Linux Internals

Dominic Duval

Student Guide

Revision 1.3

Object Innovations Course 310

Linux Internals

Student Guide

Object Innovations, Inc.
420 Boston Turnpike
Shrewsbury, MA 01545
781-272-3860
www.ObjectInnovations.com

Printed in the United States of America.

Table of Contents

Chapter 4

Inter-Process Communication

Inter-process Communication

Objectives

After completing this unit you will be able to:

- **Understand why User mode processes need to synchronize themselves and exchange data.**
- **Identify the main mechanisms that UNIX makes available for inter-process communication.**
- **Identify the three elements of the Sys V IPC communication standard.**
- **Make decisions regarding the type of interprocess communication that would best fit in a particular software design.**

Interprocess Communications

- **Interprocess communication refers to the exchange of data between two different processes running on a single system. The primary interprocess communication mechanisms that will be discussed in this chapter are the following:**
  - Pipes
  - FIFOs
  - IPC communications
  - Sys V Socket based communications
- **Note that UNIX signals are often used for interprocess communication, but they will not be described in the current chapter since this topic was discussed in Chapter 2.**

Communication via Files

- **The two simplest process communication mechanisms are arguably pipes and FIFOs.**
- **Both of these mechanisms involve file objects in the Kernel. They are handled in a first-in, first-out order (FIFO).**
- **Since the data flow is handled sequentially, no file positioning is allowed with pipes and FIFOs.**
- **As we will see, the major difference between pipes and FIFOs is that communication channels related to pipes are anonymous, whereas channels associated with a FIFO have a name.**
- **The next few pages will focus on those two communication mechanisms so that you can identify where to use pipes instead of FIFOs, and vice versa.**

Pipes

- **Pipes are known as the oldest communication mechanism under UNIX. They can be used on all UNIX operating systems, so there are usually very few portability problems involved with this method.**
- **Pipes only provide half-duplex communication, so information flows only one way. Another disadvantage is that they can only be used by processes that have a common ancestor.**
- **In UNIX/Linux, a pipe is used when we execute the following command:**

```
ls | more
```

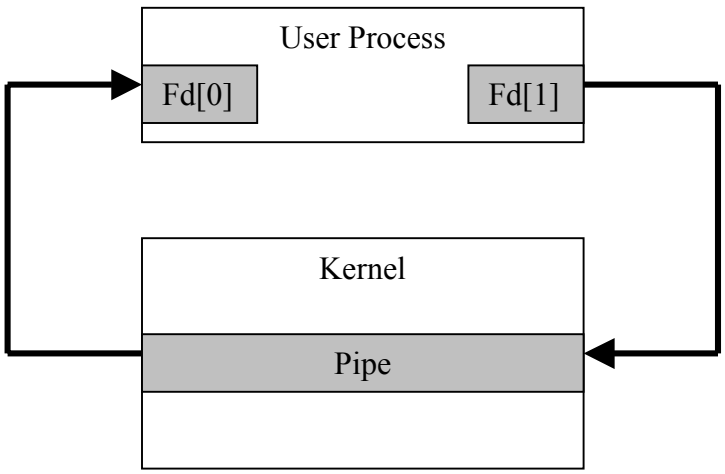
- **In the previous example, the two executed processes are linked together by a pipe. The standard output of the first process is sent to the pipe, which sends the data to the input of the second process. Other shell commands involving the "<,>,>>" characters also work in a similar fashion.**
- **In a shell, pipes are made possible since the shell is the common ancestor of the two processes. The shell is forked the first time for the "ls" command and a second time for the "more" command.**

Programming With Pipes

- **A pipe is created by calling the pipe() function in the following way:**

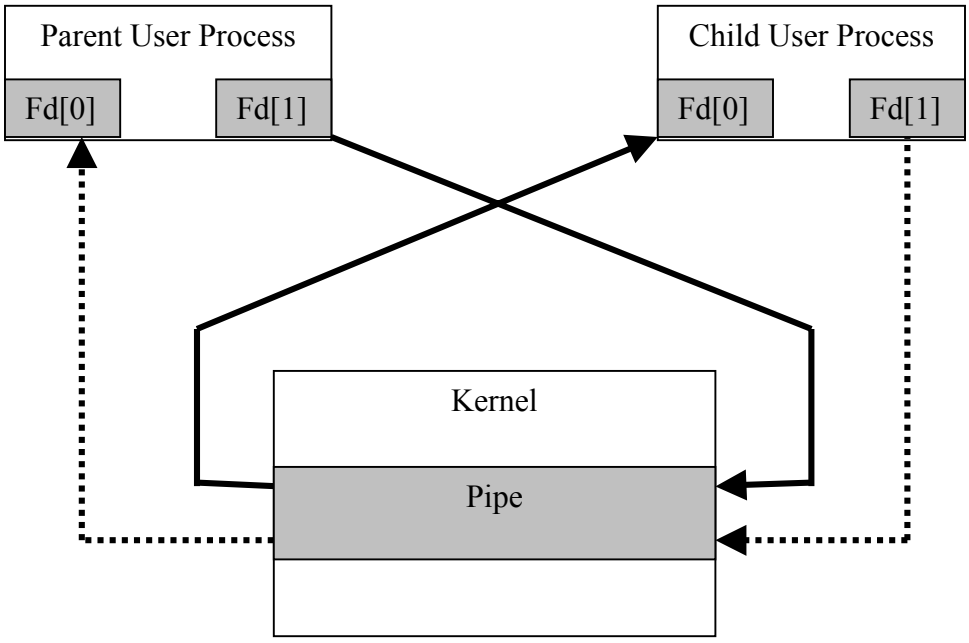
```
Int fd[2];
if (pipe(fd) < 0)
    printf("Error!\n");
```

- **Using a pipe involves the following aspects:**
  - An array for two file descriptors must first be created.
  - The **pipe**() function is invoked. This returns two valid file descriptors in the array given as the argument. The input of the first file descriptor (fd[0]) is the output of the second file descriptor (fd[1]).
  - Test the return value of the pipe() function. A negative value indicates an error.
  - Close the file descriptors before the end of the process.



