Chapter 4

Introduction to UNIX Systems Programming

4.1 Introduction

Last chapter covered how to use UNIX from from a shell program using UNIX commands. These commands are programs that are written in C that interact with the UNIX environment using functions called *Systems Calls*. This chapter covers this Systems Calls and how to use them inside a program.

4.2 What is an Operating System

An Operating System is a *program* that sits between the hardware and the application programs. Like any other program it has a main() function and it is built like any other program with a compiler and a linker. However it is built with some special parameters so the starting address is the boot address where the CPU will jump to start the operating system when the system boots.

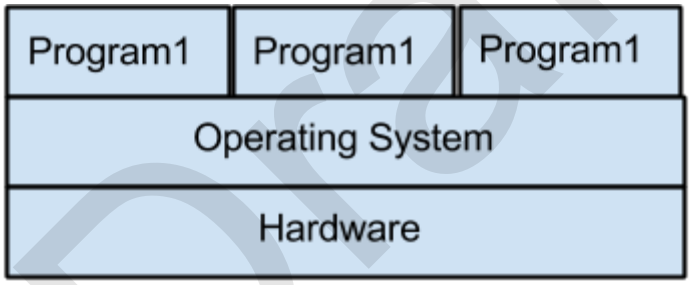


Figure 1. The Operating System interfaces the computer hardware and the user programs.

An operating system typically offers the following functionality:

- **Multitasking**
  The Operating System will allow multiple programs to run simultaneously in the same computer. The Operating System will schedule the programs in the multiple processors of the computer even when the number of running programs exceeds the number of processors or cores.
- **Multiuser**
  The Operating System will allow multiple users to use simultaneously in the same computer.
- **File system**

It allows to store files in disk or other media.

- **Networking**
  It gives access to the local network and internet
- **Window System**
  It provides a Graphical User Interface
- **Standard Programs**
  It also includes programs such as file utilities, task manager, editors, compilers, web browser, etc.
- **Common Libraries**
  It also provides libraries that are common to all programs running in the computer such as math library, string library, window library, c library etc.

The Operating System has to do all of the above in a secure and reliable manner.

Linux, MacOS, Android, and IOS are implementations of UNIX. Even though we focus in this book on UNIX, the same concepts learned in this book can be adapted to other Operating Systems such as Windows.

4.3 A Brief History of UNIX

UNIX was created in AT&T Bell Labs in 1969 by Ken Thompson, Dennis Ritchie, Brian Kernighan, and others. UNIX was a successor of another OS called MULTICS that was too big and slow for the computers at the time but it had many good ideas. UNIX was smaller, faster, and more reliable than MULTICS.

The main use for UNIX initially was the edition of documents and typesetting. It later evolved to be a general purpose Operating System that could be used to run other applications. The main way of interacting with UNIX at that time was using dumb terminals that were able to print characters in a 25 by 80 character display and take input from a keyboard. This started the use of shell programs to interact with the OS using command lines.

UNIX was initially written in Assembly Language for the Digital Equipment PDP-11 but then it was rewritten in “C” with some assembly language for some critical pieces of code. This made it easier to port UNIX to other platforms. Also, UNIX had a C compiler, linker and editors that allowed the developers to use UNIX to fix its own bugs. This “eat your own food” approach motivated the developers to create an even more reliable operating system.

One of the main successes of UNIX besides its simplicity was the commands that came with it. The commands were useful and simple to understand. The commands followed the principle of orthogonality that implies that no two commands should overlap in functionality. This kept the commands simple. Also, UNIX introduced the concept of “pipes” that allowed connecting the output of one command to the input of another one allowing the creation of more complex commands.

UNIX was a success in Universities. Students and Faculty used UNIX in PDP-11 machines that were common at that time. Researchers wanted to experiment with the UNIX internals, but since UNIX was proprietary it was not possible to change it without permission from AT&T. As a solution, the University of California at Berkeley wrote their own implementation of UNIX that provided the same commands and API as AT&T Unix. This version of UNIX was called Berkeley Software Distribution UNIX or BSD UNIX and was created in 1978.

The most known version of AT&T Unix is called Unix System V. This version was licensed to hardware manufacturers such as Sun Microsystems (that became Solaris) , Digital (that became Digital Unix) , HP (that became HP-UX) , and IBM (that became AIX) to run in their machines. On the other hand, BSD UNIX was used for research and was used to implement the first TCP/IP stack that was the basis for the Internet.

To prevent divergence among AT&T UNIX System V and BSD UNIX and all the other UNIX flavors the IEEE (Institute of Electrical and Electronic Engineers) created a standard called POSIX (Portable Operating System Interface) to define the Interface of the UNIX operating system. The hardware manufacturers agreed to follow this standard in their UNIX versions and this allowed the easy migration of software components across the different UNIX flavors by just recompiling.

It was in this environment that Richard Stallman created the GNU organization that provided Open Source implementations of many UNIX tools including compilers, editors, linkers, etc. Richard Stallman not only wrote great software like GCC, the precursor to the C/C++ compiler that is widely used now, but also was the visionary creator of the GNU General Public License or GPL. This license make the software source code available free of charge but also it asks the developers to make their contributions Open Source.

