Computer Science 392

Lecture Notes

Systems Programming

Shudong Hao

January 13, 2024

Computer Science 392

Disclaimer
The lecture notes have not been subjected to the usual scrutiny reserved for formal publications. They may not be distributed outside this class without the permission of the Instructor.

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
https://github.com/fmarotta/kaobook

Edition History
0th edition (lecture notes): January 2022;
1st edition: January 2023;
2nd edition: June 2023/January 2024.

Contents

List of Figures

List of Tables

List of Listings

What the Shell? 1

Most people get familiar with Linux system is through terminal and commands. It is the most straightforward way to execute programs and services provided by the system. When we open a terminal on a GUI (Graphic User Interface), we will usually see a prompt (an arrow, or a dollar sign), followed by a blinking cursor. The terminal invokes a program called **shell**, which interprets our commands, and executes program requested through those commands with arguments.

We will start our chapter with a brief walk-through of the system and terminal commands, and move on to writing bash scripts.

1.1 File Organization

All the files are organized as a tree structure in the Linux system, where it all begins with a root " `/` ", called **root directory** (see Figure 1.1). There are several special directories (or folders) under `/`, which is the same structure of any standard version (or distribution) of Linux system. Some directories are listed below:

- `/bin/` : Executables are also called binaries, and that's why the directory is called `/bin/`. Note that `/bin` contains only binaries necessary for system use. Another directory, `/usr/bin` contains more user-customized binaries;
- `/dev/` : Every file in `/dev/` represents some physical or logical device. Each actually contains a device driver, *i.e.*, an executable program. The interesting part is that you never "run" these files. Instead you read from them or write to them as if they were ordinary text files. We will visit this directory in future lessons;
- `/home/` : A Linux system is usually shared by multiple users. When we create a new user with a user name (say `<uname>`), the system creates a directory under `/home/` with the name `<uname>`, which is the **home directory** of the user;
- `/root/` : Even though this directory is called " `root` ", it is **not** the root directory `/` we mentioned before. For security reasons, every Linux system has a **superuser** that can check certain directories and execute certain programs that regular users cannot. For this superuser, its name is called literally `root`, and `/root/` is its home directory.

Of course there are other system directories, and we will introduce and learn them at more relevant places.

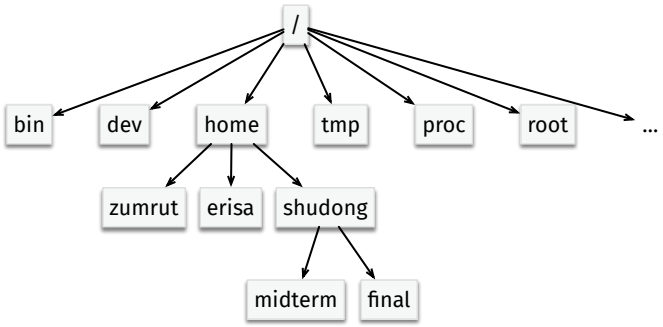


Figure 1.1: An example of file hierarchy in Linux systems.

1.1.1 Navigating through File Hierarchy

When we open the terminal (and thus bash), we are automatically at the home directory: `/home/<uname>/` where `<uname>` is the actual user name. It is also abbreviated to just a `~`, and from there we can go to other directories in the file hierarchy. To do this, the command is `cd`, or **c**hanging **d**irectory:

```
$ cd <path>
```

where `<path>` needs to be either a **relative path** or an **absolute path**.

- **Relative path** is the path that's relative to our current place in the file hierarchy. Using Figure 1.1 as an example. Assume I log into the system using my account, and I'm automatically at my home directory `/home/shudong/`. If I want to visit `/proc`, I can use the following command:

  ```
  $ cd ../../proc/
  ```

  where `..` means we are going back one level of directory. Thus, the first `../` will take me back to `/home/`, and the second will take me back to `/`. A single `.` refers the current directory;

- **Absolute path** refers to the path that always starts from the root directory `/`. Using the same example as above but with an absolute path, the command will be:

  ```
  $ cd /proc/
  ```

If `<path>` is empty, it automatically takes us to the home directory.

⚠ Caution!

One thing to pay particular attention is that if we want to go to a subdirectory, we can just type its name after `cd`, or relative path, such as `$ cd midterm` or `$ cd ./midterm`. As long as we start the path with a `/`, that's an absolute path that starts with the root.

1.1.2 Examining Files

When we are at a directory, we can use command `pwd` to print out the current directory. It's also called **working directory**.

Another command we use a lot is `ls`, which is to list the names of all the files under a specific directory:

```
$ ls <path>
```

When `<path>` is empty, it lists all the files under current directory.

An interesting command that actually shows the tree structure of the file hierarchy is `tree`:

```
$ tree <path>
```

It formats the output nicely so you can see the actual tree structure. See Figure 1.2.

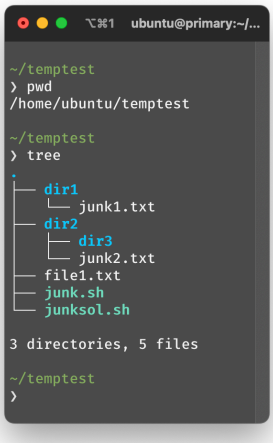


Figure 1.2: An example of the output of `tree` command.

To show the file content on terminal, we can use `cat` command:

```
$ cat <file>
```

If we want to get some basic statistics of a file, we can use `wc` command, which is short for **w**ord **c**ount:

```
$ wc <file>
```

which will print the number of lines, as well as word and byte counts for the `<file>`.

1.1.3 Creating and Deleting Files

To create a directory, we can use command `mkdir` followed by the directory name or a path. It is a bit tricky to create a file, because we need to put content in it. If just an empty file, we can use `touch`.

To delete a file, the command is `rm` followed by the path to the file. To delete a directory, the command is also `rm` but we need to add a flag `-r` which stands for "recursively".

A very important thing to notice is that Linux systems trust their users. When you're deleting something, it'll be deleted permanently (*i.e.*, there's no such a thing as "recycle bin" as in Windows). Also, something like this is very dangerous: `rm -rf /`, which is to delete the root directory. The system will not stop you from doing this, and because it is the root directory that's being deleted, all your stuff will be gone!

1.2 Environment Variables

When introducing file organization in the beginning, we mentioned that `/bin` stores all the executables. In fact, all the commands we are using

in the terminal are also the names of the corresponding executables. For example, when we type `ls` in the terminal, the terminal invokes an executable `/bin/ls` (or `/usr/bin/ls`).

Now we have a question — how does the shell know where to find this executable? There's a set of variables stored in the system called **environment variables** that actually points the shell to the correct location.

We can simply use command `env` to print out the environment variables:

```
USER=ubuntu
LOGNAME=ubuntu
HOME=/home/ubuntu
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
↪ sbin:/bin:/usr/games:/usr/local/games:/snap/bin
SHELL=/usr/bin/zsh
TERM=xterm-256color
XDG_SESSION_ID=48
XDG_RUNTIME_DIR=/run/user/1000
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
XDG_SESSION_TYPE=tty
XDG_SESSION_CLASS=user
MOTD_SHOWN=pam
LANG=C.UTF-8
SSH_CLIENT=192.168.64.1 60971 22
SSH_CONNECTION=192.168.64.1 60971 192.168.64.5 22
SSH_TTY=/dev/pts/0
SHLVL=1
PWD=/home/ubuntu
OLDPWD=/home/ubuntu
ZSH=/home/ubuntu/.oh-my-zsh
PAGER=less
LESS=-R
_=/usr/bin/env
```

Each line is an environment variable and its value in the format of `NAME=value`. If a variable contains multiple values, they'll be connected by a colon, *i.e.*, `NAME=value1:value2:value3`. If we want to see individual variable, we can use `echo $NAME` to print the value(s) out. Notice the dollar sign before the variable name `NAME`.

Even though we don't have to know all the variables, some of them are actually very important. First one is `HOME`, and that's why when we type `$ cd ~` it can automatically find your home directory.

Another one is `PATH`. When we invoke a command such as `ls`, the shell will iterate over all the paths listed in `PATH` to see if any of them contains an executable `ls`.

The environment variables will change depending on the actual "environment". For example, try the following: go to home directory and check out `PWD`; then go to root directory and check out `PWD` again, and you'll

see the change. The environment variables can be modified manually as well.

1.3 Special Characters

For every running program, there are three standard streams by default: `stdin` for receiving user input to the terminal, `stdout` for printing output to the terminal, and `stderr` for printing error messages to the terminal. All three streams are connected to the terminal, especially `stdout` and `stderr`. One of the reasons why we separate regular output and error message output is for logging purposes.

For now, you can just remember that each of the three standard streams has an integer number representing it: 0 for `stdin`, 1 for `stdout`, and 2 for `stderr`.[1]

1: These integer numbers are called file descriptors, which will be discussed in detail in Chapter 4.

1.3.1 Redirection

Sometimes we don't want everything to be printed on the screen; maybe because there are too many output and it got messy, or we want to save the output so we can refer to later. In this case, we can use redirection.

Try the following in your terminal:

```
$ echo hello world
$ echo hello world > temp.txt
```

The first line is just a normal `echo` command, and so by default it'll print everything on the terminal. In the second line, because we use the redirection, the output will be put into a file we specified.

The operator `>` is to redirect the output. If the destination file already exists, it'll overwrite the file. If we want to just append to the existing file, we use `>>`:

```
$ date > temp.txt
$ hostname >> temp.txt
```

We can also redirect input, but we'll postpone it till Section 1.4.5.

1.3.1.1 Make Output Disappear

There's a special file in the system located at `/dev/null`. If you redirect the output to there, all output will be gone without a trace. Try this:

```
$ ls /etc/ > /dev/null
```

1.3.1.2 Standard Error

When we are using `>`, it is by default only redirects `stdout`. If any messages are printed through `stderr` stream, it will still show up on the terminal. If we want to redirect `stderr` instead, we can use `2>`, since `2` stands for `stderr`:

```
$ cd doesntexist 2> errorlog.txt
```

assuming `doesntexist` is a directory that doesn't exist.

Let's look at the following example. First, remember `echo` will print the string to `stdout`. If we want it to print to `stderr`, we can do the following:

```
$ echo "what the shell?" >&2
```

The problem is it prints to terminal just like any message, and we are not sure if it's correctly printed through `stderr`. A very simple way to verify is to use redirection:

```
$ (echo "what the shell?" >&2) > err.txt
```

We notice the string `"what the shell?"` is still printed on the screen. This is because `>` only redirects `stdout`, which shows us that our string is indeed not printed through `stdout`. If we do this:

```
$ (echo "what the shell?" >&2) 2> err.txt
```

no string is printed, and `err.txt` successfully received the string, so our message is correctly printed through `stderr`.

If we want to merge `stderr` to `stdout`, we can combine the streams by using `2>&1`. For example,

```
$ ((echo "what the shell?" >&2); echo "392") > log.txt 2>&1
```

What this command does is to echo `"what the shell?"` first through `stderr`, and then `"392"` through `stdout`.[2] It redirects the output to `log.txt` but also merges `stdout` and `stderr`, so the file contains both strings.

2: The semicolon in the middle is to connect two separate commands together, so that they are executed sequentially and independently. Do not confuse this with the pipe operator in the next section.

Just for "fun", here are some really annoying stuff. Say we have a command called `CMD` that produces output to both `stderr` and `stdout`. We already know how to get rid of `stderr`:

```
$ CMD 2>/dev/null
```

What about the following:

```
$ CMD 2>&1 >/dev/null
```

And what about this?

```
$ CMD >/dev/null 2>&1
```

The first one is to get rid of `stderr`, which is equivalent to `$ CMD 1> /dev/null`. The second one is to get rid of both `stderr` and `stdout`.