Programming With Pipes

- **A pipe like the one that we have just created in the previous example is useless. In order to do be significantly usable, a fork() must be invoked after the pipe() function has been called. This allows two processes to communicate together.**
- **After a fork, the pipe would look like the following:**



- **The pipe is really made possible by the Kernel. It can be viewed as a mechanism existing inside the Kernel and providing access points (file descriptors).**
- **The resulting file descriptors may be read and written as we usually do for a normal file descriptor.**

Pipe Data Structures

- **Since the Kernel is the element making pipes possible, some Kernel data structures are associated with this type of operation.**
  - Since the **read**() and **write**() functions have access to the file descriptors returned by the pipe() function, the Linux Kernel must create an **inode** object associated with the pipe.
  - The inode data structure has a field named **i_pipe**, which is a pointer to a data structure **pipe_inode_info**. Needless to say, this structure is only usable in a context where a pipe is used. It contains the address of a page frame acting as a buffer containing data written to the pipe, a lock flag, a wait queue, and flags used by processes reading from or writing to the pipe.
  - Two **file** objects (one for the read access and the other for the write access) must be created as well. Each file object points to the same inode object.
- **Note that the i_size field in the inode object contains the number of bytes currently stored in the pipe buffer. This is also known as the pipe size.**
- **The Kernel does not allow two accesses to the pipe buffer at the same time.**
  - This measure is taken in order to avoid race conditions.
  - Pipe operations must also be atomic (POSIX standard).
  - Write operation cannot be executed when the buffer is full.

Pipe Kernel Implementation

- **A pipe is implemented in the Linux Kernel as a Virtual File System object (we will see in the next chapter what is a VFS object). A VFS object representing a pipe does not have any data associated to it on the physical disk.**
- **The pipe() function that we have seen previously is a libc stub for the system call associated with the sys_pipe() Kernel function.**
  - We can take a look at the implementation of the sys_pipe() function in arch/i386/kernel/sys_i386.c.
- **The sys_pipe() function calls the do_pipe() Kernel function:**
  - **do_pipe**() is the function that allocated the file objects and file descriptors for the read and write channel of the pipe.
  - It calls the **get_pipe_inode**() function in order to associate an inode object to the current pipe. This also associates a page frame for pipe buffering, and it stores its address in the base field of the **pipe_inode_info** structure.
  - A **dentry** object is allocated in order to tie together the two file objects and the inode object.
- **If no errors occurred during the sys_pipe() function, the two file descriptors are returned to the User mode process. Notice that this is accomplished with the copy_to_user() Kernel function.**

FIFOs

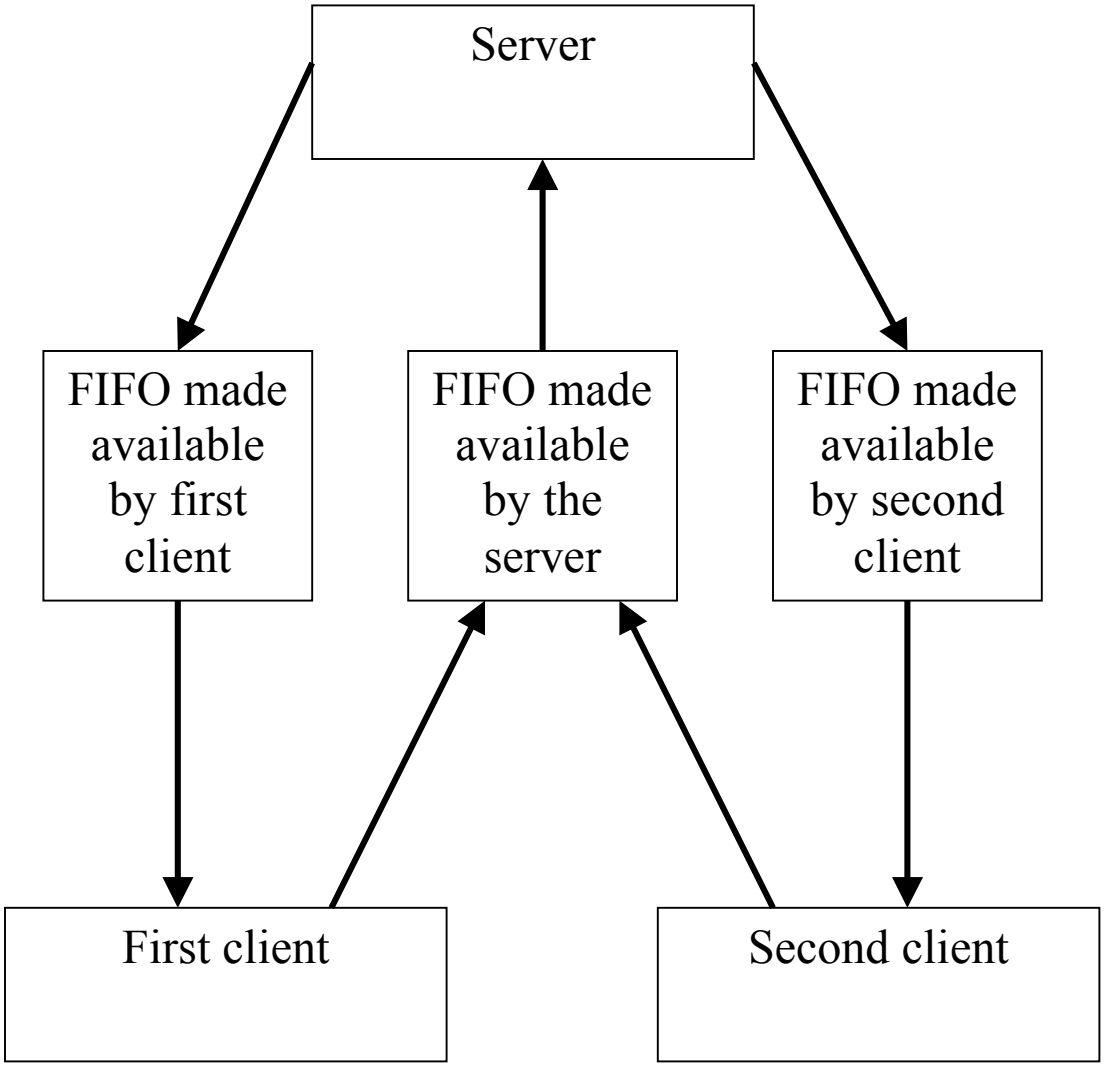
- **Pipes have the drawback of being only usable with two processes that come from a common ancestor. There is no way two unrelated processes might communicate together by using a pipe.**
- **This is a major issue for applications designed under the client-server model. In this case, a client application that has just been started by the user would not be able to connect to the server, unless the server was started at the same time, from the same parent process.**
- **FIFOs are the answer to this problem. They operate with the following aspects:**
  - FIFOs are a special file type under UNIX operating systems.
  - They have a disk **inode** but they don't take space on disks.
  - Any process can access them, since it appears on the directory tree as a file with a specific filename.
  - Just like pipes, data written to a FIFO is stored in a buffer located in memory. This fast temporary storage is therefore much more efficient than storing data on temporary disk files.
  - Client-server designs are much more easily implemented with this mechanism. A client can use the input FIFO of the server in order to issue requests, and the server writes the resulting data in the input FIFO of each client.