Currently there are many Open Source projects of very high quality that use the GPL Software license or other similar Open Source licenses. The fact that Open Source projects allow the access to the source code enables new generations of software developers to learn from the code of experienced computer programmers. Open Source has contributed in a big way to the education of software developers.

With the advent of personal computers (PCs) and the increase in their computing capacity at the beginning of the 1990s it was possible to run UNIX in PCs. Linus Torvalds, wrote his own implementation of the UNIX kernel and added the GNU tools to form what we know now as GNU/Linux or Linux for short. Linux has been so successful that now has become the best known implementation of UNIX. At the time of writing this book there have been 900 million Android activations and 1.5 million Android devices are activated every day. Since Android is based on Linux, we can say that GNU/Linux is the most used Operating System of all time.

4.4 Relation between UNIX and C

At the time UNIX was developed other Operating Systems were implemented in Assembly Language, making the implementation highly dependent on the CPU where it runs. Porting an Operating System written in Assembly Language to a different CPU requires rewriting the whole Operating System from scratch. Assembly Language was needed to write Operating Systems because by design they need to have access directly to the hardware and the memory of the computer. Other Computer Languages at the time such as Fortran were too high-level or too cumbersome to be used for an Operating System. Kernighan, Ritchie, and Thompson solved this problem by creating their own language called ***C***.

The C language is high-level enough to be portable but low-level enough to allow most of the code optimizations that until then were only possible in assembly language.

The C programming language was designed from the beginning to be a High-Level Assembly Language. This means that in one side it contained the high-level programming structures such as if/for/while, typed variables, and functions but on the other side it had memory pointers and arrays that allowed manipulating memory locations and their content directly.

The C Programming Language was designed to never get in your way to make your program faster.

For example, an array access in languages such as Pascal, Java, or C# checks the index against the boundaries of the array before doing an array access. If the index is out of bounds it throws an exception. This approach tells the programmer when an index-out-of bounds error happens. On the other hand, the cost in execution time is extremely high in programs that make many array operations.

During an array access C programs will not do any check of the index against the boundaries and the array access. This can result in the program reading erroneously a memory item beyond the range of the array or make the program crash with a SEGV if the memory access falls in an invalid memory page.

C allows very fast array access that is great if the program was written correctly. State-of-the-art libraries for sound and video coding and decoding are written in C and C++. Video games that need to squeeze every CPU cycle to run fast and keep the edge against their competitors are written in C and C++.

However, the same strength that makes the code run fast in C can make the program unstable and unsafe if the program is written incorrectly. C is prefered in pieces of code where the use of the CPU can become a bottleneck. Other languages such as Python, PHP, Java, C# etc are

prefered in software where the the CPU usage is not critical and the execution is spent in database access or network communication

4.5 Computer Architecture Review

Most modern computers use the ***Von Neumann Architecture*** where both program and data are stored in RAM. A modern computer has an address bus and a data bus that are used to transfer data from/to the CPU, RAM, ROM, and the devices.

When the ***CPU*** (Central Processing Unit) needs to read a word from memory it will put the memory address in the ***Address Bus*** and indicate also in the address bus that it needs to read an item from memory. The memory, either RAM (Random Access Memory) or ROM (Read Only Memory) will place the item in the ***Data Bus*** and it will be received by the CPU. When the CPU needs to write a word in memory, it will put the memory address in the address bus and the word to be written in the data bus. The RAM will store the data word at the address requested.

The communication between the CPU and the devices is very similar to the communication between the CPU and memory. Using ***Memory Mapped IO*** the devices are mapped to specific memory addresses. The CPU writes or reads to a device in the same way it writes to or reads from memory. The interrupt line is used by the devices to request CPU attention. By using interrupts the CPU does not need to waste CPU cycles waiting until a device is ready. We will see how interrupts work later in the chapter.

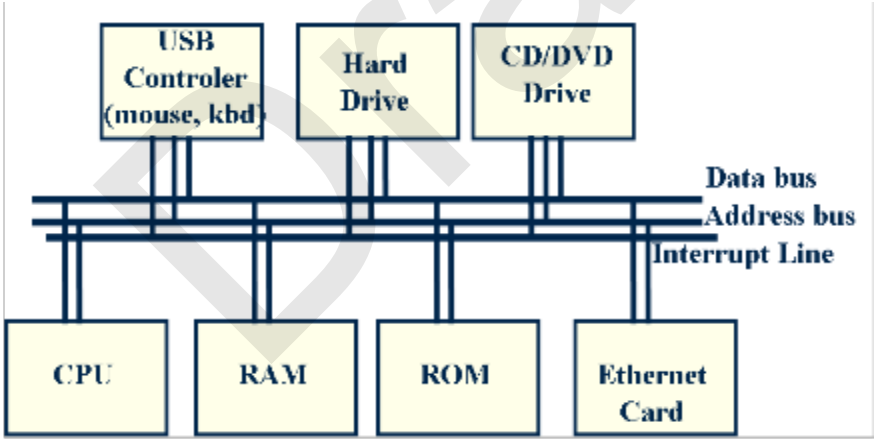


Figure 2. Architecture of a Computer showing the Data, Address Bus, and Interrupt Line.