1.3.2 Pipes

So far we've dealt with sending data to and from files. Now we'll take a look at a mechanism for sending data from one program to another. It's called piping and the operator we use is " `|` " (found above the backslash key on most keyboards). What this operator does is feeding the output from the program on the left as input to the program on the right:

```
$ prog1 | prog2 | prog3 | ....
```

where the `stdout` output from `prog1` will be fed into `prog2` as `stdin`, and so on.

The example we use here is to count the number of files in a directory. We have mentioned that `ls` can list the names of all files under a specific directory, and `wc` can count the number of words in a file. When using `wc` without a file name, `wc` will do the statistics on `stdin`. Thus, we can use the following commands with pipe:

```
$ ls | wc -w
```

where `-w` specifies that we only want the number of words.

Why does this work? When we examine the output from `ls`, we notice that between each pair of file names there are several spaces. When `wc` counts "words", it actually takes spaces as delimiters. Thus, the number of words is actually the number of files.

What if we have the following command?

```
$ ls | wc -w | wc
```

We can certainly combine pipe operators with redirection, such as:

```
$ ls | wc > stat.txt
$ cat stat.txt
```

1.4 Bash Script

For simple tasks, such as checking a file information, typing a command on the terminal is enough. For more complicated system-related tasks, it is very tedious to type all the command one by one. Remember when we were learning Python, in the beginning we used a terminal as well, but when the program becomes more complicated where we have to consider control structures and storing variables, we need to write the program in a Python script, and so we can just invoke the script from terminal, and it'll be executed from beginning to end.

It is very similar here as well. We write a script where it contains all the commands we want to execute, and invoke the script directly. Over the years, many different variations of the shell have appeared with interesting features, but the essential part and syntax are still very much alike. One of the commonly used is called ***B**ourne **A**gain **SH**ell*, or **bash**. This is also the shell we are going to learn in detail in this chapter. [3]

3: We'll use the name "shell" and "bash" interchangeably, but you should know we are following bash syntax.

The following is a very simple bash script:

```
1 #!/bin/bash
2 # hello.sh
3 echo "Hello World!"
```

Line 1 is a must: there are many difference interpreters to run our script, and they have slightly different syntax as well. Since we want it to be `bash`, we **start all your bash scripts with the line** `#!/bin/bash`. (Go to Section 1.4.2 to see why this matters). Also notice it is under `/bin/` directory. As we discussed in the previous section, everything under `/bin/` is an executable binary. Therefore, bash is an **interpreter**.

Line 2 is a comment, starting with `#`. Note that there's no block comments in bash, only single lines.

Line 3 is a statement in our script, corresponding to a command we'd use on terminal. `echo` prints out anything that follows it.

> **⚠ Caution!**
>
> When assigning value to a variable, do not put spaces around the equal sign! `STR="hello"` is to assign the string literal `"hello"` to variable `STR`, while `STR = "hello"` is to **compare** `STR` and `"hello"` and return a 0 or 1.

1.4.1 Evaluation

Whenever we see something starts with a `$`, that means we want to get the value out of it. For example, here's how we define a variable and how to use it:

Listing 1.1: Declaring a string.

```
1 #!/bin/bash
2 # variable.sh
3 STR="Hello World!"
4 echo STR
5 echo $STR
```

Notice we defined a global variable `STR` with a string. When we `echo` it without `$`, it simply prints out `STR`; but once we added a dollar sign in front it, it evaluates `STR`, and prints out the value of it.

1.4.1.1 Read Only Variables

All variables can change their values unless they are declared with `readonly` keyword:

```
#!/bin/bash
# readonly.sh
readonly STR="Hello World!"
echo $STR
STR="Changed my mind!"
echo $STR
```

Listing 1.2: Using `readonly` keyword will prevent the value of a variable being changed after its definition.

1.4.1.2 Commands

Each statement in a bash script runs a command, just like running a program. Remember when we run a C program, we always have a return value. Here, after running a command, we also have a return value from that command, and sometimes we need those values for our use.

A typical syntax is:

```
$( command )
```

A silly example is we want to store the current working directory into a variable, and print it out. This is not particularly interesting because we can simply just write `pwd`; but just use it for demonstration:

```
CURR_PATH=$(pwd)
echo $CURR_PATH
```

If we want to run multiple commands, the syntax is:

```
$(command1; command2; ...; command_N)
```

and the evaluation result is the last command. See the following example:

```
#!/bin/bash
# command.sh
PREV_PATH=$(cd ..; pwd)
echo $PREV_PATH
```

Listing 1.3: Running mutiple commands in bash.

1.4.1.3 Arithmetic

Bash scripts are so simple that it can only do integer calculations. To do floating points, we'd have to use built-in utilities like `bc` .

Note that for calculating numbers, it's also essentially to *evaluate* an expression. Therefore, the syntax is very similar:

```
$(( math-expression ))
```

which will return the evaluated result. Notice here we use double parenthesis for math expression, and single parenthesis for system commands.

Try the following:

```
arg1=20
arg2=10
vol=$(( 10 * $arg1 * $arg2 ))
echo $vol
```

1.4.2 Conditionals

The conditional structure is very similar to other programming languages:

```
if [[ Condition_1 ]]; then
    # If condition_1 is TRUE
elif [[ Condition_2 ]]; then
    # If condition_1 is FALSE and condition_2 is TRUE
else
    # If both condition_1 and condition_2 are FALSE
fi
```

The conditions can be one of the listed in Table 1.1.

The following example shows us how to use conditional expressions:

Listing 1.4: Using simple `if-else` structure in bash.

```
#!/bin/bash
# ifelse.sh

CURR_PATH=$(pwd)
if [[ $CURR_PATH = "/Users/shudong/392demo/bash" ]]; then
    echo "Yay!"
else
    echo "Nay!"
fi
```

Bash interpreter is built on the plain ol' shell script, so it has some features that shell script doesn't have. For example, in bash, when comparing strings, `==` and `=` are equivalent, and we use `[[]]` , *i.e.*, double brackets. These are invalid in shell scripts, however. Try the following script, and notice that we change the first line to using shell interpreter: [4]

4: This example will not work on macOS.

Table 1.1: Conditions for bash. **Note** that `<` and `>` are for **ASCII comparison**, not numerical. See example: https://tldp.org/LDP/abs/html/comparison-ops.html#LTREF.

Operator	Description
`! EXPRESSION`	The `EXPRESSION` is false.
`-n STRING`	The length of `STRING` is greater than zero.
`-z STRING`	The length of `STRING` is zero (*i.e.*, an empty string).
`STRING2 = STRING2`	`STRING1` is equal to `STRING2`.
`STRING1 != STRING2`	`STRING1` is not equal to `STRING2`.
`INTEGER1 -eq INTEGER2`	`INTEGER1` is numerically equal to `INTEGER2`.
`INTEGER1 -gt INTEGER2`	`INTEGER1` is numerically greater than `INTEGER2`.
`INTEGER1 -lt INTEGER2`	`INTEGER1` is numerically less than `INTEGER2`.
`-d FILE`	`FILE` exists and is a directory.
`-e FILE`	`FILE` exists.
`-r FILE`	`FILE` exists and can be read.
`-s FILE`	`FILE` exists and it's size is greater than zero (*i.e.*, not an empty file).
`-w FILE`	`FILE` exists and can be written.
`-x FILE`	`FILE` exists and can be executed.

```sh
#!/bin/sh
# ifelse.sh
CURR_PATH=$(pwd)
if [[ $CURR_PATH == "/Users/shudong/392demo/bash" ]]; then
    echo "Yay!"
else
    echo "Nay!"
fi
```

Thus, we recommend using bash syntax all the way to avoid confusion.

1.4.3 Arrays & Loops

Arrays in bash are very similar to lists in Python, meaning the elements in an array don't have to be of the same type. To declare arrays, we can use `declare` keyword, or simply use assignment:

```sh
#!/bin/bash
# array.sh

# Method 1:
declare -a array
array[0]="hi"
array[1]="bye"

# Method 2
newarray=(1 2 "three" 4 "five")
```

Listing 1.5: Declaring arrays.

It's also easy to print out array elements individually, or all elements at once:

Table 1.2: Common operations for arrays in bash.

Syntax	Description
`arr=()`	Create an empty array.
`arr=(1 2 3)`	Initialize array.
`${arr[2]}`	Retrieve the third element.
`${arr[@]}`	Retrieve all elements.
`${!arr[@]}`	Retrieve array indices.
`${#arr[@]}`	Calculate array size.
`arr[0]=3`	Overwrite the first element.
`arr+=(4)`	Append value.
`str=$(ls)`	Save `ls` output as a string.
`arr=( $(ls) )`	Save `ls` output as an array of strings.
`${arr[@]:s:n}`	Retrieve n elements starting at index s.

Listing 1.6: Printing array elements or size.

```bash
# Print elements
echo ${array[0]}
echo ${array[1]}

# Print all elements
echo ${array[*]}
echo ${array[@]}

# Print the number of elements
echo ${#array[*]}

# Print a list of indices
echo ${!array[@]}
```

Some common operations for arrays are listed in Table 1.2.

There are `for`-, `while`-, and `until`-loops in bash. Since we typically just use `for`-loop, we only introduce it here, and for the other two you can refer to other related resources.

Recall in Python, we have index-based and element-based loops. This is also in bash scripts. The following example shows us different types of `for`-loops:

Listing 1.7: Iterating over array elements.

```bash
#!/bin/bash
# array_loop.sh
array=(1 2 "three" 4 "five")

# Iterating over indices
for i in ${!array[@]}; do
    echo "[${i}]: ${array[$i]}"
done

# Iterating over elements
for elem in ${array[@]}; do
    echo ${elem}
done
```

```
14
15 # Iterating for a specific number
16 for i in {0..10}; do
17     echo "[${i}]: ${array[$i]}"
18 done
```

Note that when we use `for i in {a..b}`, the iterator variable `i` will go from `a` to exactly `b` (not $b-1$!). The two dots between `a` and `b` also need to be like that. This is called **expansion**.

One small fun thing: you can also put letters in the loop. Try the following, and see if you know what's happening here:

```
1 echo {a..z}
2 echo {a..Z}
3 echo {A..z}
4 echo {A..0}
```

1.4.4 Functions

Functions are written like this:

```
1 function f_name {
2     # Function body
3 }
```

Notice that we don't have an argument list here. When calling a function, we just put all the arguments after the function name, and in the function body, we use `$1`, `$2`,... to refer to them. This could be confusing, so writing good comments (including which parameter is for what, *etc*) is very important.

The following example declares a global variable `height`, and a function called `volume`. The function takes two parameters, referred as `$1` and `$2` inside the body:

```
1  #!/bin/bash
2  # function.sh
3  height=3
4
5  function volume {
6      # vol is a local variable
7      vol=$(( $height * $1 * $2 ))
8      echo $vol
9  }
10 volume 10 20
11 result=$(volume 10 20)
12 echo "Result: $result"
```

Listing 1.8: Creating functions in bash.

Notice here to call the function, we simply write the function name first, and put all parameters after it, separated by spaces. We can just invoke the function as in line 10, or put the function name and its parameters into an evaluation `$(...)` as in line 11. This will be evaluated to the function's return value.

1.4.5 Redirecting Standard Input and Here Documents

1.4.5.1 Standard Input

As you can expect, if redirecting output is `>`, redirecting input is then `<`. The operator `<` is to redirect keyboard input to a file. Thus, to use this operator, we have to make sure the command is expecting an input from `stdin`: not command-line arguments, not files.

Let's look at the following example:

```
1 #!/bin/bash
2 # read.sh
3 read input
4 echo "Result: ${input}"
```

Here we use a command called `read`, which upon execution, will wait for our keyboard input. Whatever we typed on the keyboard with an enter key will be stored into the variable `input`.

Now, if we store our input in a file in advance, say `random.txt`, we can invoke the script with redirection:

```
1 $ ./read.sh  <  random.txt
```

This time the script doesn't wait for out input, because `stdin` has been redirected to the file we specified.