Programming With FIFOs

- **From the perspective of User space programs, FIFOs are usually created with the mkfifo() function. This function, in conformity with the POSIX standard, invokes the mknod() system call in order to communicate with the Kernel.**
- **The arguments of this system call are the path and name of the file to create or use as the FIFO, along with the value S_IFIFO and the file permissions in the flag field.**
- **After it is created, the FIFO can be accessed with the same functions as pipes, namely the open(), read(), write() and close() calls.**
  - The VFS Kernel subsystem determines if these functions are called in the context of a FIFO.
  - The VFS adapts file operations so that they can make sense on a FIFO. Otherwise, the file would be treated as a normal file located on the filesystem, and errors would result. When the FIFO is opened, the **i_op** field (which describes the possible operations on the current file) is set to the address of the **fifo_inode_operations** table.
  - A process may open a FIFO for reading, writing, or both at the same time.
- **FIFOs use the same data structures as pipes.**

Programming With FIFOs

- **FIFOs may be used under a client-server model as in the following:**



File Operations on Pipes and FIFOs

- **The file and inode objects contain a field called f_op and i_op, respectively, which determines which file operations can be executed on the associated object.**
- **For pipes, this field f_op is modified (at the file object level) so that it now points to new operations:**
  - The **f_op** field of the read channel is replaced with the address of the **read_pipe_fops** table.
  - The **f_op** field of the write channel is replaced with the address of the **write_pipe_fops** table.
- **For FIFOs, the associated file operations are modified at the inode object level:**
  - The **i_op** field of the inode object now points to the **fifo_inode_operations** table.
  - File objects associated with FIFOs are initialized with the file operations defined in the inode **i_op** field.

Questions

1. What restriction is imposed on pipes concerning the number and the type of processes that can use them?
2. Why do we need an array of two file descriptors when we create a pipe with the pipe() function?
3. What function does the program usually call once the pipes are created? What happens if we don't call this function?
4. What other name do we often give to FIFOs?
5. What is the main advantage of FIFOs over pipes?
6. Multiple processes may simultaneously write to a given FIFO. What problem could then result?

System V IPC

- **IPC stands for "Inter Process Communications". This standard, which officially appeared for the first time in UNIX System V release (in 1983), is commonly used by UNIX operating systems due to the fact that it increases portability from one UNIX to another.**
- **IPC contains three different communication mechanisms:**
  - **Messages**: sends or receives messages to or from any process.
  - **Semaphores**: allows unrelated processes to synchronize their execution.
  - **Shared memory**: allows unrelated processes to share a memory area.
- **There are many similarities between these three IPC mechanisms. We will focus on these similar aspects before focusing on each IPC resource.**

IPC Identifier and Key

- **Each IPC structure has an identifier associated to it:**
  - Each IPC structure (message queue, semaphore or shared memory area) is referred to in the Kernel by an IPC **identifier**. This identifier is a large positive number stored as a 32-bit integer. This number is continually incremented as IPC resources are created and erased.
  - This identifier is all what we need in order to use the resource (message, semaphore or shared memory). For example, for sending or retrieving a message, all we need to know is the identifier of the message and the queue containing the message.
  - The Identifier needs to be a truly unique number representing an IPC structure in the Kernel. This number is assigned by the Kernel and cannot be repeated twice. In fact, the Kernel almost never recycles old identifiers.
  - Processes that want to communicate through an IPC structure refer to each other with the IPC identifier of the structure.
- **Each IPC structure also has a 32-bit bit integer that is called the key.**
  - The programmer specifies this **key** in order to differentiate resources of the same type launched from a single process.
  - A key is not necessarily unique within the system, since the programmer specifies them. If several instances of the same program are running on the system, chances are that the same key number is used in each independent process.

IPC Structure Creation

- **IPC structures are generated by calling the function associated with the structure we want to create:**
  - The **semget**() function is used to create a semaphore structure.
  - The **msgget**() function is used to create a message queue.
  - The **shmget**() function is used to create a shared memory segment.
- **These three "get" functions all have two similar arguments: a key and a flag.**
  - The key is of type **key_t** and is defined by the programmer. The key will be associated with an identifier by the Kernel. If IPC_PRIVATE is defined as the key, a new IPC structure will be created (instead of referencing an existing one).
  - The **flag** parameter is an integer in which we usually place predefined values. IPC_CREAT means that the IPC resource must be created. IPC_EXCL indicates that the "get" function must return an error if the structure already exists and the IPC_CREAT is also set.

