Exam
Operating Systems - OS

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

February, 15th, 2011

Authorized documents: Nothing! The exam takes into account that you don't have the right to have any document with you.
A grade is provided for every question. 1 additional point is given as a general appreciation, including written skills and readability.

1 Booting Linux (10 points, ∼70 minutes)

This exercise is based on an excerpt of a paper - slightly adapted - published by *M. Tim Jones*, Consultant Engineer, Emulex, and published by IBM in May 2006.
It is available at *http://www.ibm.com/developerworks/linux/library/l-linuxboot/index.html*.
Questions are first based on the overall understanding of the paper, and then, on more precise elements of the boot process raised by the paper. Apart from question 1, **answers shall be very short, only a few lines long**.

Inside the Linux boot process

The kernel image isn't so much an executable kernel, but a compressed kernel image. Typically this is a zImage (compressed image, less than 512KB) or a bzImage (big compressed image, greater than 512KB), that has been previously compressed with zlib. At the head of this kernel image is a routine that does some minimal amount of hardware setup and then decompresses the kernel contained within the kernel image and places it into high memory. If an initial RAM disk image is present, this routine moves it into memory and notes it for later use. The routine then calls the kernel and the kernel boot begins.

When the bzImage (for an i386 image) is invoked, you begin at ./arch/i386/boot/head.S in the start assembly routine. This routine does some basic hardware setup and invokes the startup_32 routine in ./arch/i386/boot/compressed/head.S. This routine sets up a basic environment (stack, etc.) and clears the Block Started by Symbol (BSS). The kernel is then decompressed through a call to a C function called decompress_kernel (located in ./arch/i386/boot/compressed/misc.c). When the kernel is decompressed into memory, it is called. This is yet another startup_32 function, but this function is in ./arch/i386/kernel/head.S.

In the new startup_32 function (also called the swapper or process 0), the page tables are initialized and memory paging is enabled. The type of CPU is detected along with any optional floating-point unit (FPU) and stored away for later use. The start_kernel function is then invoked (init/main.c), which takes you to the non-architecture specific Linux kernel. This is, in essence, the main function for the Linux kernel.

With the call to start_kernel, a long list of initialization functions are called to set up interrupts, perform further memory configuration, and load the initial RAM disk. In the end, a call is made to kernel_thread (in arch/i386/kernel/process.c) to start the init function, which is the first user-space process. Finally, the idle task is started and the scheduler can now take control (after the call to cpu_idle). With interrupts enabled, the pre-emptive scheduler periodically takes control to provide multitasking.

During the boot of the kernel, the initial-RAM disk (initrd) that was loaded into memory by the precious stage of the boot loading process is copied into RAM and mounted. This initrd serves as a temporary root file system in RAM and allows the kernel to fully boot without having to mount any physical disks. Since the necessary modules needed to interface with peripherals can be part of the initrd, the kernel can be very small, but still support a large number of possible hardware configurations. After the kernel is booted, the root file system is pivoted (via pivot_root) where the initrd root file system is unmounted and the real root file system is mounted.

The initrd function allows you to create a small Linux kernel with drivers compiled as loadable modules. These loadable modules give the kernel the means to access disks and the file systems on those disks, as well as drivers for other hardware assets. Because the root file system is a file system on a disk, the initrd function provides a means of bootstrapping to gain access to the disk and mount the real root file system. In an embedded target without a hard disk, the initrd can be the final root file system, or the final root file system can be mounted via the Network File System (NFS).

After the kernel is booted and initialized, the kernel starts the first user-space application. This is the first program invoked that is compiled with the standard C library. Prior to this point in the process, no standard C applications have been executed.

---

1. Provide a summary of this article, in 200 words, with a +/- 10% margin. [2 points]

2. "*This routine sets up a basic environment (stack, etc.)*" → What do you think the author had in mind with "etc."? [1 point]

3. "*With interrupts enabled, the pre-emptive scheduler periodically takes control to provide multitasking*".

   (a) Why are interrupts necessary for the scheduler to take the control on the system? [1 point]

   (b) More generally, why are interrupts configured during the boot process? [1 point]

   (c) What exactly is configured regarding those interrupts during the boot process? [1 point]

4. RAM-disk

   (a) Why does the kernel try to avoid using hard disks during the boot process, i.e., why does it rely on a RAM disk? [1 point]

   (b) What is the purpose of the *pivot_root* function? [1 point]

5. User and system processes

   (a) What is the difference between a user-level process and a system-level process? [1 point]

   (b) Is it always useful for (Real-Time) Operating Systems to have both? [1 point]

2 Programming threads (9 points, ∼50 minutes)

Threads have now become an important element of recent Operating Systems. We've spent several lectures and lab sessions on threads. During one of them, we started using processes communicating using shared memory.

1. Basically explain the most important steps before two **processes** (so, not threads) can effectively communicate using shared memory. [3 points]

2. Is the use of shared memory objects useful when communicating between two threads? Give the sketch - in an approximate C code - of two threads exchanging a string value using the technique of your choice: the first one puts a value in a string, the second one reads it, and so on. I do know that you don't have any document with you, and so, I don't expect the syntax of your code to be correct. [6 points]