1.4.5.2 Here Documents

What about `<<`? Unlike `>>` which is to append output, the operator `<<` is to create a Here document, or **heredoc**. This operator should follow a command that's expecting a file input.

Imagine you're trying to test `cat` command. Normally, you should pass a file name to the command, so it can print out the file content, such as `cat random.txt`. Now, if you don't have that file, you don't really need to create one; you can just use `<<` to create a temporary file. The syntax of writing `heredoc` takes the following form:

```
1 COMMAND << DELIMITER
2 HERE-DOCUMENT
3 DELIMITER
```

The `DELIMITER` can be any string you like, usually we use `EOF` or `END`.
The `HERE-DOCUMENT` contains multiple lines that'll be fed into the `COMMAND`.

Let's try the following on terminal:

```
$ cat << EOF
```

Once we hit enter key, we'll see something like this:

```
heredoc>
```

This looks just like another shell waiting for our input, but it's the interface to write a `heredoc`. Then we can write the content of a file here. Once finished, type the same delimiter to exit `heredoc` interface, and we'll see what we wrote has been printed out from the `cat` command.

What this implies is that sometimes we don't really need to create an actual file for commands that's expecting a file input; instead we can temporarily create a file right here in the terminal.

Here's another useful feature of using `heredoc` in a bash script. Let's see the following example.

Listing 1.9: An example of `heredoc`.

```
#!/bin/bash
# here.sh

cat << EOF
The current working directory is: $(pwd)
You are logged in as: $(whoami)
EOF
```

You can see that the commands `whoami` and `pwd` are also evaluated before the `heredoc` is fed into `cat`. If we are just using `cat` with regular file, this will not happen.

This is a very useful feature when we want to print out some help messages:

Listing 1.10: Another example of using `heredoc`.

```
#!/bin/bash
# here2.sh
echo "Type a path: "
read path
if [[ $path = "help" ]]; then
    cat << EOF
The current working directory is: $(pwd)
All you need to do is to type it!
EOF
elif [[ $path = $(pwd) ]]; then
    echo "Good job!"
else
    echo "Type help next time!"
fi
```

1.4.6 User Interface

1.4.6.1 Positional Arguments

Positional arguments refer to those that don't have any flags. For example, if the script name is `test.sh`, you can type

```
$ ./test.sh hello world whats up
```

The four strings, **separated by spaces**, are the positional arguments. You can catch them using a dollar sign and the position, such as:

```
echo $1  # Prints out "hello". Notice it starts from 1.
echo $4  # Prints out "up"
```

Or you can use a loop to catch them all:

```
echo "There are $# arguments:"
for word in "$@"; do
    echo "$word";
done
```

Note that we use `$#` to get the number of arguments. This number **does not include the command** (which is different than C or C++). Also, the arguments are numbered from 1.

1.4.6.2 Arguments with Flags

You probably noticed that when we use some programs, we use some options or flags: we add `-l` to `ls` to show files in a long format; we add `+x` to `chmod` to make the file executable, *etc.* When we write our own bash scripts, we can also use a built-in utility called `getopts` to process the options passed by users.

`getopts` parses short options, which are a single dash (`-`) and a letter or digit. Examples of short options are `-2`, `-d`, and `-D`. It can also parse short options in combination, for instance `-2dD`.

However, `getopts` cannot parse options with long names. If you want options like `--verbose` or `--help`, use `getopt` instead (notice it doesn't have an `s`!).

1.4.6.3 Using `getopts`

There could be multiple options passed to the script, and `getopts` can process one option at a time, therefore we usually use `getopts` in a loop, and the loop will end when there's no more options to process from the terminal. The general structure is like this:

```
while getopts "<option-list>" <option-variable>; do
    case "${<option-variable>}" in
        <option-1>)
        # Process for option 1
        ;;
        <option-2>)
        # Process for option 2
        ;;
        *)
        # If none of the above
        ;;
    esac
done
```

Let's look at some elements in the structure above.

- `<option-list>`: In `getopts`, each flag is a single letter. This list includes all the acceptable letters. For example, if our program only accepts flags `-n`, `-p`, and `-i`, the list should be `"npi"`;
  - Some flags need to receive an argument. For example, we want to pass `josh` as the argument of flag `-n`, *i.e.*, `./demo.sh -n josh`. In that case, we need to add a colon after that letter, so the list becomes `"n:pi"`;
  - If we specify that flag `-n` needs to receive an argument but we forgot to pass one, or we passed an unacceptable flag, the bash script will automatically print an error message, *e.g.*,

    ```
    $ ./demo.sh -a
    ./demo.sh: option requires an argument -- a

    $ ./demo.sh -k
    ./demo.sh: illegal option -- k
    ```

    If we don't want to see this message but rather using our own, we can silence the automatic message by adding a colon to the front of the list: `":a:pi"`;

- `<option-variable>`: This is the variable that represents each flag received. `getopts` checks each flag in a `while`-loop, and use the nested `case` structure to process each flag;

- `<option-x>`: You should put one of the letter from `<option-list>` here, and write how to process this flag. The double `;;` is similar to `break`, so you should have it at the end of every case;
  - If a flag requires an argument, we can use `${OPTARG}` to retrieve the argument. Note that `OPTARG` is a fixed name, so it can't be changed;
  - When using `*` as an option, it means a "wild card" – any flag that doesn't fall into the `<option-list>` or missing a required argument will fall into this case.

Let's look at the following example:

Listing 1.11: Using `getopts` to receive option list from terminal.

```
#!/bin/bash
# demo_getopts.sh

print=0
name=""
repeat=1

while getopts "n:pi:" options; do
case "${options}" in
    n)  name=${OPTARG}
        ;;
    p)  print=1
        ;;
    i)  repeat=${OPTARG}
        ;;
    *)  echo "Oops" >&2
        exit 1
esac
done

if [[ $name = "" || $print -eq 0 || $repeat -lt 1  ]]; then
    echo "Bad arguments." >&2
    exit 1
fi

msg=""
if [[ print -eq 1 ]]; then
    msg="omg ${name} h"
fi

for i in $(seq $repeat); do
    msg="${msg}i"
done

echo $msg
```

The first thing that needs to mention is how we structure our program. In this example, the string we want to print out, `msg`, is formatted based on the value of the arguments. Notice instead of formatting the string right inside the `getopts` cases, we set variables instead, and deal with them all at once **after** the flags have been processed. This is the recommended way when using `getopts` in your program.

Second, notice when there's something wrong, we print the error message through `stderr`. This is a good practice to separate regular printing messages from error messages.

Last, we use `exit 1` to exit the program with errors. This is also a typical practice. The number `1` here is returned to the system to notify that something went wrong with our program. When everything's normal, the script automatically exits with number `0`.

C Programming Language 2

2.1 Introduction

To use some of the functions in C, we usually use the following syntax to include some standard libraries. It starts with `#include`, and the name of the library surrounded by a pair of <>. The most frequently used library is called `<stdio.h>`, which will help us print things to console, receive input from a keyboard to console, and more.

```c
#include <stdio.h>
```

The `main()` function is the main entrance of any C program. Its name has to be called `main`. All functions in C are within a pair of curly brackets.

The word `int` at the start of function indicates that the last line of the code within curly brackets will return an integer value. This is carried out using the statement `return 0;`. Returning the value zero indicates that the program has reached the end without encountering any problems.

Also notice that every statement in C ends with `;` a semi-colon.

```c
int main (int argc, char* argv []) {
    return 0;
}
```

Comments are very important in any programming languages. In C, we put all our comments in a pair of `/* */`:

```c
/* Or you can start from here
And go on
And go on
Till the end of the world
But remember to close the comment! */
```

In newer standards of C, we can certainly use `//` for single line comments.

2.1.1 Data Types

2.1.1.1 Primitive Types

To define a variable, you need to indicate the data type, give the variable a name, and optionally initialize it with some appropriate values. They can be declared using the following keywords:

- Integers: `int`
- Float points: `float`
- Double precision float points: `double`
- A single character: `char`

```c
int    radius =   10;
float  pi     =   3.14;
double ex     =   2.7;
char   c      =   'a';
```

Note that `char` represents a single character. A single `char` has to be paired with single-quotes, `''`. This is different from Python.

Also notice that C doesn't have a native boolean type, so by convention we use `int` for that purpose, and take `1` as true, and `0` as false. [1]

1: In fact, in C, number 0 is defined as false, while all non-zero numbers are treated as true.

In addition to those types we're quite familiar with, there's also one data type we see a lot in loops, which is `size_t`. This type is simply an unsigned integer, and is commonly used as array indices to avoid negative values.

There are also other simpler types. For example, `uint_32` represents a 32-bit unsigned integer. We'll introduce them later when necessary.

2.1.1.2 Arrays

Arrays are fixed size, and occupy a *contiguous* area of memory. When you declare them, you need to specify the data types of each element in this array, as well as the dimensions of the array. The following example declares a one-dimensional array of `double`, and a two-dimensional array of `int`. Higher dimensions are also possible.

```c
double matrix_1[12];
int    matrix_2[3][3];
```

You can assign values to the arrays when declaring them, using curly brackets:

```c
int vector[3] = {1, 10, 100};
```

In this case, you don't have to specify the dimension; the compiler will decide the size based on the initial values you assigned:

```c
/* The compiler will decide vector_2 has 5 values. */
int vector_2[] = {1, 10, 100, 1000, 10000};
```

You should be very clear by now that in computer science, indices always start from 0, instead of 1. To get the element at `i`-th row and `j`-th column, we can use:

```
matrix_2[0][2];
```

2.1.1.3 Character Arrays (aka Strings)

C doesn't have a type for strings, but can be easily (or not...) managed by creating `char` arrays, because essentially a string is just an array of characters.

To declare a C string, there are two typical ways:

```
char str[] = {'h', 'e', 'l', 'l', 'o', 0};
char str[] = "Hello";
```

Note that we don't have to specify the length of the array, because the compiler calculates it based on our initialization. All C strings are NULL-terminated, which means the last character of the string needs to be a `NULL`, whose ASCII value is zero. That's why you see a 0 at the end of the first line.

Some handy functions for strings are included in `<string.h>`. For example, `strlen()` calculates the length of the array:

```
char str[] = "Hello";
printf("%d\n", strlen(str));
printf("%d\n", strlen("Hello!"));
```

Note that `strlen()` doesn't count the null terminator.

Another example, `strcmp()` compares if two strings are the same; if it's the same, it returns 0; otherwise a non-zero value:

```
char str1[] = "Hello";
char str2[] = "hello";
if (strcmp(str1, str2) == 0) printf("Same string!\n");
else printf("Different string!\n");
```

2.1.1.4 C `struct`

The data types mentioned above are primitive data types. It is quite common that we sometimes want to group related primitive data together as a whole. For example, a rectangle has its width and length. Then "rectangle" here refers to a type of special composite data, and all the data it contains are of the primitive data types provided by C, such as `float`.

In C, we implement this as a `struct`:

```c
struct Rectangle {
    float width;
    float length;
};
```

Then we can create a variable or **object** of this struct:

```c
struct Rectangle box1;
struct Rectangle box2;
```

To use a data member in an object, we can use "`.`" operator:

```c
box1.width   =  10;
box1.length  =  20;
box2.width   =  13.5;
box2.length  =  33;
```

If we are using pointers to a struct, we need to use "`->`" operator:

```c
/* Assume p_box1 is of type struct Rectangle* */
p_box1->width = 10;
```

It's sometimes annoying to write `struct` all the time when defining a variable, so we sometimes use the keyword `typedef`:

```c
typedef struct {
    float width;
    float length;
} Rectangle;
```

Then when declaring a variable, we can simply write:

```c
Rectangle anotherone;
```

The keyword `typedef` can be viewed as giving an alias to a data type, so we can use this alias later. The following is only an example; don't do it!

```c
typedef int integer;
integer num = 10;
```

What's different than classes in C++ is, all the members in structs are public, and we cannot declare any functions inside it. It is, however, to declare function pointers, which will be discussed in depth in Section 2.4.4.