IPC Permission Structure

- **Each IPC structure (message queue, semaphore or shared memory area) has a permission structure called ipc_perm associated with it. This structure contains the following information:**
  - The owner's **group** and **user ID**
  - The creator's **group** and **user ID**
  - Access **modes** (permission bit mask)
  - Slot usage sequence number
  - Key number
- **All fields (except seq) in the permission data structure are initialized when the IPC structure is created.**
- **The creator of the IPC structure or the superuser may modify the user ID, group ID and access mode fields by calling msgctl(), semctl() or shmctl() (for the message queue, semaphore or shared memory area respectively). These functions also allow us to retrieve some information about an IPC structure.**
- **Note that the ipc_perm structure is defined as ipc_perm in /usr/include/bits/ipc.h (libc includes) and as ipc64_perm in the include/asm-i386/ipcbuf.h (Kernel sources).**

IPC System Call

- **When programs running in User space call a function like msgget(), the program actually calls a libc library function (a wrapper) that invokes the ipc system call.**
- **Due to historical design decisions, each IPC function in User space is converted to the ipc system call. This was considered a better decision than implementing separate system calls for each one of the IPC functions.**
- **The sys_ipc Kernel function (the implementation of the ipc system call) determines which function called the system call and invokes the corresponding Kernel function.**
- **In fact, some implementations of Linux on other architectures (the Alpha for example) provide separate system calls for each single IPC function called by User space programs.**
- **The implementation of sys_ipc can be found in arch/i386/kernel/sys_i386.h. The comments included in the code should give you an idea of how ugly this system call is perceived by Linux Kernel developers.**

IPC: Messages

- **Processes can send and receive messages by using IPC messages. A message is sent by a process to an IPC message queue, where it can be retrieved by another process for reading.**
- **The msgsnd() function is used to send a message. Its parameters include:**
  - The IPC identifier of the message queue where the message should be sent
  - The size of the message in bytes
  - The address of a message buffer in User space that contains the message type and the message (in clear text) itself.
- **The msgrcv() function does the corresponding operation of receiving a message which is waiting in the message queue, and it must include the following parameters:**
  - The IPC identifier of the message queue from which the message is received.
  - The address of a buffer where the message will be stored.
  - The size of the buffer
  - A value t describing which message should be retrieved. If this value is 0, the first message in the queue will be selected. If it is a positive number, the first message that contains a matching type value will be selected.

IPC: Messages (Cont'd)

- **A message resource is described by the msqid_ds data structure, defined in /include/asm-i386/msgbuf.h. This structure contains the following information:**
  - Permissions
  - Last time a message was received and sent
  - Last time the structure was changed
  - Number of bytes and messages in the queue
  - Maximum number of bytes in the queue
  - PID of the last process which received and sent a message
- **The message queue is described by the msg_queue data structure. It carries the following information:**
  - The message queue contains the same information as the msqid_ds data structure.
  - It also contains pointers to the actual messages; q_messages is a doubly linked list of messages (each node contains a next and prev pointer that refers to a msg_msg structure).
- **The messages are contained in msg_msg (one data structure for each message). It contains:**
  - Pointers to next and previous msg_msg structures in the list
  - Message type and size
  - The actual message

IPC: Semaphores

- **Semaphores are not similar to the conventional interprocess communication mechanisms that we are used to see. A semaphore is basically a counter that provides access for two or more processes to a shared object.**
- **The value of the semaphore determines if a process can access the data or not:**
  - The semaphore value is positive if the data is available. In this case, the process that needs access to the data has to decrement the value of the semaphore before accessing the data.
  - Its value is negative or 0 if the data is already being access by another process, and therefore unavailable. In this case, the process has to wait until it becomes positive.
  - When a process that was using the data stops using it, it increments the value of the semaphore so that the other processes that were waiting for it could wake up and take control of the data.
- **The process of testing the semaphore value must be atomic, so they are implemented in the Kernel (implementing it in User space could not make it absolutely atomic).**

IPC: Semaphores (Cont'd)

- **In order to access a shared object controlled by a semaphore, a process has to go through the following steps:**
  - Call the **semget**() function with the IPC key of the semaphore and the number of necessary semaphores as the arguments. The process will then be able to acquire the semaphore identifier. If the semaphore does not already exist, the IPC_CREATE flag must be set, so that a new structure is created.
  - The **semop**() function is invoked to test and decrement the semaphore specified in the arguments.
  - The process accesses the shared resource.
  - The function **semop**() is invoked for the second time, now to increment the semaphore involved in the operation and let other processes access it if necessary.
  - The function **semctl**() can optionally be called in order to remove a semaphore from the Kernel.

IPC: semid_ds and sem

- **The semget() function can create, with the flag argument properly set, a new semaphore. The created structure is in fact an array of counters (called primitive semaphores, their number is specified as a semget argument) inside the semaphore. The semaphore array is described by the semid_ds data structure, which contains the following fields:**
  - Permissions
  - Last semop() time
  - Last time the semaphore was modified
  - Number of semaphores in the array
- **Each primitive semaphore is described internally by the sem data structure, which contains:**
  - The current value of the primitive semaphore
  - The PID of the last process which accessed the semaphore

IPC: Shared Memory

- **The IPC shared memory is arguably the most popular IPC mechanism. It allows processes to access some common data structures by making them available in a shared memory segment.**
  - This is often the fastest method for exchanging data between processes. Since there is no need to copy the information between the processes, the Kernel has less work to do and can operate more quickly.
  - The most important issue raised with shared memory is synchronization. The shared memory area must be managed so that a process cannot write this area when another one reads it. Semaphore can therefore be used to protect those memory areas.
- **Definitions and data structures associated with shared memory can be found in include/linux/shm.h.**
- **The shmget() function must be invoked to obtain a shared memory identifier. The following arguments are needed:**
  - The key field identifies the memory region in the program.
  - The size value is the minimum length of the shared segment. For an existing segment we can take the value 0.
  - The flags IPC_CREATE and/or IPC_EXCL, along with the access mode bits.
  - The shared memory ID is returned if there was no error.

