more correctly - tasks, can be created using the clone[32] system call. A process can also be created using the fork[33] system call.

A process can be terminated using the _exit[34] system call, which may be invoked from the exit[35] C library function.

A Linux process switch[36] involves, as is the case for an RTOS, saving and restoring of hardware context. Registers are saved on the *kernel mode stack* of each process, and in its *process descriptor*.

A process descriptor (the task_struct[37] struct in the kernel), includes e.g. process state, process id (pid), reference to the kernel mode stack, and much more. Process descriptors can be stored in *linked lists*.

A process switch also involves *memory management*, since address spaces need to be changed - page tables for the new process need to replace page tables for the old process.

Linux processes are *preemptible*. This means that a processes can be suspended, not only voluntarily as is the case when performing a system call, but also *involuntarily*, e.g. as a consequence of an inter-

[32] http://linux.die.net/man/2/clone

[33] http://linux.die.net/man/2/fork

[34] http://linux.die.net/man/2/exit

[35] http://linux.die.net/man/3/exit

[36] http://www.makelinux.net/books/ulk3/understandlk-CHP-3-SECT-3

[37] http://lxr.linux.no/linux+v3.6.6/include/linux/sched.h#L1234

rupt, or when its time quantum has expired.

Also the Linux kernel is preemptible[38]. This means that a process switch can take place during execution inside the kernel, e.g. during execution of a system call.

Linux processes are scheduled according to a scheduling policy, defined by a scheduling class[39]. There are two *real-time scheduling classes* - called *SCHED_FIFO* (similar to priority-based RTOS-scheduling) and *SHED_RR* which is *SCHED_FIFO* with time-slicing.

There is one normal scheduling class, called *SHED_NORMAL* which is the time sharing CFS method[40].

The real-time scheduling classes use static priorities, assigned in a dedicated real-time priority range. The static priority and the scheduling policy can be modified using system calls, e.g. nice[41] for changing the static priority, and sched_setscheduler[42] for changing scheduling policy.

[38] http://www.linuxjournal.com/article/5600

[39] http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/#domains

[40] http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler

[41] http://linux.die.net/man/2/nice

[42] http://linux.die.net/man/2/sched_setscheduler

Real-time Linux

There are variants of Linux which are adapted for better real-time properties. Sometimes these variants are referred to as different ways of obtaining a Real-time Linux[43].

The Linux 2.6 kernel has a *CONFIG_PREEMPT* configuration option which allows process to be preempted even if they are executing a system call.

The Linux 2.6 series has the *O(1) scheduler*, which can perform scheduling in constant time, and the *CFS* scheduler, which gives improved responsiveness for interactive tasks.

In addition, there is the CONFIG_PREEMPT_RT[44] patch, which allows nearly all of the kernel to be preempted.

Another method for adding real-time capabilities to Linux can be described as a *thin-kernel approach*, where Linux is executed as a low-priority task in a thin kernel which runs directly on the hardware (as an RTOS). Some examples of this approach are RTLinux, RTAI[45], and Xenomai[46].

For additional information about real-time aspects of Linux, see e.g. this article about real-time in Linux[47].

[43] http://www.realtimelinuxfoundation.org/events/rtlws-2012/ws.html

[44] https://rt.wiki.kernel.org/

[45] https://www.rtai.org/

[46] http://www.xenomai.org/

[47] http://www.ibm.com/developerworks/library/l-real-time-linux/index.html

Programming with Pthreads

POSIX threads, also referred to as *Pthreads*, are available in Linux, and also in other flavors of UNIX. Information about programming with Pthreads can be found e.g. in a tutorial from Lawrence Livermore National Laboratory[48], and in a tutorial from YoLinux[49].

Pthreads are used in the course in *Lab 2 - Embedded Linux*. Below

[48] https://computing.llnl.gov/tutorials/pthreads/

[49] http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

follows a description on how pthreads can be used, with application in *Lab 2*.

Compilation and linking

A program using pthreads should use an include directive of the form

```
/* pthread include */
#include <pthread.h>
```

It should be linked using the linker switch *-lpthread*. In *Lab 2*, it should be compiled using the compiler switch *-DPTHREADS*.

Thread definition

A thread is defined by a *thread handle*. An example, for one thread, is given by

```
/* handle for lift task */
pthread_t Lift_Handle;
```

Thread handles can also be defined for an indexed set of threads, as

```
/* handles for passenger tasks */
pthread_t Passenger_Handle[MAX_N_PERSONS];
```

A function to be used as thread is defined as

```
/* lift_thread: moves the lift */
void *lift_thread(void *thread_param)
```

A parameter value can be transferred to a thread when a thread is created. In the thread code, the parameter value can be received by declaring and assigning a variable, as

```
    /* passenger id */
    int id;

    /* receive id */
    int *id_ref = (int *) thread_param;
    id = *id_ref;
```

Thread creation

Threads can be created, e.g. from the *main* function of a program, as

```
    /* create lift thread */
    pthread_create(&Lift_Handle, NULL, lift_thread, NULL);
    /* create user thread */
    pthread_create(&User_Handle, NULL, user_thread, NULL);
```

A thread can also be created from another thread. This is practised in *Lab 2*, where threads for the lift passengers can be created as

```
/* create passenger thread */
pthread_create(&Passenger_Handle[id], NULL, &passenger_thread,
               (void *) &Passenger_Ids[Passenger_Id_Index]);
```

Wait for a specified amount of time

A thread can be forced to wait a specified amount of time. This can be accomplished using the function *sleep* or the function *usleep*, which are available after including *unistd.h* as

```
/* unistd is needed for usleep and sleep */
#include <unistd.h>
```

A specified waiting time can then be requested, e.g. after a lift travel is finished, using code such as

```
/* make the journey */
lift_travel(Lift, id, from_floor, to_floor);

/* sleep for a while */
usleep(5000000);
```

Mutual Exclusion

Mutual exclusion can be implemented using binary semaphores, also referred to as *mutexes*. A mutex can be declared as

```
/* mutex for mutual exclusion */
pthread_mutex_t mutex;
```

A mutex can be initialised as

```
/* initialise mutex */
pthread_mutex_init(&lift->mutex, NULL);
```

A resource can then be reserved, using a *Wait*-operation, as

```
/* reserve lift */
pthread_mutex_lock(&lift->mutex);
```

and released, using a *Signal*-operation as

```
/* release lift */
pthread_mutex_unlock(&lift->mutex);
```

Condition variables

A condition variable can be declared as

```
/* condition variable, to indicate that something has happend */
pthread_cond_t change;
```

It can be initialised as

```
/* initialise condition variable */
pthread_cond_init(&lift->change, NULL);
```

An *Await*-operation can be performed as

```
    pthread_cond_wait(&lift->change, &lift->mutex);
```

A *Cause*-operation can be done, as

```
/* indicate to other tasks that the lift has arrived */
pthread_cond_broadcast(&lift->change);
```