2.1.2 Functions

An important part of C functions, different than Python, is some compilers require the **declaration** of a function before the definition. [2]

2: A declaration simply has the header of a function, while a definition has all the code/implementation of a function. In other words, declaration tells the compiler how to *call* the function, while definition tells the compiler how to *execute*.

The declaration of a function looks almost the same as function definitions, which has return type, function name, and arguments, but it doesn't contain code to be executed, and it ends with a semi-colon:

```
double get_volume (double length, double width, double
↪ height);
```

In the function declaration, the names of the arguments are optional, and it doesn't matter if you have different names than function declarations. The C compiler only needs to match the argument data types:

```
double get_volume (double, double, double);
```

Once we've written the declaration of the function, we can implement it in our program. The function body needs to be surrounded by curly brackets. Note that now we have to give arguments names:

```
double get_volume (double l, double w, double h) {
    double vol = l * w * h;
    return vol;
}
```

The argument names don't have to be the same as in the declaration. Remember, the only thing that matters between prototype and the implementation is the matching of data types of arguments.

If a function doesn't need to return any value, you can set it to `void`. At the end of the function, simply write `return;` :

```
void doing_nothing() {
    return; /* This is optional. */
}
```

2.1.3 Command-line Arguments

Flags are good for compiling. Sometimes, we want to pass something to our program during run time. For example, we want to receive two strings, representing the user's first name and last name, and we want to print out a message saying `Welcome, <first> <last>!`. Apparently, when writing the program, we don't know what they are, and they can only be determined when actually running the program.

We can achieve this by using command-line arguments. Remember the main function looks like this:

```
int main (int argc, char* argv[])
```

When we run our executable (say it has a name of `a.out`), shell actually passes the executable file `a.out` as a `char` array (6 element, why?) to the

`argv` array as the first element. Since we didn't pass anything else, `argv` has only one element, which is the string `a.out`. The integer `argc` is then set to be the length of the `argv` array, which in this example, is 1.

Try the following:

```c
#include <stdio.h>
int main (int argc, char* argv[]){
    printf("%d\n", argc);
    printf("%s\n", argv[0]);
    return 0;
}
```

We can also add more command-line arguments based on our need, but remember:

- The first element in `argv` is always the name of our executable file; and
- Every element in `argv` is an array of `char`, even if we type some digits.

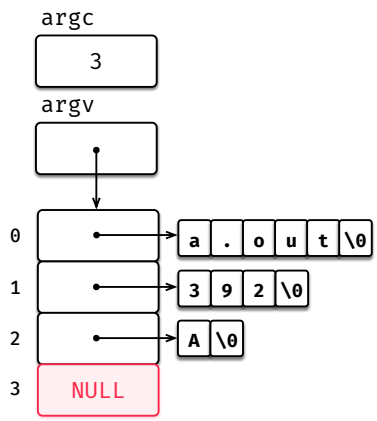


Figure 2.1: Storage of command-line arguments of C programs.

Figure 2.1 shows the structure of how command-line arguments of C programs are stored in memory. Note that in addition to using `argc` to loop over all arguments, we can also check if the argument is `NULL` or not to terminate the loop.

2.1.4 Return or Exit?

Later in many of our demo code, you will see we use `exit()` a lot instead of `return`. The declaration of the function is as follows:

```c
void exit(int status);
```

When we call this function, the process will exit with an exit code status. C standard library `<stdlib.h>` has defined two macros, `EXIT_SUCCESS` and `EXIT_FAILURE` to indicate successful and unsuccessful termination. You can use the following command to check the exit code status of a previously executed program:

```
$ ./a.out
$ echo $?
```

What's the difference of using `return 0` and `exit(0)`? The most obvious one is, `exit(0)` will terminate the entire program, regardless of where it's used, while `return 0` will return back to the caller function.

In terms the difference when used in `main()`, Figure 2.2 shows us how C programs started and terminated. When we use `return 0` in `main()`, we didn't terminate the program immediately (even though from our end

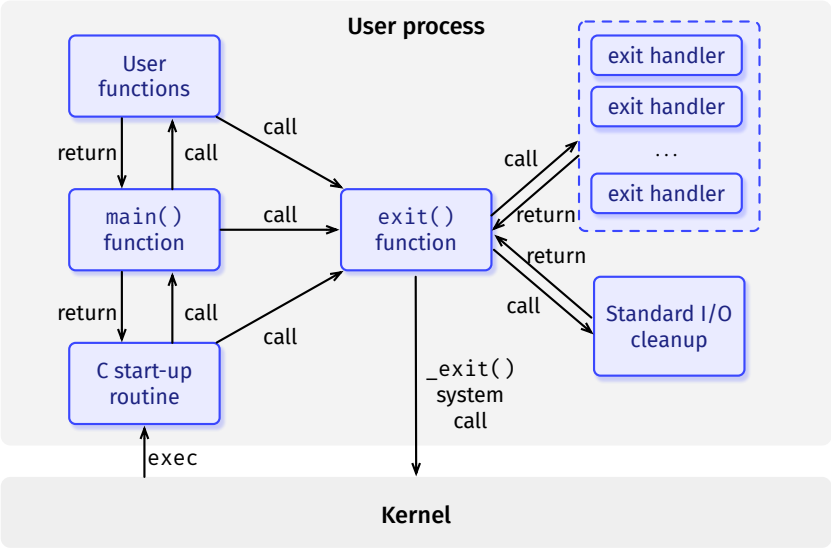


Figure 2.2: Diagram of executing and terminating a C program.

it looks like the program is terminated right away); instead, it returns to the C start-up routine, which calls `exit()` function on our behalf. But certainly we can call `exit()` function in `main()` by ourselves too.

On the other hand, when `exit()` function is called, it first calls a couple of "exit handler", and then standard I/O cleanup routine that cleans up buffer. Once all the things are done, it calls `_exit()` which is a *system call* (that never returns), and exits the user process entirely. [3]

3: Of course we can write our own exit handlers. If you're interested, look for `atexit()` function.

2.2 C Standard I/O Library

So far the most common function we have used in C is probably `printf()` (well, other than `main()`, obviously). This function is formatted output, declared in the C standard I/O library `<stdio.h>`. This library defines all the necessary functions to process input and output, including interacting with files, terminal, keyboard, strings, and so on. This website (https://www.ibm.com/docs/en/zos/2.3.0?topic=files-stdioh-standard-input-output) lists all the functions defined. We are not going to talk about all of them, but pick some important ones. In this section we mainly focus on file input/output.

2.2.1 Streams & Standard I/O Streams

C handles files using a `struct` pointer `FILE*`, which is usually called **stream**. It's actually not something new: you must have heard of standard input/output. They are exactly streams. Every UNIX or Linux system has three file streams opened upon booting automatically, and they're attached to the terminal: `stdin`, `stdout`, and `stderr`. In C, they're declared as follows:

```c
#include <stdio.h>
extern FILE* stdin;
extern FILE* stdout;
extern FILE* stderr;
```

`stdin` is for receiving user's input, `stdout` for printing output, and `stderr` for printing diagnostic or error messages. [4]

4: Error messages and regular messages are separated through standard error and standard output because we don't want to mix error log with normal output.

To read from and write to a file stream, the functions we usually use are:

```c
int fprintf ( FILE* stream, const char* format, ... );
int fscanf  ( FILE* stream, const char* format, ... );
```

You can see they look just like the functions we are familiar with— `printf()` and `scanf()`, but with an `f` prefix in the function names, and `FILE*` as the first argument.

Previously when we wrote programs, we printed everything—including usual messages and error messages—out to terminal using `printf()`. In systems programming that is actually a bad practice. From now on, please print all error messages through `stderr`. For example,

```c
if (argc != 2) {
    fprintf(stderr, "Usage: %s <file>\n", argv[0]);
    exit(EXIT_FAILURE);
}
```

where we notice the first parameter of `fprintf()` is `stderr` instead of `stdout`.

You probably already realized that the following two statements are equivalent:

```c
printf("Hello!\n");
fprintf(stdout, "Hello!\n");
```

The `printf()` function's default destination is `stdout`. The same applies to `scanf()`:

```c
scanf("%d", &num);
fscanf(stdin, "%d", &num);
```

2.2.2 Opening & Closing a Stream

Other than the three system-defined file streams, we can certainly use `FILE*` to open text files and to read/write. In the following example, we use `fopen()` to open a file, and `fclose()` to close the file. The first

argument of `fopen()` is the path to the file as a string, and the second is the open mode (`w+` for read/write with override)

```c
#include <stdio.h>
FILE* fp = fopen("path/to/the/file", "w+");
if (fp == NULL) {
    fprintf(stderr, "Error in opening file!\n");
    exit(EXIT_FAILURE);
}

/* Do file processing */

fclose(fp);
```

One thing to mention about the path is it can be either absolute or relative path. If it's a relative path, it is relative to the executable file, *i.e.*, the location where the program is being executed.

Upon successful opening, `fopen()` will return a pointer to `FILE*`, which is used for file I/O. We have to consider the case where the file cannot be opened successfully to prevent the program from crashing, so you should check file stream after calling `fopen()` all the time.

2.2.3 Reading Lines from Files

One common task is to read all the content from a file, and the function we want to use is `getline()`. The following code listing is modified from manpage. [5]

5: https://man7.org/linux/man-pages/man3/getline.3.html.

Listing 2.1: Reading lines from a file using `FILE*`.

```c
/*** readline.c ***/
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    FILE*   stream;
    char*   line   = NULL;
    size_t  len    = 0;
    ssize_t nread  = 0;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if ((stream = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "Error in opening file!\n");
        exit(EXIT_FAILURE);
    }

    while ((nread = getline(&line, &len, stream)) != -1) {
        printf("Retrieved line of length %zd:\n", nread);
```

```
23      }
24
25      free(line);
26      fclose(stream);
27      exit(EXIT_SUCCESS);
28  }
```

The description of `getline()` on manpage is quite precise and clear, so we will avoid repetition here.

> *From manpage, https://man7.org/linux/man-pages/man2/read.2.html.*
>
> ---
>
> The types `size_t` and `ssize_t` are, respectively, unsigned and signed integer data types specified by POSIX.1.

2.3 Error Handling

In GNU C library, a global variable called `errno` is declared in `<errno.h>`. Whenever a library function goes wrong and returns -1, it also sets a value to `errno`. If a function runs correctly, it doesn't change the previous `errno`. The values of `errno` are declared as macros, and you can find them here: https://www.gnu.org/software/libc/manual/html_node/Error-Codes.html.

Inside the system, each error code macro is also associated with a text message describing the error. When possible, we should use system-defined error messages instead of writing our own. Let's look back at Listing 2.1, specifically lines 16 – 19 where we check if the file can be opened:

```
1  if ((stream = fopen(argv[1], "r")) == NULL) {
2      fprintf(stderr, "Error in opening file!\n");
3      exit(EXIT_FAILURE);
4  }
```

In fact, file I/O error is so common there's already a macro defined. When `fopen()` failed to open a file in the case that it doesn't exist, in addition to returning a `NULL` pointer, it also sets `errno` to `ENOENT`. Therefore, instead of writing our own error message, we should use `perror()` function to print out the standard error message:

```
1  if ((stream = fopen(argv[1], "r")) == NULL) {
2      perror("fopen");
3      exit(EXIT_FAILURE);
4  }
```

The output will look like this:

```
fopen: No such file or directory
```

where `"No such file or directory"` is the standard error message associated with error code of `ENOENT`.

2.4 Pointers

At this point you should already know that every variable in memory has an address. To get a variable's virtual memory address, we can use *address-of* operator, "`&`". The following code defines an integer variable `a`, and prints out the address of it:

```c
#include <stdio.h>
int main () {
    int a = 0;
    printf("%p\n", &a);
    return 0;
}
```

The address that's printed out can be different every time we run the program.