IPC: shmat Function

- **After a shared memory region has been created with the shmget() function, the process needs to attach it to its own address space by invoking the shmat() function.**
  - The ID number that was returned from shmget() is the first argument of shmat().
  - The address in the process memory space at which the newly shared memory region is attached must be defined by the second argument. This is usually equal to 0, since this value lets the Kernel decide where to place the new memory segment. Other values may bring errors if the program is ported to other architectures.
  - The third argument is the flag field. We could, for example, specify the SHM_RDONLY so that the process only gets read-only access to the memory region.
  - The returned value of shmat() is the address where the shared memory region is shared.
- **The shmdt() function is called in order to detach the memory region from the current process. A single argument is needed in this case, which is the memory address of the shared memory region (this address was returned by the first call to shmat).**

IPC: Shared Memory Data Structures

- **Note that the mechanism used to keep track of the shared memory areas in the Kernel is quite different in the Kernel 2.4.x if we compare with the older versions, so don't expect to find the same mechanisms in 2.2.x.**
- **A shmid_kernel data structure describes the memory area that is shared among two or more processes. This memory is calculated in terms of shared page frames.**
- **A set of pages is linked to this descriptor. This is the actual shared memory region.**
- **A linked list of vm_area_struct data structures represents the memory areas associated with the shared memory for each process. If two processes are sharing the same memory region, this list will contain two vm_area_struct elements.**
  - The implementation of this data structure can be found in include/linux/mm.h.

Sockets

- **Processes may communicate together by using socket-based mechanisms (also known as BSD sockets). There are two main types of socket addressing formats:**
  - **AF_INET** are sockets that use Internet addresses (four bytes numbers written as four decimal numbers separated by periods) as their addressing format. Combined with the use of port numbers, this addressing scheme allows more than one AF_INET socket on a given machine and more than one machine to communicate.
  - **AF_UNIX** are sockets that may only be used on the current local machine. They use UNIX pathnames as the addressing scheme and are very useful for communication between processes running on the same machine.
- **This section will focus on AF_UNIX sockets, also called UNIX domain sockets. AF_INET sockets will be discussed in Chapter 10, which deals with networking internals.**
- **The implementation of the AF_UNIX can be found in net/unix/af_unix.c.**
- **BSD sockets may be either anonymous (which is the case when they are created by the socketpair() function) or linked with a socket entry in the filesystem, in which case the socket would be subject to the associated filesystem permission checks.**

Sockets (Cont'd)

- **The socket() function returns a socket descriptor, which acts similarly to file descriptors returned by the open() function.**
  - Each descriptor number represents an open file, a socket, or neither. A descriptor number never represents both a socket and a file.
  - Many system calls (such as the close function) accept both socket and file descriptors.
- **The socket() function needs the address format as its first argument. All subsequent operations on the returned socket will be interpreted accordingly to the specified address format (in our case AF_UNIX).**
- **A socket does not initially have a name (or address) associated with it. Since two processes need to access the same socket, they must be able to recognize it:**
  - The bind() function is used bind the socket descriptor to an address.
  - A socket does not have a name (thus, it cannot be used by any other process) until an address is assigned to it with the bind() system call.
  - The bind() function takes the socket descriptor, the address (of type sockaddr) and the length of the address in bytes are the argument of the bind() function.
- **Socket internals will be discussed in Chapter 10.**

Questions

7. What are the two elements common to every IPC mechanism?
8. What is the advantage of grouping all IPC-related functions that are called from User space in a single system call named sys_ipc?
9. How can a client and a server agree on a common IPC structure in order to communicate together?
10. Why are IPC and UNIX sockets considered reliable compared to network sockets?
11. What is the fastest form of IPC, and why?
12. What are race conditions?

Summary

- **Pipes and FIFOs are easily used for interprocess communication in UNIX programs and can usually be ported to other UNIX flavors without too much effort.**
- **The IPC standard allows three communication mechanisms: messages, semaphores and shared memory areas.**
- **Using the IPC standard is advantageous because all three mechanisms are used in a similar fashion and can be quite easily ported across other UNIX flavors.**
- **Socket-based interprocess communication can be more complex to use, but interprocess local communications may be easily converted to interprocess network communications for an application using a client-server model.**

Chapter 12

Boot Process

Boot Process

Objectives

After completing this unit you will be able to:

- **Explain what happens when a PC is turned on and boots the Linux Kernel.**
- **Identify what functionality is expected from a boot loader such as LILO.**
- **Identify where every component of the Linux Kernel is initialized.**
- **Configure, install and troubleshoot LILO, the Linux Boot Loader.**

Booting Process – Introduction

- **The booting process is architecture specific. Booting a Sun SPARC machine, for example, will involve a totally different process from booting an Intel machine.**
- **The booting process for a PC under Linux can be separated in five different steps:**
  - System startup, during which the boot device and boot sector are selected by the BIOS.
  - The boot sector loads the setup code, decompression routines and compressed kernel images.
  - The kernel is decompressed in protected mode.
  - Low-level initialization is performed.
  - High-level C initialization of the Kernel is completed.
- **We will describe each one of these steps in the following pages.**

Step One : System Startup

- **In this stage the system's hardware devices, including RAM, are in a random, inconsistent state.**
  - The RESET pin on the CPU is asserted.
  - Some registers on the processor are set to fixed values; ds, es, fs, gs and ss are all set to 0.
  - The code at the physical memory address 0xfffffff0 is executed. This memory location corresponds to a read-only memory chip (ROM) that contains a suite of programs traditionally called BIOS (Basic Input/Output System). This checks the system devices in order to make sure they work properly and initializes the interrupt vector at physical address 0.
  - The BIOS invokes its bootstrap loader function, which copies the first sector of the bootable device (usually the floppy drive or the hard disk, as determined by the BIOS settings) at physical address 0x00007C00, then executes the code at this address.
  - The BIOS also provides several functions for handling low-level devices' input and output. Some operating systems, such as MS-DOS, make fairly extensive use of these functions. However, the Linux Kernel provides its own mechanisms for handling these low-level tasks. This is due to the fact that the Kernel executes its operations in protected mode, whereas the BIOS works in real mode. It is thus impossible to share the BIOS functions with the Linux Kernel.