4.6 Kernel Mode and User Mode

Modern processors have two modes of execution: ***Kernel Mode*** and ***User Mode***.

When running in ***Kernel Mode*** the CPU is able to run any type of instruction. Additionally, all registers are accessible to the program as well as all memory locations. In this mode the processor can modify any location in memory and may access any device register. In Kernel

mode there is full control of the computer. ***The Operating System services run in kernel mode.***

When running in ***User Mode*** the CPU can use only a limited set of instructions
In user mode the CPU can only modify the sections of memory assigned to the program. Also, only a subset of registers can be accessed by the CPU and it cannot access registers in devices. In user mode there is a limited access to the resources of the computer. ***The user programs run in user mode.***

Kernel Mode is also called ***Protected Mode***, and User Mode is also called **Real Mode**.

When the OS boots, it starts in kernel mode. In kernel mode the OS sets up all the interrupt vectors and initializes all the devices.Then it starts the first process and switches to user mode. The first process, often called ***init***, starts all system processes that will run in the background offering services such as secure login (sshd) and remote file systems (nfsd). These programs that run in the background offering additional Operating System services are called daemons in the UNIX world, or services in the windows world. Finally the OS runs the first user shell or windows manager.

Quick Summary

- User programs run in user mode.
- The programs switch to kernel mode to request OS services (system calls)
- Also user programs switch to kernel mode when an interrupt arrives.
- They switch back to user mode when interrupt returns.
- The interrupts are executed in kernel mode.
- The interrupt vector can be modified only in kernel mode.
- Most of the CPU time is spent in User mode

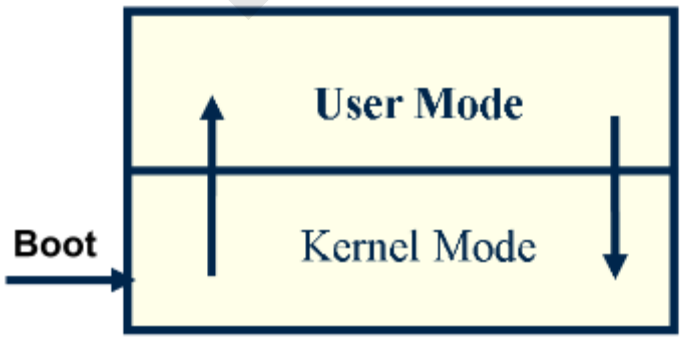


Figure 3. Going from Kernel Mode to User Mode and Viceversa.

System Calls

The System Calls of an OS is the list of services or functions that the Operating System provides. You can think of the System Calls as the API (Application Programming Interface) of the Operating System. We saw previously that System Calls run in a special mode in the CPU called ***Kernel Mode*** that uses an extended set of instructions and can access all the registers of the CPU. In contrast, application programs such as your web browser or your favorite editor run in ***User Mode*** where a restricted set of instructions can be run and only a portion of the registers is accessible. The separation of User mode and Kernel Mode give the security, protection, and reliability of an Operating System.

You can find the list of system calls in the file ***/usr/include/sys/syscall.h***. Here is an example of this file from BSD UNIX.

```
/* /usr/include/sys/syscall.h */
#define SYS_syscall     0
#define SYS_exit        1
#define SYS_fork        2
#define SYS_read        3
#define SYS_write       4
#define SYS_open        5
#define SYS_close       6
#define SYS_wait4       7
#define SYS_creat       8
#define SYS_link        9
#define SYS_unlink      10
… and many more.
```


When a new system call is added to the Operating System, it is added to the syscall.h file and a new syscall number is created. Since system call numbers are added in monotonical order the syscalls.c file also gives you a history of how the UNIX operating system evolved.

When an application program runs and invokes a system call like open() in user mode it generates a "software interrupt" to cross the user/kernel mode boundary. Then the System Call for open starts running in Kernel Mode where it checks the arguments and validates that the arguments are correct and that the owner of the process can open the file. Then, it performs the operation and returns the file handler of the open file. If there is an error in any of the arguments, the system call will return -1 and it will set a global variable called "***int errno".***

This global variable "***int errno***" is defined in the standard C library ***libc.so*** and stores the status of the last system call executed. It is either 0 on success or an error number. The list of all the errors can be found in ***/usr/include/sys/errno.h***