This applies to arrays as well. Say we have an array of float numbers. Let's take a look at the following code:

```c
/*** arrayaddr.c ***/
#include <stdio.h>
int main () {
    float array[10] = {0.0};
    printf("%p\n", array);
    return 0;
}
```

Listing 2.2: For an array, its name is already the address of its first element.

Something different here is that the value of the array's identifier is the *address* of the first element. Thus, when you run the code above, it doesn't print out `0`; it prints out an address instead.

Most of the time, we'd also like to store an address in another variable so that we can use later. To do this, we need to declare *pointers*:

```c
int  a   = 0;
int* ptr = &a;
```

In this example, `ptr` is called a pointer, and its value is the address of variable `a`. To declare a pointer, we need to specify the data type of the variable stored in the address, or the data type of the variable it points to. Variable `a` is an integer, so to declare a pointer that's pointing to it, the type of the pointer should be `int*`. The star `*` indicates that this is a pointer instead of a normal variable.

We can also use pointers in arrays, like this:

```
float  list[10];
float* ptr = list;
```

We didn't use the address-of operator in the second line. Why?

The star can also be used as a **dereference** operator. In the example above, `ptr` stores the address of variable `a`. Now that we have this address, we can certainly print out the value at that location:

```
printf("%d\n", *ptr);
```

This time, `*` means we want to get the *value*, instead of the address of the pointer `ptr`. Note that to dereference, `*` has to be followed by a pointer, not a variable. All the computations are still valid for dereferenced pointers, as in the original variables. For example, say we have an integer variable `a`, and a pointer `ptr` pointing to it. If we want to change the value of `a` to 10, we can operate directly on the variable, *i.e.*, `a = 10`, or use the pointer: `*ptr = 10`.

Quick Check 2.1

```
int  first  = 10;
int  second = first;
int* third  = &first;
printf("first: %d, second: %d, third: %d\n",
        first, second, *third);
second   ++;
(*third) ++;
printf("first: %d, second: %d, third: %d\n",
        first, second, *third);
```

Please describe what you observe from the output, and why it is happening. *Hint:* analyze this from the perspective of memory.

Things get interesting when we're dealing with arrays. Say we have an integer array, and we want to print out all the elements in this array. The following code is what we've been using:

```
int vector[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i ++) {
    printf("%d\n", vector[i]);
}
```

Recall that the identifier of arrays has the address of the first element. In this example, the value of `vector` is the address of `vector[0]`. Thus, when we use `vector[i]` to print out the value, we can think this as shifting from the beginning of `vector` for `i` locations, and get the value

there. Note that this operation is possible because arrays occupy a *consecutive* space in memory, which means the elements in arrays are staying next to each other.

Now that we know `vector` is the address of the first element, we can also print out `vector[0]` like this:

```
printf("%d\n", *vector);
```

If we want to print out other elements as well, we need to know their physical addresses, and then dereference them.

```
int vector[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i ++) {
    printf("%d\n", *(vector + i));
}
```

The operation `vector + i` is to shift the address of `vector` by `i` elements, and then the `*` dereferences the address and get the value stored there.

In the example above, we put parenthesis around `vector + i` to shift the address first, and then dereference it using `*`. If we take off the parenthesis, it'll be another story:

```
int vector[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i ++) printf( "%d\n", *vector + i );
```

Instead of printing out the five elements in the array, the program prints out `10 11 12 13 14`. This is because dereference operator `*` has a higher precedence than addition operator `+`. Thus, without parenthesis, C operates on `*vector` first to get the value at the address, which is `10`, and then add the variable `i`. The operator precedence table can be found at: https://en.cppreference.com/w/c/language/operator_precedence.

2.4.1 Pass by Value

Sometimes we want to change some variables in a function. A straightforward way to do this is to pass that variable to the function as an argument.

```
/*** passvalue.c ***/
#include <stdio.h>

void doubling(int x, int y) {
    x = x * 2;
    y = y * 2;
}

```

Listing 2.3: C is a pass-by-value language, therefore to change variables passed to a function, we need to pass pointers.

```c
int main () {
    int x = 2, y = 10;
    printf("(%d,%d)\n", x, y);
    doubling(x, y);
    printf("(%d,%d)\n", x, y);
}
```

In the listing above, we created a function `doubling()` trying to double the two numbers, but after calling the function, the values of `x` and `y` didn't change at all. This is because when passing them to the function, we are not actually handing out the actual two variables; instead, we are only copying the values to the parameters.

Remember that each function has its own space in memory, where it stores its local variables. Once the function is finished, the space will be freed, and all local variables will be destroyed.

In the code above, when we are calling the function `doubling()`, C creates two local variables called `x` and `y` in the space of `doubling`, and then copy the values of `x` and `y` in the `main()` function. Whatever we did in `doubling()` stays in `doubling()`, and it doesn't affect the values in `main()` function. This is called **passing by value**, and is applied to all data types. Remember, C language is always passing by value.

To actually change the value in a function, we have to pass the *address* of that value, so that we can operate on the exact variables even in another function.

⚠ Caution!

You probably have learned that in C++, we can simply modify the parameters of `doubling()` to `int&`, as in `void doubling(int& x, int& y)`. However, using `&` in function headers in C is illegal.

Recall that if we have the address, we can just dereference and operate on it. Then let's pass address, or pointers:

```c
void doubling(int* x, int* y) {
    *x = (*x) * 2;
    *y = (*y) * 2;
}
```

The `*`'s in the function header means we're passing an address or a pointer, while `*`'s before `x` or `y` are dereferencing operators. And of course, the `*` in `* 2` is multiplication operator.

The problem is, now that we're passing pointers not values, `doubling (x,y)` would be a wrong call as well. Thus, we need to pass the addresses using `&`, the address-of operator:

```c
doubling (&x, &y);
```

The complete code is in `passaddr.c`.

We said that C language is a pass-by-value language. This applies to pointers too. Let's take a look:

```c
/*** changeptr.c ***/
#include <stdio.h>
void increment_ptr (int* p) { p ++; }
int main () {
    int vector[5] = {10,20,30,40,50};
    int* ptr = vector;
    printf("%d\n", *ptr);
    increment_ptr(ptr);
    printf("%d\n", *ptr);
    return 0;
}
```

Listing 2.4: Pointers are also variables. If we want to change the value of a pointer in a function, we need to pass the pointer of that pointer.

The output of two `10`'s indicates that the value of `p` is changed only in the function `increment_ptr()`. So how to deal with this? (No illegal C++ stuff here!) Let's face it: `int*` is also a data type. Then the same idea applies here: we'll just pass the address of this variable:

```c
/*** changeptr2.c ***/
#include <stdio.h>
void increment_ptr (int** p) { (*p) ++; }
int main () {
    int vector[5] = {10,20,30,40,50};
    int* ptr = vector;
    printf("%d\n", *ptr);
    increment_ptr(&ptr);
    printf("%d\n", *ptr);
    return 0;
}
```

Listing 2.5: Correct version of Listing 2.4

Here, the data type of the parameter is `int*`. Thus, if we want to pass the address of this parameter, we need another `*`. When we dereference it using `*p` (why do we need a parenthesis here?), it gives us the address of `ptr`.

Now let's extend `doubling()` function, so that we can double the value of all elements in an integer array:

```c
void doubling(int* array, int size) {
    for (int i = 0; i < size; i ++) array[i] = array[i] * 2;
}

int main () {
    int vector[5] = {10, 20, 30, 40, 50};
    doubling(vector, 5);
}
```

In the code above, notice that we didn't pass `&vector`, because remember as the identifier of an array, `vector` is already a pointer, or an address. In `doubling()` function, we also didn't explicitly dereference `array` using

`*`; we use subscriptions `[i]` directly.

Given a function declaration like this:

```
void doubling(char** array) {
    /* some code here */
}
```

we can actually pass many types of things as `array`, but of course, the code inside this function should be different based on what kind of things we passed to it.

First, we can pass the pointer of a pointer, *i.e.*, passing `int**` to `int**`.

```
char   sym     =   'C';
char*  p_sym   =   &sym;
char** pp_sym  =   &p_sym;
doubling (pp_sym);
```

We can also pass an array of pointers:

```
char  a = 'A', b = 'B', c = 'C';
char* vector[] = {&a, &b, &c};
doubling (vector);
```

Lastly, we can pass a two-dimensional `char` array to it, which is essentially an array of strings.

2.4.2 `NULL` Pointers

One special type of pointer is null pointers:

```
int* ptr  =  NULL;
```

Regardless of types, any pointer can be a null pointer. This simply means a null pointer points to nothing in memory. A null pointer also has a value of 0.

2.4.3 `void` Pointers

One important and special type of pointer in systems programming is `void*`, the void pointers. Like any other type of pointers, a void pointer also stores an address; unlike any other type of pointer, a void pointer **cannot** be dereferenced. Why? Remember if we want to dereference a variable from a pointer, we need to know how many bytes we're going to take starting from the address pointed by that pointer. And that's why pointers have data types such as `int*` or `char*`. However, what type it is for `void*`? How do you know how many bytes you're going to take? Therefore, something like this is nonsense:

```c
void* a;
printf("%d\n", *a);
```

Then why do we need void pointers? Because it's extremely versatile and thus useful in designing system function interfaces. One example you will see is `malloc()` that'll be introduced in Section 2.5.2.

Let's start with a small, simple, and silly example (that's my "3S rule"!). Say we are designing a system function that any user can use, called `last_elem()`. This function will return the address of the last element of the array passed to it.

Now we have a problem. This function should be able to take any type of arrays: integer, float, char, or even user-defined `struct` arrays. Should the declaration be `int* last_elem(int*)`? Or `float* last_elem(float*)`? We have no idea what type of array the user will pass to the function, so we can't write any type of specific pointers as the return value and arguments.

Here's how void pointer comes in handy. We can declare our function like this:

```c
void* last_elem(void* arr, int elem_size, int arr_len);
```

You see both return value and the first argument are of `void*` type, which can accommodate to any type of pointers and arrays. Even if `void*` is magical, it cannot decide the array length and type, so we have to pass some information. In this task, the useful information would be how many bytes an element takes, and how many elements there are in the array. Therefore, we added two integer arguments, `elem_size` and `arr_len`.

Inside `last_elem()`, we take `arr` as the starting address of the array, so the address of its last element should be straightforward: [6]

```c
void* last_elem(void* arr, int elem_size, int arr_len) {
    return arr + elem_size * (arr_len - 1);
}
```

Now let's change our role to system function designer to a user, and we'd like to use this function on one integer array and one double array:

```c
/*** void.c ***/
#include <stdio.h>
void* last_elem(void* arr, int elem_size, int arr_len) {
    return arr + elem_size * (arr_len - 1);
}

int main(int argc, char* argv[]) {

    int  intarr[4]  = {10,20,30,40};
```

6: Notice here, even if `arr` is `void*`, we can still do pointer arithmetics. If we have `arr + x` where x is an integer, it'll shift the address x-bytes from `arr`.

Listing 2.6: An example of using `void*` for system function design.

```
    int* last_int_p = (int*)last_elem(intarr,
                                      sizeof(int),
                                      4);
    printf("The last element is: %d\n", *last_int_p);

    double  doublearr[3]  = {1.23, 4.56, 7.89};
    double* last_double_p = (double*)last_elem(doublearr,
                                        sizeof(double),
                                        3);
    printf("The last element is: %lf\n", *last_double_p);

    return 0;
}
```

One thing you must have noticed is we cast the return of `last_elem()` back to a specific type of pointer. This is very natural, because in the code we want to dereference the pointer and print the value out, but `void*` cannot be dereferenced until it's cast to a type.

This simple example shows use why C language has a `void*` which is seemingly unnecessary.