Step Two : Boot Loader

- **At this point, the BIOS is executing the boot loader located at physical address 0x00007C00.**
  - The boot loader is a program invoked by the BIOS to load the image of the operating system Kernel into RAM.
- **For a PC system booting Linux, the boot loader may be either:**
  - The Linux boot sector, which is coded in arch/i386/boot/bootsect.S and is automatically included with any Kernel image.
  - LILO (Linux Loader), which is a program allowing the user to select which operating system will be launched on the system, in cases there are more than one.
  - No boot loader is also a possible option. In this case, a program such as LOADLIN may be used.
- **The boot process at this stage is heavily dependent on the media being used to boot Linux.**
  - Booting from a floppy disk will only involve copying the first sector to RAM and executing it.
  - Booting from a hard disk will involve copying the Master Boot Record (MBR), which includes the partition table and a program that loads the first sector of the partition containing the OS (which is the *active* partition if MS-DOS is used). In this case, Linux replaces this program by another one (like LILO) that is more versatile than the original one supplied with MS-DOS or other similar operating systems.

Linux Boot Sector

- **The first option that is offered in order to load Linux involves using the Linux boot sector, and is often used on bootable floppy disks.**
  - The Linux boot sector is always included in Kernel images.
  - When a new Kernel is compiled, the assembly code located in the arch/i386/boot/bootsect.S is automatically copied at the beginning of the Kernel image file.
  - Thus, the first few kilobytes of code in the Linux Kernel image are not the Kernel itself, but a totally different program that is used to load the Kernel.
- **Typically, we can make a floppy boot disk by using the following command:**

```
dd if=bzImage of=/dev/fd0
```

- **This command will dump the Kernel image on the floppy disk.**
  - The beginning of the Kernel image will correspond to the first sector of the floppy disk.
  - When the BIOS loads the first sector of the floppy disk, the beginning of the Kernel (the code of the Linux boot sector) is actually the part that is loaded into RAM.

Linux Boot Sector (Cont'd)

- **The Linux boot sector goes through the following steps when it is invoked by the BIOS:**
  - It first moves itself from physical address 0x00007C00 (where it was copied by the BIOS) to physical address 0x00090000. These addresses are defined by the BOOTSEG and INITSEG variables in bootsect.S.
  - The real mode stack is initialized at address 0x00003FF4.
  - The disk parameter table is set up. This is used by the BIOS in order to use the floppy device driver.
  - A BIOS function displaying the "Loading…" message is invoked.
  - A BIOS function is executed that will load the setup() function of the Kernel image in RAM, precisely at address 0x00090200 (defined by DEF_SETUPSEG). Note that the setup function is not executed at this point.
  - Another BIOS procedure is invoked to load the whole Kernel image in RAM at physical address 0x00010000 if the Kernel was compiled with "make zImage" or at physical address 0x00100000 if it was built with "make bzImage". These two options are often referred to as the Kernel being loaded low or high.
  - The setup() function is now executed.

Linux Loader (LILO)

- **Linux is generally loaded from a hard disk, where the most commonly used boot loader is LILO.**
- **LILO may be located either on the Master Boot Record or in the boot sector of a disk partition.**
  - If it is placed on the MBR, it will replace the small program that was originally placed at this location.
  - If it is placed on the boot sector of a disk partition, this partition must usually be made active so that the boot loader of the MBR can properly switch to it.
- **LILO is divided into two components. Otherwise, making it fit into the MBR would be impossible due to the relatively large size of this program.**
  - The MBR or partition boot sector contains the smallest part of LILO.
  - When it is executed, this smaller part moves itself to 0x0009A00.
  - The read mode stack is initialized.
  - The second component of LILO is loaded into RAM at 0x0009B00. When executed, this part reads a list of available OS and asks the user to choose among one of them.
  - LILO then loads the corresponding boot sector into RAM. If this is a Linux operating system, LILO essentially executes the same steps as the ones we described for the Linux boot sector. The setup() function is finally executed.

Setup Function

- **The setup function is coded in assembly language and is located in Setup.S.**
- **This function initializes various hardware devices and sets up the environment for the execution of the Kernel.**
  - It first finds the amount of RAM available.
  - Even though the BIOS already initialized most devices running on the system, Linux does not rely on it and uses its own initialization procedures. This is done to enhance portability and avoid problems in case of a defective BIOS.
  - Parameters concerning the hard disk, mouse controller and APM support are fetched from the hardware.
  - The compressed Kernel image is moved to the physical address 0x10000.
  - The interrupt descriptor table (IDT) and global descriptor table (GDT) are set up.
  - The PIC (Programmable Interrupt Controller) is reprogrammed to map 16 hardware interrupts from vector 32 to 47.
  - The CPU is switched from real mode to protected mode by setting a bit on the cr0 status register located on the CPU.
  - The startup_32() function, located at physical address 0x00100000 (if the Kernel is loaded high) is executed.

Step Three: Kernel decompression

- **The startup_32() function, located in arch/i386/boot/compressed/head.S, is executed in this step.**
- **The decompress_kernel() function is invoked, first displaying the "Uncompressing Linux…" message.**
  - The Kernel image is decompressed and placed at physical address 0x00100000 if it was originally loaded low.
  - The decompression code comes from boot/compressed/misc.c. It includes the gzip algorithms used for uncompressing the zImage or bzImage, which are both compressed with gzip.

```
Note that the difference between 'zImage' files and
'bzImage' files is that 'bzImage' uses a different
layout and a different loading algorithm, and
therefore has a larger capacity. Contrarily to
common belief among Linux users, both files use
gzip compression. The 'bz' in 'bzImage' stands for
'big zImage', not for 'bzip'!
```

  - If the Kernel image was loaded high, its decompressed code is placed just after the location of the compressed image. It is then moved to address 0x00100000.
- **The code at address 0x00100000 (the location of the decompressed Kernel image) is then executed. This actually starts the Linux Kernel.**

Step Four: Low-level initialization

- **The first part of the Kernel image is located in arch/i386/kernel/head.S and is also called the startup_32() function.**
  - This function is totally different from the one we described in step two, even if they use the same name. Besides confusing Kernel programmers, having the same name for both functions does not create any problem.
- **This second startup_32() function prepares the operating system environment to start the first process. The following steps are performed:**
  - Segmentation registers and page tables are initialized.
  - The Kernel mode stack is set for the first process (process 0).
  - The setup_idt() function is called in order to fill the IDT with null interrupt handlers.
  - Parameters of the system obtained from the BIOS and the user are stored in the first page frame.
  - Information about the current processor is gathered.
  - The address of the GDT and IDT are stored in the gdtr and idtr CPU registers.
  - The start_kernel() function is executed. In case the system contains multiple cpus, only the first one calls this function.