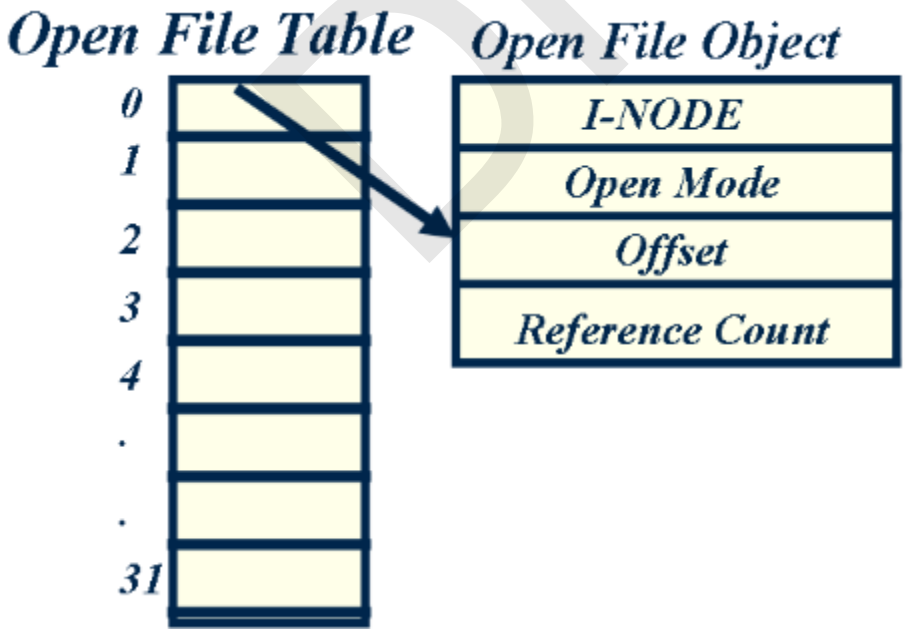
```
/* /usr/include/sys/errno.h */
#define EPERM   1       /* Not super-user    */
#define ENOENT  2       /* No such file or directory */
#define ESRCH   3       /* No such process */
#define EINTR   4       /* interrupted system call  */
#define EIO     5       /* I/O error  */
#define ENXIO   6       /* No such device or address */
… and many more
```

You can print a human readable error message that corresponds to errno to stderr using ***perror(s)***; where s is a string prepended to the message.

The Open File Table

The process table has a list with all the files that are opened. Each open file descriptor entry contains a pointer to an open file object that contains all the information about the open file. Both the **Open File Table** and the **Open File Objects** are stored in the kernel.

System calls like **write**/**read** refer to the open files with a file descriptor that is an index into the table. The maximum number of file descriptors per process is about 256 by default but but it can be changed with the shell command ***ulimit*** up to 1024. or more.



File Table and Open File Table

An Open File Object contains the state of an open file with the following entries:

- I-Node – It uniquely identifies a file in the computer. An I-nodes is made of two parts:
- Major number – Determines the devices
- Minor number –It determines what file it refers to inside the device.
- Open Mode – How the file was opened: Read Only, Read Write, Append
- Offset – The next read or write operation will start at this offset in the file. Each read/write operation increases the offset by the number of bytes read/written.
- Reference Count – It is increased by the number of file descriptors that point to this Open File Object. When the reference count reaches 0 the Open File Object is removed. The reference count is initially 1 and it is increased after fork() or calls like dup and dup2. In UNIX also the reference count is increased when the file is opened. This will prevent a file to be removed while it is still opened by the Operating System.

When a process is created, there are three files opened by default:

0 – Default Standard Input
1 – Default Standard Output
2 – Default Standard Error

write(1, “Hello”, 5) Sends Hello to stdout
write(2, “Hello”, 5) Sends Hello to stderr

Stdin, stdout, and stderr are inherited from the parent process.

The open() system call

The ***open*** system call opens the file in filename using the permissions in ***mode***.

int open(filename, mode, [permissions]),

The values in mode can be:

- O_RDONLY - Open the file in read-only mode. write operations are not allowed.
- O_WRONLY - Open the file in write-only mode. read operations are not allowed.
- O_RDWR - Open the file in read-write mode. Both read and write operations are allowed.
- O_CREAT If the file does not exist, the file is created.Use the permissions argument for the initial permissions. Bits: rwx(user) rwx(group) rwx (others) Example: 0555 – Read and execute by user, group and others. (101B==5Octal)
- O_APPEND. Append at the end of the file.
- O_TRUNC. Truncate file to length 0.

See “man open” for more details.

The close() System Call

The close system call closes a file.

void close(int fd)

close(fd) decrements the count of the open file object pointed by **fd.** If the reference count of the open file object reaches 0, the open file object is reclaimed.

The fork() System Call

The fork() system call creates a new process that is copy of the parent process that is calling fork().

int fork();

This is the only way to create a new process in UNIX.