2.4.4 Function Pointers

In virtual memory, functions are not that different than variables. Remember we talked about how memory is organized: local variables are stored in the stack area, while dynamically allocated variables are in the heap area. There's also an area where it stores the code. All our functions are translated into machine language, and stored into that area. Since it's in the memory, of course they also have addresses. The address of a function is the address of the function's first machine instruction.

Note that as in variable pointers, function pointers are associated with the prototype or declaration of the function they want to point to. Say we have a function like the following:

```
int addition (int x, int y) { return x + y; }
```

then to define a pointer to this function, we can write:

```
int (*func_ptr)(int, int);
```

where `func_ptr` is the pointer's name, and we add `*` and parenthesis around it to show that this is a function pointer. This pointer can be used to point to any function who receives two integers as parameters, and returns an integer. [7]

7: Why do we have to put parenthesis around `*func_ptr`, and what would happen if we don't?

Similarly, we can let the pointer point to `addition` function:

```
func_ptr = addition;
printf("%p\n", func_ptr);
```

The first line lets `func_ptr` point to the `addition()` function, and the second line prints out the value of `func_ptr` which is also the address of the first instruction in `addition()` function.

It's also easy to use function pointers. Continuing with the last example, we just need to dereference the function name, and pass arguments to it:

```
int result = (*func_ptr)(4, 5);
int result = func_ptr(4,5);     // This also works
```

and we will have `result == 9`.

A big question is, why do we need function pointers? We have to think about the answer from the perspective of a system designer. As a system designer, we want to provide good programming interfaces to some core utilities, usually a function. A typical example is a quick sort function provided by C library called `qsort()`:

```
#include <stdlib.h>
void qsort(void* base, size_t nmemb, size_t size,
           int (*compar)(const void*, const void*));
```

For sorting algorithms, we know that we need an array. However, when we are developing this function, we have no idea what users will sort. Is it an integer array? Or a double array? Or even an array of a user self-defined struct? We have no idea. Therefore, we have to require the first three arguments: a base address `base`, the number of elements in this array `nmemb`, and the size of each element `size`.

There's another problem – because we have no idea what kind of array the user will use `qsort()` to sort, it is also not known what kind of criteria the user will use. Does the user want to sort it ascendingly? Descendingly? If it's an array of a user self-defined struct, we can't sort the objects, and we have to know which member variable in the struct will be used for sorting.

This is where function pointer comes in handy. Remember all sorting algorithms comes down to the comparison of two elements in the array. Instead of guessing, we let the users pass a function pointer `compar`. All they need to do is to define how they want to compare two elements in their array by creating a function, and pass the pointer to the `qsort()` function. During the iteration, whenever we need to compare two elements, we just call the function passed by the user, and make judgment.

Now let's change our role to a user and take a look at the following example. Say we have a struct of employees:

```c
struct Employee {
    char* name;
    int   age;
    int   salary;
};

struct Employee workers[20];
```

If we want to sort them based on salary (ascendingly), we just need to define our function like this:

```c
int salary_up(const void* a, const void* b) {
    struct Employee* pa = (struct Employee*)a;
    struct Employee* pb = (struct Employee*)b;

    if (pa->salary < pb->salary) return 1;
    else if (pa->salary > pb->salary) return -1;
    else return 0;
}

qsort(workers, 20, sizeof(struct Employee), &salary_up);
```

If we want to sort them based on salary but descendingly, we only need to change our function definition, and pass the new function:

```c
int salary_down(const void* a, const void* b) {
    struct Employee* pa = (struct Employee*)a;
    struct Employee* pb = (struct Employee*)b;

    if (pa->salary > pb->salary) return 1;
    else if (pa->salary < pb->salary) return -1;
    else return 0;
}

qsort(workers, 20, sizeof(struct Employee), &salary_down);
```

2.5 Dynamic Memory Management

2.5.1 A Brief Introduction to Process Image

In general, when we compile our C code, the compiler will first translate that into assembly code, which includes variables and machine instructions. These things will be stored and organized into an area of the memory (Figure 2.3).

The virtual memory address starts from `0x0` to a large number. On a 32-bit machine, the virtual memory size is always 4GB: 3GB for the user addressable portion as in Figure 2.3, while the other 1GB for the kernel addressable portion (detail in Section 5.1.1).

The lowest addresses from `0x0` to a small random address is unused for security reasons.

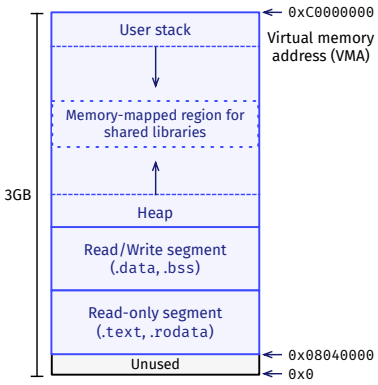


Figure 2.3: The memory diagram for a process on a 32-bit machine.

2.5.1.1 Read-Only Segment

The read-only segment occupies the second lowest address space in the virtual memory, where it contains `.text` and `.rodata` segments. The `.text` is where the binary machine code, translated from our C code, is stored. If we declared a function pointer:

```
1 int (*func_ptr)(int, int) = addition;
```

where `func_ptr` is pointing to `addition()`, then the value of `func_ptr` is an address in `.text` segment, because it points to the first instruction of `addition()`, which is stored there.

The `.rodata` segment stores read-only data, which mainly translates to string literals. [8] Consider the following two ways of declaring a string:

```
1 char  str1[]  =  "abc";
2 char* str2    =  "def";
```

The four characters `"abc"` will be placed on the local stack if declared in a function, or `.data` if declared globally, while `"def"` in the `.rodata` segment. Data on stack are mutable, while data in `.rodata` are not, obviously, so `str1[0] = 'A'` is ok, but `str2[0]= 'D'` is not.

8: Note that `const` variables are stored locally on stack.

2.5.1.2 Read/Write Segment

Going up a bit, we have read / write segment, including `.data` and `.bss`. Global variables that do not belong to any function will be placed in the `.data` segment if initialized, otherwise `.bss` (***B**lock **S**tarted by **S**ymbol*).

2.5.1.3 Stack

All the data are stored in a large area of memory, which includes stack, heap, and unused area. Stack is managed by the compiler, and contains local variables, function parameters, function returning address, and so forth. Notice that the stack grows downwards towards the empty area. This area of memory is automatically prepared by the compiler, and is where static allocation happens.

2.5.1.4 Heap

Heap is in the lower address area of memory, and grows upwards. This area is managed by us, and is dynamically allocated by our code. If somewhere in our code we want to allocate some new spaces to use, it's allocated in heap. And of course, when we're done using the space, we have to manually release it.

2.5.2 Dynamic Allocation

Recall that memory is separated into stack area and heap area. Static allocation happens in the stack area and during compilation. Once a space is allocated in this phase, we wouldn't be able to remove or free it up from memory until the program finishes. Sometimes some memory spaces are huge but only used for a small part of the program; using static allocation is obviously not optimal. Or sometimes the size of the array cannot be decided during **compilation time** but only **run time**. In this case, we can use *dynamic allocation*.

Dynamic allocation happens during run time, which means the space created by it won't be allocated during compilation. Also, dynamic allocation happens in the heap area of the memory, and allows us to "delete" that in our program so that the space can be freed.

To allocate a new space in memory manually, we use `malloc()` function declared in `<stdlib.h>`:

Listing 2.7: An example of using `malloc()` to dynamically allocate space for an integer.

```c
/*** usemalloc.c ***/
#include <stdlib.h>
int main () {
    int*   ptr;
    ptr    =  (int*)malloc(sizeof(int));
    (*ptr) =  252;
    return 0;
}
```

Here we first declare an integer pointer `ptr`. Because we didn't initialize it, it's a null pointer. We then use `malloc()` to a allocate space of an integer in the heap memory, and this function returns the address of the new allocated space to `ptr`. Because the address of this space is random, it couldn't be determined until the program executes this line. Note two things in the demo code:

- `malloc()` function takes the size we want to allocate in memory as its argument, in **bytes**. We can simply use `sizeof()` to decide the number of bytes a data type takes;
- `malloc()` function returns a `void*`, so we have to explicitly cast it to the pointer type we want.

If we want to allocate an array with, say, 10 elements, the syntax is very similar, and we just need to specify the size:

```
int* ptr  =  (int*)malloc(sizeof(int) * 10);
```

Now that `ptr` points to an array, we can use subscription to get access to its elements:

```
for (int i = 0; i < 10; i ++) ptr[i] = i;
```

Recall that the *lifetime* of statically allocated variables depends on the scope it's defined. If it's an array defined in a function, the array dies when the function dies. This is the job done by the compiler or the program.

Things are different in dynamic allocation here. Since it's us who allocated the space manually, it's also our responsibility to release it once we're done with it so that the space can be reused. To release a *dynamically allocated* space, we use `free()` function:

```
int* ptr_integer = (int*) malloc(sizeof(int));
free(ptr_integer);
int* ptr_array = (int*) malloc(sizeof(int) * 10);
free(ptr_array);
```

Caution!

A very common mistake is when allocating space for strings. Say the string is `str`, then `strlen(str)` returns the number of characters **without** the null terminator. A call such as `malloc(strlen(str))` is problematic. Instead, the following should be used: `malloc(strlen(str)+1)`

2.5.3 Memory Leak

Now that we are in charge of an area of memory, there are definitely careless people who just abuse the power. Let's look at this guy:

```
int* ptr = (int*) malloc(sizeof(int) * 100);
/* Oops, I changed my mind;
   I only need an array of 10 elements;
   There we go: */
ptr = (int*) malloc(sizeof(int) * 10);
/* OK here I do something to the 10-element array; */
free(ptr);
```

It looks fine, because there is a `free()` at the end. However, this will cause **memory leak**, which could be a very serious problem in larger projects. First, an 100-element array is allocated in memory, and its address is recorded in `ptr`. Without releasing this area, `ptr` then points to another area of the memory, which is allocated with ten integers. What happens to that 100-element array? It's not used, but still there; that part of the memory is leaked. It won't disappear until we manually `free()` it, but there's no way to do it because its address is lost. Without the address, or the pointer, we simply can't do it.

The following example will compile and run but it'll crash. Could you explain why, and provide a solution?

Listing 2.8: A subtle error that causes memory leak (and segmentation fault).

```c
/*** memleak.c ***/
#include <stdio.h>
#include <stdlib.h>
void create_array (int* vector) {
    vector = (int*) malloc(sizeof(int) * 10);
}

int main () {
    int* vector;
    create_array(vector);
    for (int i = 0; i < 5; i ++) printf("%d\n", vector[i]);
    free(vector);
    return 0;
}
```

2.5.4 Multi-Dimensional Arrays

We know there's static way to create multi-dimension arrays, such as

```c
char mat[3][4];
```

and so on. We can certainly dynamically create multi-dimensional arrays as well, but it might need a little bit more attention to details.

If we'd like to create a two-dimensional array, or a matrix, we can certainly delcare a `char**`. The first star is a pointer to an array of `char*` pointers, while the second star to an array of actual `char`'s.

In the following example, we first initialize `char**` by calling `malloc()`.

```c
#define N_ROWS   5
#define N_COLS   4
char** group = (char**) malloc(sizeof(char*) * N_ROWS);
```

Notice that we cast the return of `malloc()` to `char**` to match the left side of the expression. Because each element in this array is a `char*` not a `char`, we pass `char*` to `sizeof()`. You can think this is the step to initialize all row pointers.

Now for each row, we need to actually create an array of char's. This needs to be done for each row individually by using a `for`-loop:

```c
for (int i = 0; i < N_ROWS; i ++) {
    group[i] = (char*) malloc(sizeof(char) * N_COLS);
}
```

Releasing the memory also needs caution. Something you definitely shouldn't do is to release the row pointers first like `free(group)`, because you'll

lose the address to each individual row, and thus have a memory leak. We should first release each row, and lastly release the row pointer:

```
for (int i = 0; i < N_ROWS; i ++) free(group[i]);
free(group);
```

2.6 C Compilation Process

The tool we use a lot, `gcc` (which stands for the GNU Compiler Collection), contains all the key software we need for turning a C source code to an executable: preprocessor, compiler, assembler, and linker. When we type `gcc` without any flags, it runs through the entire compilation chain, and generates the final executable. We can specify flags to check intermediate products as well.