Step Five: High-level initialization

- **The start_kernel() function is the last step in Kernel startup.**
  - This function is coded in C, and is located in init/main.c.
  - It is architecture-independent, so no operation included in this function should involve hardware-specific elements.
- **When invoked, start_kernel() goes through the following steps:**
  - It asks for a global Kernel lock, so that only one processor can execute the initialization code.
  - It prints the Linux "banner" containing the version number, compiler used to build it, the date at which it was compiled, etc. This is taken from the variable linux_banner defined in init/version.c and can be retrieved in /proc/version.
  - The command line options passed by LILO are parsed and processed.
  - Traps and IRQs are initialized by trap_init() and init_IRQ(). This constitutes the final initialization of the IDT.
  - Data required by the scheduler is initialized by sched_init().
  - The system date and time are set up by time_init().
  - The software IRQ subsystem is initialized by softirq_init().
  - The console is initialized by console_init(), so that a remote serial console will start receiving the Kernel output.

High-level initialization (Cont'd)

- **The following steps are also executed in start_kernel() but some of them are invoked only if they were compiled in the Kernel:**
  - The dynamic module loading mechanisms are initialized by the init_modules() function.
  - Most of the slab allocator is initialized by kmem_cache_init(). The remaining part will be set by kmem_cache_sizes_init().
  - Interrupts are enabled by set().
  - The BogoMips value corresponding to the current CPU is calculated by calibrate_delay().
  - The mem_init() function initializes page descriptors.
  - Data structures used by the /proc filesystem are set up by proc_root_init().
  - The fork_init() function initializes the maximum number of threads and processes based on the amount of memory available on the system.
  - Caches for the Virtual File System are initialized in vfs_caches_init(), based on the number of pages available on the system.
  - The IPC subsystem and quota support are set up.
  - check_bugs() check the CPU for known hardware bugs.
  - A kernel thread starts executing the init() function.

High-level initialization (Cont'd)

- **Now that the basic components of the Linux Kernel are up and running, the init() thread will start invoking the initialization function for the higher-level elements of the Kernel.**
- **At this point, the CPU, memory and process management mechanisms are running. The do_basic_setup() function is called by the thread and executes the following steps:**
  - Bus-related elements are first initialized by calling pci_init(), sbus_init(), mca_init(), isapnp_init() and other similar functions.
  - do_initcalls() is invoked, which goes through the list of functions registered by means of __initcall or module_init() macros and executes them.
  - Various filesystems are initialized, along with the PCMCIA subsystem if it is available on this system.
  - The root filesystem is mounted, so that the Kernel may access files located on it.

The Init Thread

- **After do_basic_setup() has finished, the Kernel returns to the init() thread in order to continue the system initialization:**
  - The free_initmem() function is executed, which frees the initialization code that we no longer need. If you recall, various functions in the Kernel were registered with the __init macro. These functions are simply removed from memory at this point since the Kernel will never call them again (typically, an initialization function is called only once, when the system is started). Note that this is architecture-specific, so it might be unavailable for some processors.
  - The console is opened in read-write mode, so that the user may input some commands.
- **The first running process is finally started.**
  - If the "init=" option was passed to the Kernel by the loader, the program specified as the argument will be executed. This allows the local user to specify any program located on the root partition that should replace init.
  - If no init option was specified in the loader, the Kernel tries to execute, in order, /sbin/init, /etc/init, /bin/init and /bin/sh.
  - If none of these files are found on the root partition, the " No init found. Try passing init= option to kernel." Message is printed on the console. In this case, the system is completely unusable since no user can log in and daemons cannot be started.

LILO: The Linux Loader

- **LILO, the Linux boot loader, is primarily known as a tool that allows multiple operating systems to share one or more hard drives on a PC.**
- **However, LILO could also be used for other purposes.**
  - In our specific case, LILO allows Kernel developers to interactively specify from which Kernel they want to boot their computer. If a development Kernel proves to be unstable, the developer may just reboot the machine and specify another Kernel image that boots properly.
  - LILO is also very effective for passing parameters to the Kernel. For example, if you want to modify which program will be used as the init process, you may enter the following line at the LILO prompt:

    ```
    LILO: linux init=/sbin/init_test
    ```

- **We will describe in the following pages the basic structure upon which LILO is built and how it fits in the MBR, the files associated with it, its parameters, and finally its startup and error messages.**

LILO-Related Files

- **The map installer is usually located in /sbin/lilo. This program installs LILO on the computer it is executed on.**
  - Whenever the Kernel is modified and copied in /boot (this is the default location for Kernel images), the **lilo** executable should be run in order to refresh the MBR or the boot sector of the partition on which LILO is installed.
  - This program puts all files belonging to LILO at their appropriate place and stores information about the location of data needed at boot time, such as the Kernel images.
- **The /etc/lilo.conf is the LILO configuration file, which contains various settings and configurations concerning the way LILO operates on the current system.**
- **The boot loader is located in /boot/boot.b.**
  - It is the part of LILO that is loaded by the BIOS and that loads the kernel or the boot sector of other operating systems handled by LILO.
  - It also provides a simple command-line interface to interactively select the item to boot and to add boot options.
- **The map file, which is located in /boot.**
- **Kernel images, usually located in /boot (although this is not a requirement) are accessed by the map installer.**