The call :

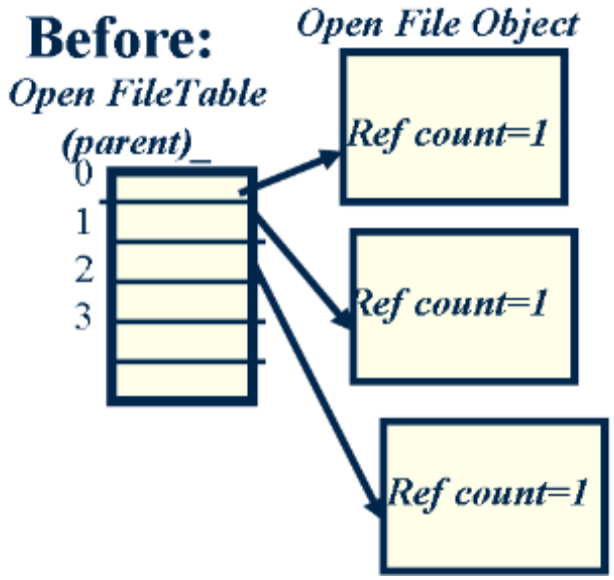
```
int ret;
ret = fork();
```

returns:

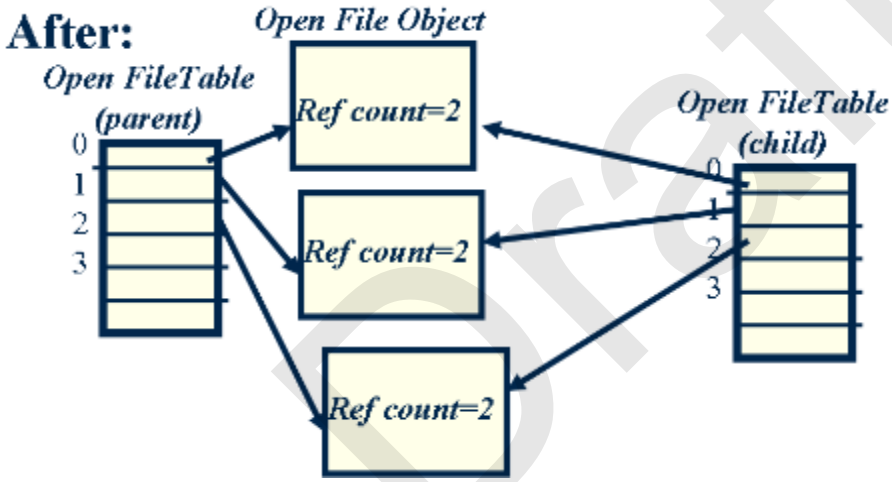
- ret == 0 in the child process
- ret == pid > 0 in the parent process.
- ret < 0 if there is an error

The memory in the child process is a copy of the parent process's memory. This copy is optimized by using VM copy-on-write, that is, the memory of the parent will be shared with the child keeping only one copy the memory in physical memory. Only when one page is modified by either the parent or the child, the OS will make a copy of the modified page. This "lazy" copy improves the execution of fork() since most of the time only a few pages are modified.

During fork() the Open File table is copied in the child. However, the Open File objects of the parent are shared with the child. This allows the communication between the parent and the children. Only the reference counters of the Open File Objects are increased.



Open File Table and File Objects Before fork()



Open File Table and File Objects After fork().

As you see in the figure, both parent and child process have different Open File Tables but they share the same open file objects. By sharing the same open file objects, parent and child or multiple children can communicate with each other. We will use this property to be able to make the commands in a pipeline communicate with each other.

The execvp() system call

The execvp system call loads a new program in the current process.

int execvp(progname, argv[])

During execvp, the old program is overwritten. ***progname*** is the name of the executable to load. ***argv*** is the array with the argument where ***argv[0]*** is the progname itself. The entry after the last argument in ***argv*** should be a ***NULL*** so ***execvp()*** can determine where the argument list ends. If successful, ***execvp()*** will not return since the current program is overwritten by the new program.

The following example shows runs "ls -al" from a C program using execvp.