We use Figure 2.4 to show the entire process, as well as Table 2.1.

2.6.1 Preprocessor

You probably noticed that `#include` starts with a hashtag. They are **directives** for the preprocessor. When we invoke `gcc`, before the compiler translates our C code into assembly, the first small step is to run preprocessor.

To see the output of preprocessor, use `-E` option:

```
$ gcc macro.c -E
```

You can also use redirection to save the output to a file:

```
$ gcc macro.c -E > macro.i
```

2.6.1.1 Macros

We sometimes want to have symbolic representation of numbers so that it's more convenient and readable. For example, which of the following expression has better readability?

Table 2.1: List of four steps in the compilation process.

	Input	Software	`gcc` flag	Output
Preprocessing	C source code	Preprocessor	`-E`	Preprocessed file
Compiling	Preprocessed file	Compiler	`-S`	Assembly code
Assembling	Assembly code	Assembler	`-c`	Object files
Linking	Object files	Linker	N/A	Executable

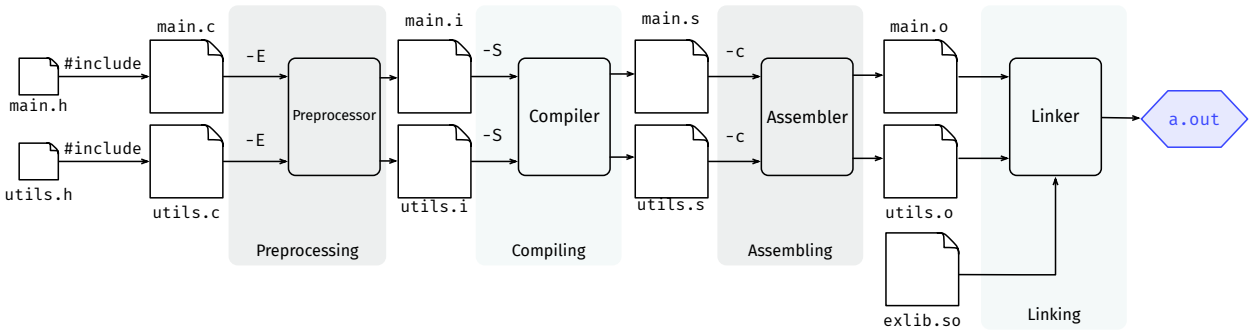


Figure 2.4: The compilation process of `gcc`, with command line flags in each step. Note that linking phase does not need any flags.

```
double x = 2 * PI   * 4;
double x = 2 * 3.14 * 4;
```

To define macros, we usually put the statement at the beginning, starting with a hashtag:

```
#define   PI   3.14
```

Basically anything can be "defined". For example:

```
#define   BADLINE    double x = 2;
int main () {
    BADLINE;
}
```

What happens is during preprocessing phase, `BADLINE` everywhere in our program will be replaced by what we defined.

Let's macro-define a `main()` brackets:

```
#define MAIN_START   int main () {
#define MAIN_END     }
```

Now we can write our `main()` in a fun way:

```
#define  MAIN_START  int main () {
#define  MAIN_END    }

MAIN_START
return 0;
MAIN_END
```

This is only for the purpose of showing you what a macros is. Unnecessary macros only make the code difficult to read and debug, so only use it when necessary!

2.6.1.2 Macro Parameters

We can also pass parameters to macros like functions:

```c
#define   CALC_AREA(x,y)   x * y
int main () {
    double length =  10;
    double width  =  20;
    double area   =  CALC_AREA(length, width);
    return 0;
}
```

What happens is `length` and `width` will replace what we defined in the macro:

```c
double area   =  length * width;
```

What we defined can be a large chunk of code as well:

```c
#include <stdio.h>
#define  COMPARE(x,y) if (x > y) printf("Larger!\n"); else
 ↪  printf("Not larger!\n");
```

To increase readability of the code, we can put our definition in a separate line:

```c
#include <stdio.h>
#define  COMPARE(x,y) \
         if (x > y) printf("Larger!\n"); \
         else printf("Not larger!\n");
```

Notice that we have a backslashes to indicate that the line is not over yet. `#define` is processed one line at a time, so we need to make sure everything will be recognized as one line.

2.6.1.3 Conditional Macros

We can also use other macros to control which part we want to compile. Generally, the conditionals have the following structures:

```c
#ifdef <symbol1>
    /* Stuff that'll be compiled
       if symbol1 is defined */
#elseif <symbol2>
    /* Stuff that'll be compiled
       if symbol2 is defined */
#else
    /* All others */
#endif
```

Let's look at the following example:

Listing 2.9: A simple example of conditional macros.

```c
/*** macro.c ***/
#include <stdio.h>
#define FINISHED
int main (int argc, char** argv) {
    #ifdef DRAFT
        printf("I'm a draft!\n");
    #else
        printf("I'm finished!\n");
    #endif
    return 0;
}
```

Try change `#define` to `DRAFT` and see what happens.

This feature is very handy when we need to change multiple parts of a code at the same time, because now we can just change the symbol defined, and the compiler will accordingly compile different parts.

The following code shows us how to use this trick to do testing. This example is to find the maximal number in an array. Assume you're only supposed to print out the final result, but there's something wrong with your code and you'd like to print out immediate result during the loop. Instead of commenting out every print statement, using macros is a way more convenient and better way:

Listing 2.10: Conditional macros can be useful for debugging code without constantly commenting out millions of `printf()`.

```c
/*** macro2.c ***/
#include <stdio.h>
#define DEBUG

int largest(int arr[], int n) {
    int i;
    int max = arr[0];
    for (i = 1; i < n; i ++) {
        #ifdef DEBUG
           printf("Checking number [%d]: %d\n", i, arr[i]);
            printf("Current max: %d\n", max);
        #endif
        if (arr[i] > max) max = arr[i];
        #ifdef DEBUG
            printf("Current max: %d\n\n", max);
        #endif
    }
    return max;
}

int main (int argc, char** argv) {
    int arr[] = {10, 324, 45, 90, 9};
    int n     = 5;
    printf("Largest in given array is %d\n",
            largest(arr, n));
    return 0;
```

```
}
```

2.6.2 Using Makefile

When a project gets larger, the compilation process can be very difficult to maintain, especially the file dependencies, such as who includes who, which file needs to be compiled first, and so on. **Make**, therefore, has become a very powerful tool to automate compilation process. The script executed by `Make` is called a **makefile**.

You can think makefile is sort of like a bash script but in a very different way. When running a bash script, the execution flow of the commands is basically linear, from the beginning to the end, and sometimes involves logic flow control. Although makefile can also execute commands, the order of execution largely depends on the dependencies of the source code, and thus it is structured in a different way, and has a different syntax.

A makefile primarily consisting of rules formatted like this:

```
target: dependencies
<tab>system_command_1
<tab>system_command_2
<tab>system_command_3
...
```

Caution!

In front of the system command it has to be a **tab**, not spaces!

Let's take a look at a very silly example first. Assume in a project, we have a makefile like the following:

```
all: dep1 dep2 dep3
    @echo "* Linking all objects..."
    @echo "Done!"

dep1: dep1.1 dep1.2
    @echo "dep1..."

dep2:
    @echo "I'm dep2"

dep3:
    @echo "I'm dep 3"

dep1.1:
    @echo "dep1.1"

dep1.2:
    @echo "dep1.2"

clean:
    @echo "Cleaning up..."
```

We name this file as `Makefile`, and simply use command `make` to invoke this. The output looks like this:

```
dep1.1
dep1.2
dep1...
I'm dep2
I'm dep 3
* Linking all objects...
Done!
```

Apparently, the execution of those `echo`'s are not from the top to the bottom as it would be in a bash script. If we draw a diagram based on each target and its dependencies, we'll end up with a tree structure as in Figure 2.5. Putting the figure and the output together, it becomes obvious that `make` executes everything from bottom to the top, which entirely follows the dependencies. For example, in order to execute commands in target `dep1`, it needs to execute the commands in `dep1.1` and `dep1.2` first.

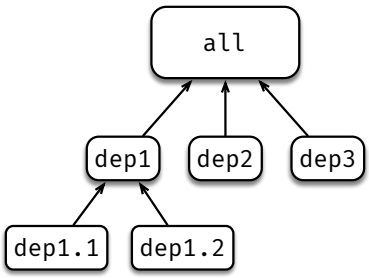


Figure 2.5: `Make` executes system commands from the bottom to the top.

We'd like to mention several small details from this silly example. First, the system commands within each rule are executed linearly. Second, if a command is prefixed with `@`, the command will be executed as usual; otherwise the command itself will also be printed out. Try this on your own by removing some of the `@`'s in the example, and see what happens. Lastly, as a convention, we have a target called `clean`, which is used to remove some of the by-products during the execution of the makefile (we'll see what it means soon). You can invoke this target by using `make clean`.

2.6.2.1 Compilation

Now that we have seen how Makefile executes, the connection with a C project file structure becomes very clear. Let's say we have a project for matrix operations, and its structure is shown in Figure 2.6.

To write out the makefile for this project, we first look at the top. The goal here is to generate a single executable `a.out`, and it depends on four object files: `transpose.o`, `linear.o`, `scalar.o`, and `external.so`. Thus, we can write our first rule:

```
all: transpose linear scalar
    @gcc transpose.o linear.o scalar.o external.so -o a.out
```

Here we made three dependencies, each of which is corresponding to an object file. Notice `external.so` is an existing shared library file that do not need to be compiled from source.

The following steps are straightforward: we just need to define the rules individually:

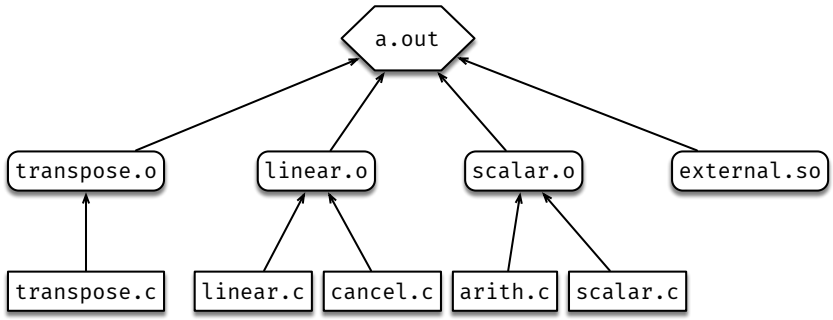


Figure 2.6: A tree structure of a C project that can be compiled using `make`.

```
transpose: transpose.c
    @gcc transpose.c -o transpose.o

linear: linear.c cancel.c
    @gcc linear.c cancel.c -o linear.o

scalar: arith.c scalar.c
    @gcc arith.c scalar.c -o scalar.o
```

Notice during the compilation we generated some object files that are necessary for the final executable, but once the executable has been generated we probably don't need them anymore. We can remove them in a `clean` target:

```
clean:
    rm -f *.o
```

2.6.2.2 Using Variables

You can also use variables when writing Makefiles. It comes in handy in situations where you want to change the compiler, or the compiler options.

```
CC=gcc
CFLAGS=-c -Wall
all: transpose linear scalar
    $(CC) transpose.o linear.o scalar.o external.so -o
    ↪ a.out
transpose: transpose.c
    $(CC) $(CFLAGS) $^ -o transpose.o

linear: linear.c cancel.c
    $(CC) $(CFLAGS) $^ -o linear.o

scalar: arith.c scalar.c
    $(CC) $(CFLAGS) $^ -o scalar.o

clean:
    rm -f *.o
```

Here `$^` is a macro that refers to all the files listed in the dependencies of that target.