LILO Structure In The MBR

- **The most commonly used configuration of LILO involves the boot loader being installed in the Master Boot Record (MBR).**
  - In this configuration, LILO takes control of the entire boot procedure. It may have to decide, for example, whether Linux or DOS should be loaded.
  - This has one drawback: the old version of the MBR is completely erased by the map installer in order to install LILO. Thus, the old version needs to be saved if we might need to reinstall it later.

```
Note: The MBR may be saved with the dd command:
dd if=/dev/hda of=/mnt/fd/MBR bs=512 count=1
```

- **The MBR always contains the following fields, independent of the booting technique that is used:**
  - The first 446 bytes of the MBR contain the loader program. This is often referred to as the first stage loader in LILO.
  - The next field is the partition table, which is 64-byte long.
  - The last two bytes contain a magic number, which is used to check if a given sector is a boot sector.
- **LILO cannot be stored at the following places:**
  - Boot sector of a non-Linux partition or floppy-disk
  - On a Swap partition or the second hard disk
  - Boot sector of a logical partition

LILO Startup Messages

- **When the computer boots, the "LILO" message is sent to the console. This message is also used to diagnose malfunctions in the boot process.**
  - Each letter is sent to the console after a specific step has been executed in the boot process.
- **The following sequence corresponds to the described errors:**
  - No parts of LILO have been loaded if nothing is printed. This means that LILO is not loaded, or the partition on which it is installed is not active.
  - If "L" followed by a number are sent to the console, then the first boot loader has been executed but it cannot, for some reasons, load the second boot loader. The number after the "L" indicates what type of error has occurred.
  - An "LI" message means that the two boot loaders were loaded, but the second one failed to execute. This may be caused by a disk geometry error.
  - "LIL" indicates that the second boot loader has been executed, but it cannot load the descriptor table from the map file (/boot/map).
  - "LIL?" means that the second loader is loaded at a wrong address due to a geometry mismatch.
  - "LIL-" indicates that the descriptor table is corrupted.
  - "LILO" indicates that every stage was completed successfully.

LILO Startup Error Numbers

- **As we have seen previously, an "L" followed by one of the following numbers indicates a disk error:**
  - 0x00 refers to an internal error. A file may be corrupted.
  - 0x01 indicates that the disk is not supported by the BIOS.
  - 0x02 means that the disk is experiencing some problems.
  - 0x03 indicates that write operations failed due to a read-only disk.
  - 0x04 refers to a geometry mismatch because the sector was not found.
  - 0x06 is usually a temporary flaw. Reboot and try again.
  - 0x07 indicates that the device controller was not properly initialized.
  - 0x0C indicates a media error. Reboot and try again.
  - 0x10 refers to a CRC error. This could be serious if rebooting again does not solve the problem.
  - 0x80 indicates a disk time-out, which means that the disk is not ready.
  - 0x80 means that the BIOS returned an error.
- **If the error occurred during a write operation, the number will be prefixed with a "W".**
- **Look at the LILO documentation for other, less common errors that you may encounter.**

LILO Error Messages

- **Some error messages may be presented to the user when executing the /sbin/lilo command. We will describe the most important ones.**
- ***Boot sector of hda doesn't have a [boot or LILO] signature.***
  - This indicates that the sector from which the user tried to uninstall LILO does not appear as a LILO boot sector.
- ***Can't put the boot sector on logical partition 1,***
  - This error message means that the map installer made an attempt at installing LILO on the current root partition, which is also a logical partition. This is a common problem when installing LILO on a logical partition, and we should try whenever possible to install it on a primary partition.
- ***Checksum error***
  - Indicates that the descriptor table in the /boot/map file has an invalid checksum. The file is therefore inconsistent.
- ***Device 0x01: Configured as inaccessible.***
  - Means that the corresponding device is not accessible from the BIOS.
- **Device 0x01: Got bad geometry <sec>/<hd>/<cyl>**
  - The corresponding SCSI device driver does not support automatic disk geometry, so it should be specified manually in the lilo.conf file.

LILO Error Messages (Cont'd)

- ***/dev/tape-d is not a valid partition device***
  - The device is not identified as a valid one for installing LILO.
- ***Duplicate entry in partition table***
  - The partition table is inconsistent because it contains the same entry twice.
- ***First sector of hdc doesn't have a valid boot signature***
  - The boot device specified does not have a valid boot sector, probably because the wrong device was specified in lilo.conf.
- ***geo_comp_addr: Cylinder number is too big (1248 > 1023)***
  - This indicates that LILO is trying to access a file that goes beyond the 1024th cylinder of the partition. Note that this error no longer exists for newer versions of LILO, since files located after the 1024th cylinder are now supported.
- ***Kernel is too big***
  - This could occur when Kernels located bellow 0x10000 are larger 512 KB. A simple solution to this problem is to compile the kernel as a "big zImage" (bzImage), which will load the Kernel on a higher memory address (0x100000). If this is not possible, the only way to resolve the problem is to take out some components of the Kernel.

LILO Error Messages (Cont'd)

- ***Partition entry not found***
  - This indicates that the partition entry from which another OS was supposed to be booted does not exist in the partition table.
- ***write <item>: <error_reason>***
  - The disk on which lilo tried to write the new boot sector is mounted as read-only or does not have any space left.

Questions

1. Explain why using a password for the root user is not really efficient for protecting a computer against malicious access if physical access is not properly controlled.
2. How could we modify the Kernel to fix at least this particular security problem?
3. What compression algorithm is used to compress Kernel images? What would we have to do in order to modify this algorithm or replace it with a more efficient one?
4. Why is LILO separated in two parts during the boot process?
5. What is the difference between loading the Kernel “high” and loading it “low”?
6. The number of BogoMIPS associated with the current machine is calculated during the boot process. What does the resulting number mean? By looking at the algorithm used in main.c, is the BogoMIPS value reliable?

Summary

- **Loading a PC running Linux involves five steps:**
  - System startup
  - Code loading by the boot sector
  - Kernel decompression
  - Low-level initialization
  - High-level C initialization
- **At the end of these five steps, the init process takes control of the system boot process and executes the remaining stages of the system boot process.**
- **LILO is the most commonly used program for booting Linux system.**
- **LILO replaces the code in the boot sector by its own, and allows the user to select which OS the computer should boot.**