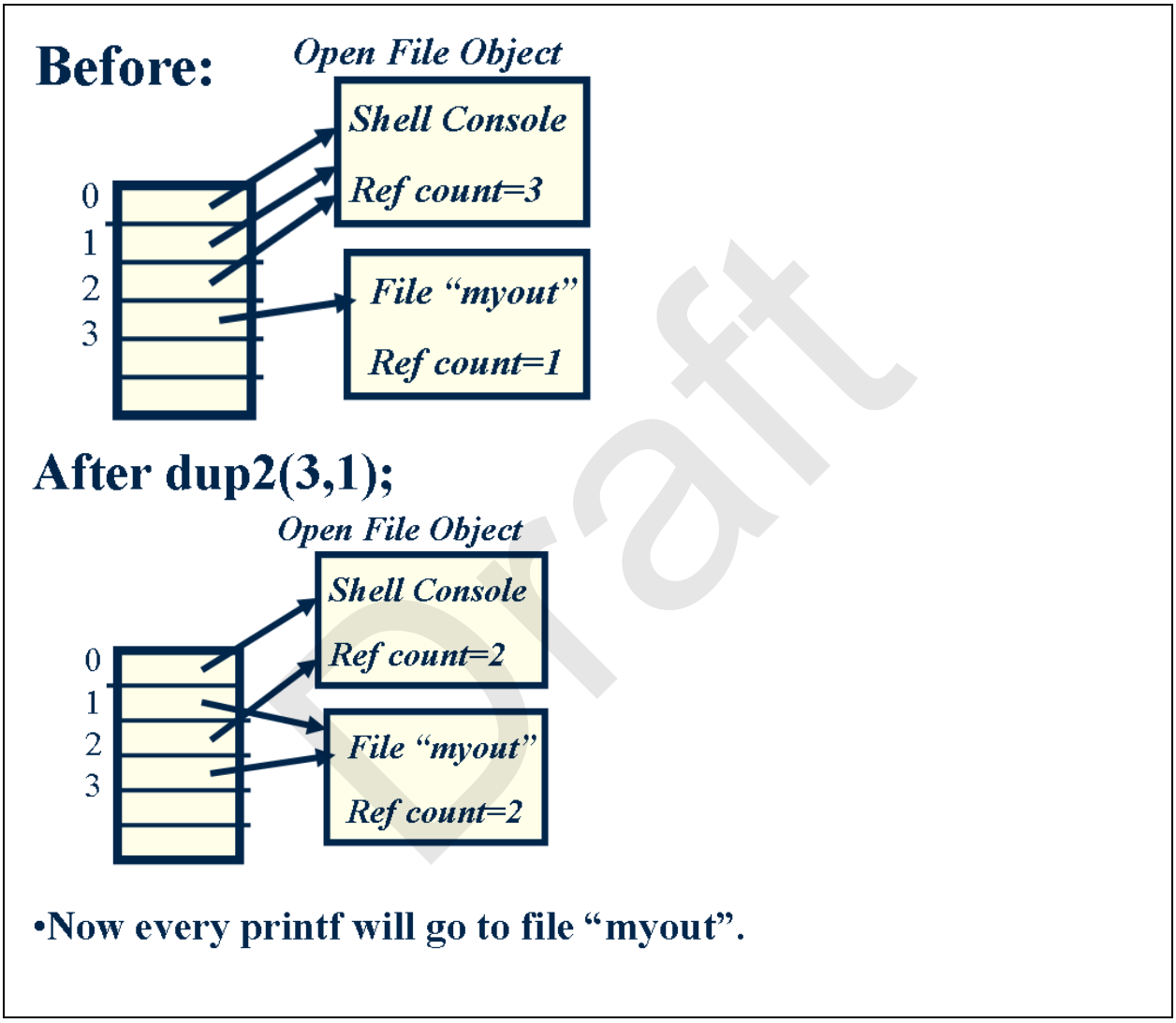
```
void main() {
  // Create a new process
  int ret = fork();
  if (ret == 0) {
     // Child process.
     // Execute "ls -al"
     const char *argv[3];
     argv[0]="ls";
     argv[1]="-al";
     argv[2] = NULL;
     execvp(argv[0], argv);
     // There was an error
     perror("execvp");
     _exit(1);
  }
  else if (ret < 0) {
     // There was an error in fork
     perror("fork");
     exit(2);
  }
  else {
     // This is the parent process
     // ret is the pid of the child
     // Wait until the child exits
     waitpid(ret, NULL);
  } // end if
}// end main
```


The dup2() System Call

The dup2 system call is used to redirect a file descriptor to a different file object.

int dup2(fd1, fd2)

After calling dup2(fd1, fd2), fd2 will refer to the same open file object that fd1 refers to. The open file object that fd2 referred to before is closed. The reference counter of the open file object that fd1 refers to is increased. dup2() will be useful to redirect stdin, stdout, and also stderr when working on the shell project.



Example program that redirects stdout to a file myoutput.txt

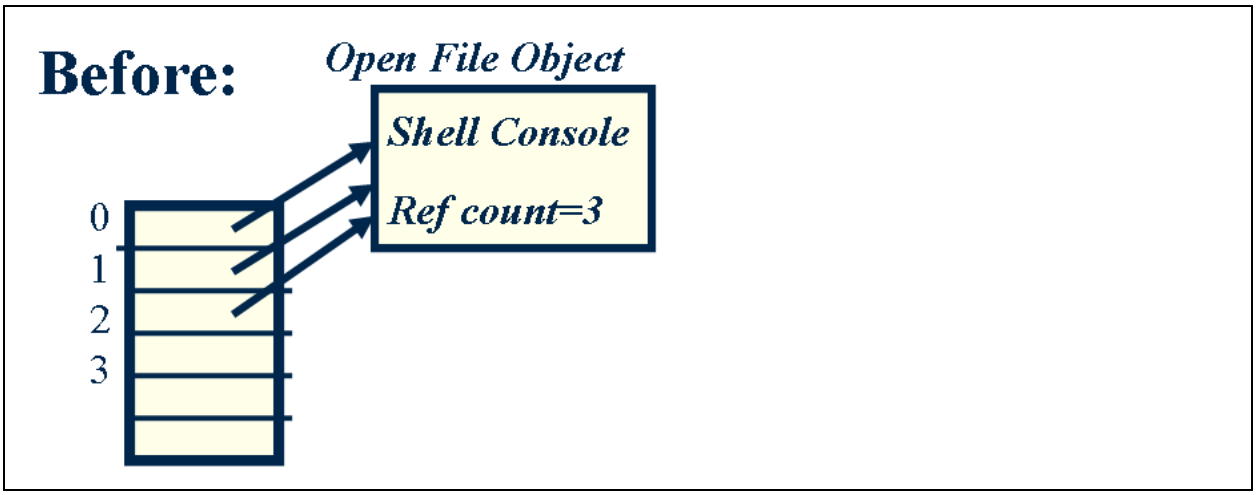
```
int main(int argc,char**argv)
{
  // Create a new file
  int fd = open("myoutput.txt",
     O_CREAT|O_WRONLY|O_TRUNC,
     0664);
  if (fd < 0) {
     perror("open");
     exit(1);
  }
  // Redirect stdout to file
  dup2(fd,1);

  // Now printf that prints
  // to stdout, will write to
  // myoutput.txt
  printf("Hello world\n");
}
```

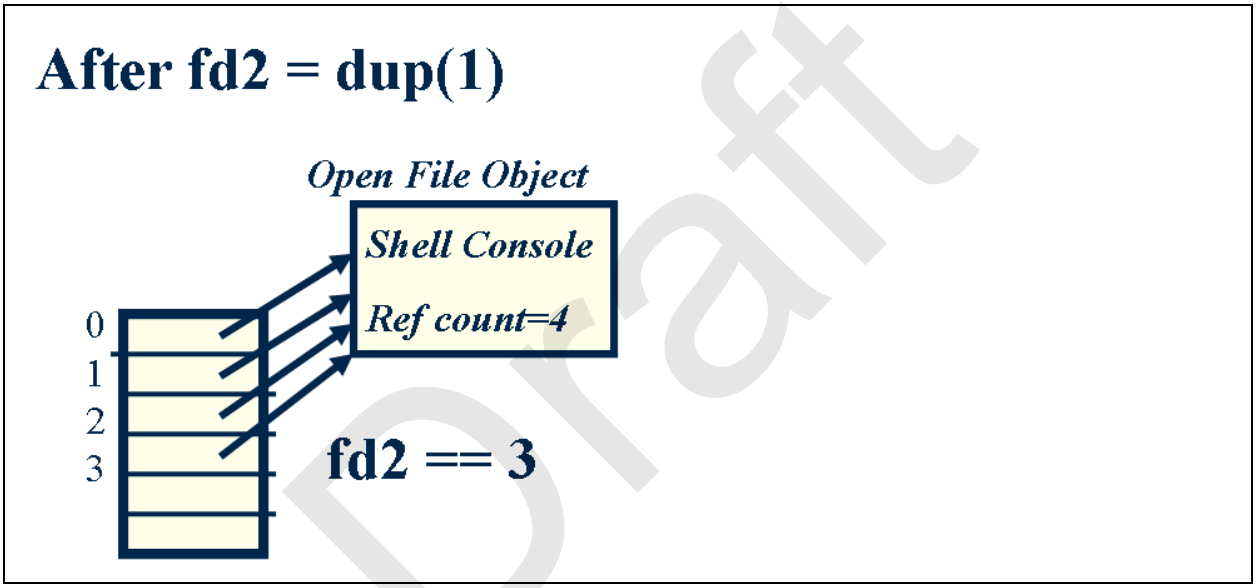

The dup() System Call

The dup system call is used to create a different file descriptor to an existing file object.

fd2=dup(fd1)

dup(fd1) will return a new file descriptor that will point to the same file object that fd1 is pointing to. The reference counter of the open file object that fd1 refers to is increased. This will be useful to "save" the stdin, stdout, stderr, so the shell process can restore it after doing the redirection.



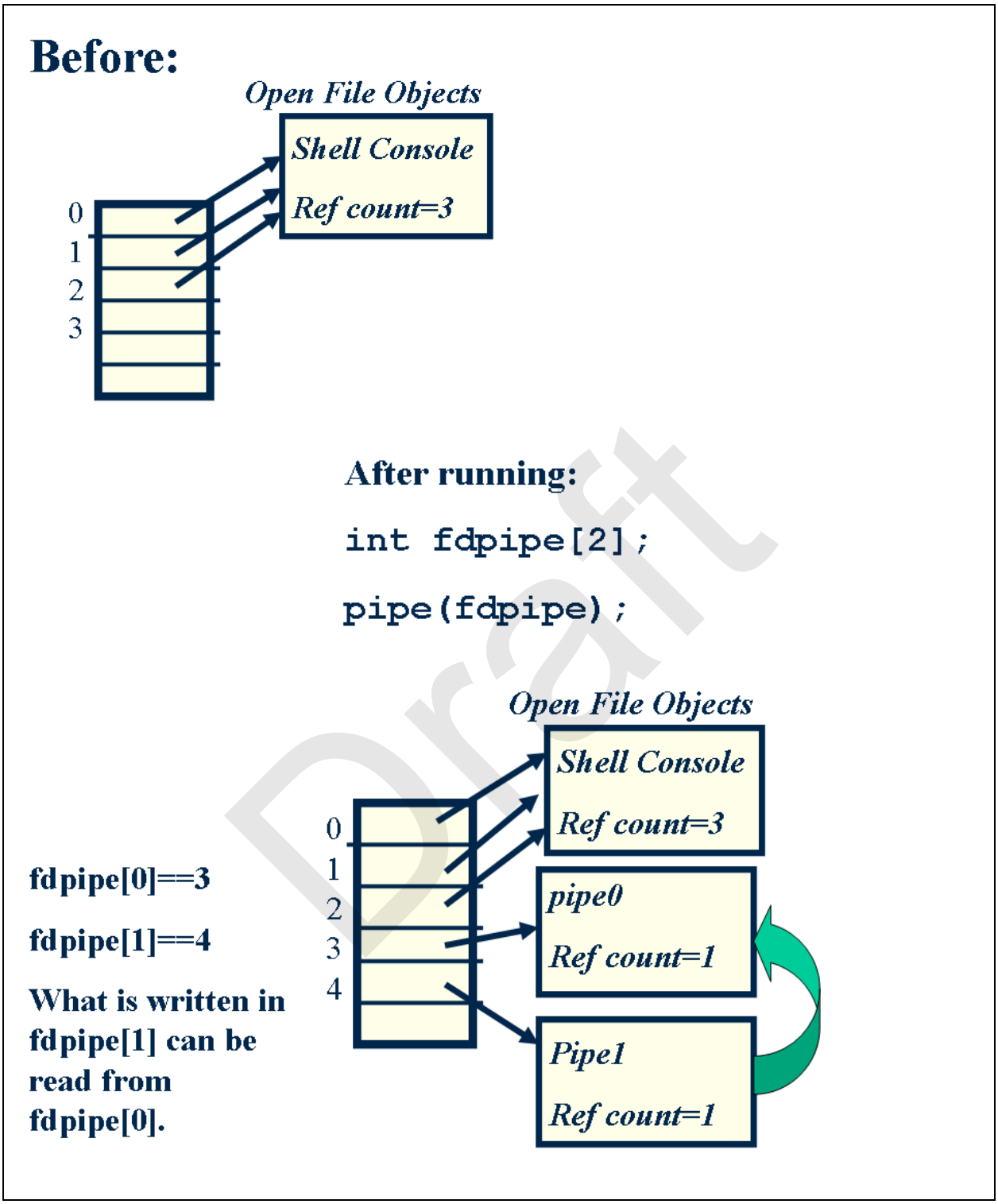


The pipe() system call

The pipe system call creates a pipe that can be used for interprocess communication.

int pipe(fdpipe[2])

fdpipe[2] is an array of int with two elements. After calling pipe, fdpipe will contain two file descriptors that point to two open file objects that are interconnected. What is written into fdpipe[1] can be read from fdpipe[0]. In some Unix systems like Solaris pipes are bidirectional but in Linux they are unidirectional.



Here is an example of implementing a program that executes “lsgrep” that runs “ls –al | grep arg1 > arg2”. Example: “lsgrep aa myout” lists all files that contain “aa” and puts output in the file myout.

```
int main(int argc,char**argv)
{
    if (argc < 3) {
      fprintf(stderr, "usage:”
        “lsgrep arg1 arg2\n");
      exit(1);
    }

    // Strategy: parent does the
    // redirection before fork()
    //save stdin/stdout
    int tempin = dup(0);
    int tempout = dup(1);

    //create pipe
    int fdpipe[2];
    pipe(fdpipe);

    //redirect stdout for "ls“
    dup2(fdpipe[1],1);
    close(fdpipe[1]);
    // fork for "ls”
    int ret= fork();
    if(ret==0) {
      // close file descriptors
      // as soon as are not
      // needed
      close(fdpipe[0]);
      char * args[3];
      args[0]="ls";
      args[1]=“-al";
      args[2]=NULL;
      execvp(args[0], args);
      // error in execvp
      perror("execvp");
      _exit(1);
    }

    //redirection for "grep“
    //redirect stdin
    dup2(fdpipe[0], 0);
    close(fdpipe[0]);
```

```
    //create outfile
    int fd=open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0600);
    if (fd < 0){
      perror("open");
      exit(1);
    }

    //redirect stdout
    dup2(fd,1);
    close(fd);

    // fork for "grep"
    ret= fork();
    if(ret==0) {
      char * args[3];
      args[0]="grep";
      args[1]=argv[1];
      args[2]=NULL;
      execvp(args[0], args);
      // error in execvp
      perror("execvp");
      _exit(1);
    }

    // Restore stdin/stdout
    dup2(tempin,0);
    dup2(tempout,1);
    // Parent waits for grep
    // process
    waitpid(ret,NULL);
    printf("All done!!\n");
} // main
```

In this program the parent performs all the redirection before executing fork(). In this way the child starts already running with the input and output already redirected to the right files. To be able to restore the input and the output at the end, the parent process has to ***save*** the input and output before it starts the redirection.