There are many many other useful features of `make`. For example, you can also write a loop in a target; you can add suffixes and a lot of other special macros; and so on. Even though the example we've been using seems too simple and silly, the benefits of `make` will become more obvious and appreciable when you're working on a super large project. Just download an open-source software that needs to use `make` to install, and examine their `makefile`, and you'll see how complicated it is. However, as to the course, we will stop here and leave the rest to yourself.

Systems Programming Concepts 3

In this chapter, we offer a brief overview of fundamental concepts that will be used throughout the rest of the course. Figure 3.1 shows an organization of different parts in the system.

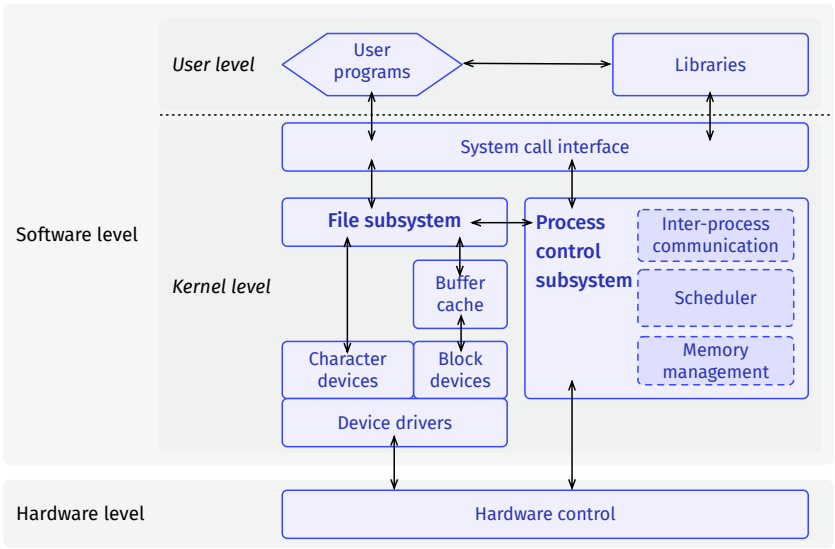


Figure 3.1: UNIX system diagram redrawn from *The Design of the UNIX Operating System* by Maurice J. Bach (1986).

In Chapter 4, we will discuss the file subsystem and buffering mechanisms. In Chapter 5, we will focus on the process control subsystem. Among the three major tasks of the process control subsystem, we will only focus on inter-process communication in Chapter 6, as the other two are out of the scope of this course. By the end of this course, we hope that you will gain the skills and basic understandings on the kernel level of Linux system.

3.1 The Kernel

All the programs running in our system need to utilize hardware resources, such as CPU, physical memory, devices, *etc*. For a computer system that can execute multiple programs, allocating and scheduling the usage of these resources is critical. The one software that does this in our system is called **kernel**. The kernel also refers to the UNIX operating system. In a sense, the kernel is the operating system.

The kernel carries out a numerous important tasks. They are also considered the essential services provided to users. When we say "services", you can simply think they are functions offered to us to use. These services include:

- Process scheduling and management;
- signaling and inter-process communication;

- Physical and virtual memory management;
- Provision of a file system;
- Creation and termination of processes;
- Device management;
- Networking services;
- Provision of a system call application programming interface (API).

3.2 Kernel Space and User Space

When UNIX boots, the kernel is loaded into the portion of the physical memory called **kernel space** and stays there until the machine is shut down. User processes (*i.e.*, the programs we usually write and execute) are not allowed to access system space; instead they stay in the **user space**.

Think about an airplane. You, as the user, took a flight to Orlando. The pilot, as the kernel, received your order, and was going to provide the service to you, by manipulating the real plane (the hardware). While the pilot was flying the plane, you were not allowed to go to his space (the kernel space), because you knew nothing about aircraft, and you'd endanger the entire system (and everybody's life!). Here, the main cabin is the user space, while the cockpit is the kernel space.

See Figure 3.2 for an illustration.

Almost all the programs that are not the kernel are executing in the user space. When a program is running in the user space, we also say it's in the **user mode**. Typically, a program needs to get access to devices. For example, when you write `printf()` in a C code, displaying the string on the screen needs to get access to the monitor. Remember we discussed that devices can only be accessed by the kernel which is running in the kernel space, so inevitably we need to request a service to the kernel to print the string for us.

During the printing, the program will have to transition into the kernel space, and thus run in the **kernel mode**. Once the string has been printed, it'll come back to the user space and thus the user mode.

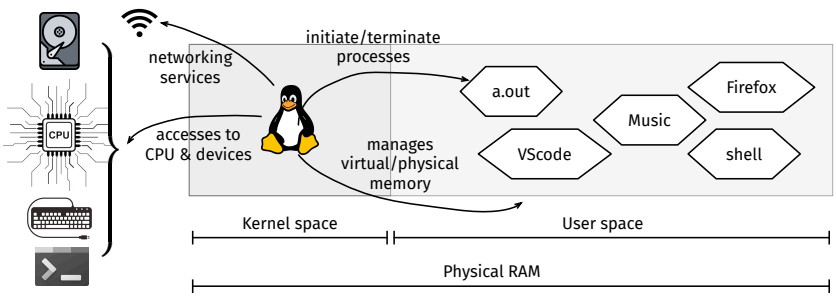


Figure 3.2: Physical RAM is split into kernel space and user space.

3.3 System Calls

How does a user program request a service from the kernel? This is done by a special group of functions, called **system calls**.

System calls are special in the sense that during the call it is kernel that's running, and therefore they have accesses to hardware and have privileged instructions. When a user runs a program that needs system call, the program goes from user mode to kernel mode, which requires a change in the execution mode of the processor.

We can invoke the system call with a special function named `syscall()`, passing the system call's identification number and arguments. Every system call has a unique number associated to it.

Generally, if a system call's name is `fun`, its syscall number is defined by a macro named either `__NR_fun` or `SYS_fun`. To use those, we usually need to include the header `<sys/syscall.h>`. You can get the complete set of system calls here: https://man7.org/linux/man-pages/man2/syscalls.2.html.

We don't always have to use `syscall()` function; instead, we can just invoke the system calls by their C wrapper functions. This is actually more common.

Example 3.1

This example is borrowed from https://jameshfisher.com/2018/02/19/how-to-syscall-in-c/, and it's a very good example to show how C functions we've been familiar with are executed. Figure 3.3 illustrates this process.

If we want to print `"Hello World!"` in C, the first thing that came to mind is:

```c
#include <stdio.h>
...
printf("Hello World!\n");
```

Here `printf()` is a C library function from the header file `<stdio.h>`. Because printing needs access to the hardware device, we have to invoke a system call. In fact, inside `printf()` somewhere, it looks like this:

```c
#include <unistd.h>
...
write(1, "Hello World!\n", 13);
```

So we can also use `write()` function instead of `printf()`, but remember the pain when you want to print a number? Indeed,

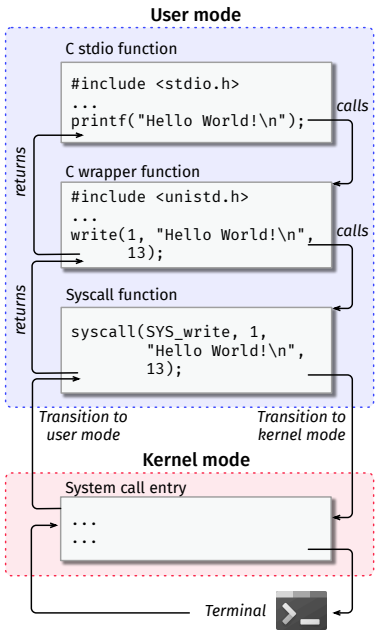


Figure 3.3: An illustration of Example 3.1.

`printf()` is a more user or programmer friendly function.

Now `write()` is still a C wrapper function of the system call, but not the actual `write` system call itself. Thus, somewhere in `write()` function, it invokes the actual system call like this:

```
#include <sys/syscall.h>
...
syscall(SYS_write, 1, "Hello World!\n", 13);
```

where `SYS_write` is the system call number.

3.4 A Really Brief Timeline on UNIX

- **1969:** Ken Thompson and Dennis Ritchie wrote the first version of UNIX at AT&T Bell Labs;
- **1973:** Thompson and Ritchie published the first paper on UNIX (see Resources on Canvas);
- **1974:** UC-Berkeley (UCB) acquired the system and started adding more features. Their version of UNIX is usually called Berkeley Software Distribution (BSD); AT&T started liscensing UNIX to universities as well;
- **1974 – 1979:** UCB and AT&T were developing their own versions on UNIX independently;
- **1979:** AT&T started trademarked UNIX, selling UNIX commercially, and made it expensive to own it. BSD remains free to acquire;
- **1985:** Steve Jobs was fired from Apple , and founded a company called NeXT, Inc. To provide a system for his products, he acquired a copy of 4.3BSD UNIX;
- **1991:** Linux was developed by Linus Torvalds. Linux itself contains only the kernel, and it behaves ***like*** UNIX. It doesn't, however, contain any code of UNIX;
- **1993:** AT&T divested itself of UNIX, selling it to Novell, which one year later sold the trademark to an industry consortium known as X/Open;
- **1996:** Apple bought NeXT, and therefore acquired Steve Jobs' operating system (based on BSD UNIX). This has become the early version of macOS.

At this point, with numerous distributions and versions of UNIX and UNIX-like systems, compatibility becomes a huge issue. Therefore, the IEEE Computer Society specified a family of standards, called Portable Operating System Interface (POSIX).

POSIX is also a trademark of IEEE, and therefore it costs a fortune to make an operating system POSIX-certified. Only a small number of systems are

POSIX certified. Apple's macOS, and most of the Linux distributions are "mostly POSIX compliant".

File Subsystem 4

4.1 Basic Concepts of Files

When talking about files, you probably would think of pictures, a video clip, a Word document, *etc*. In our course, "file" is just a generic term, because **everything is a file in UNIX**: directories (folders), devices (like keyboards, monitors, and printers), documents, programs, network connections, and even random access memory. By making everything a file, UNIX's designers simplified the operating system, making it easy to extend and maintain. [1]

1: Fun fact: when UNIX first introduced this concept of "file" into operating systems, it was considered a rather radical idea.

4.1.1 The `ls` Command

The most straightforward way to examine the attributes of files is to use one of the most commonly used command `ls`. It is also a good start to see what kind of information about a file does the system store. There are lots of flags we can pass to `ls` command to see different output. [2] In the following, we use `ls -l ~` to show the files in the home directory, and the example output is shown in Figure 4.1.

2: You can simply run `ls --help` to check it out.

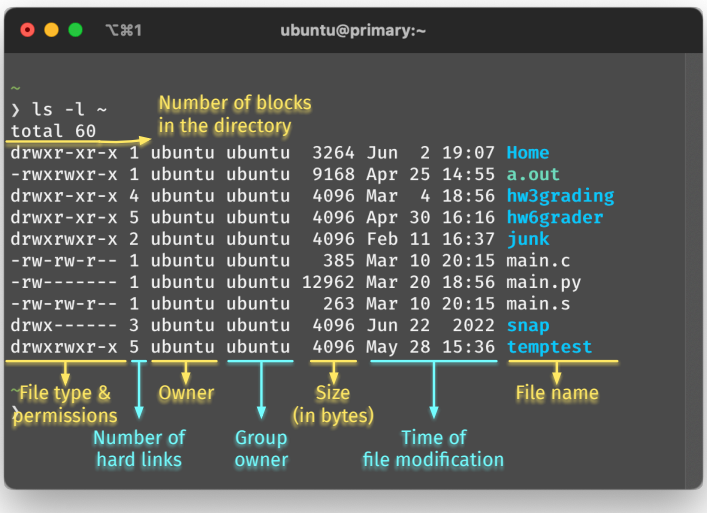


Figure 4.1: Output and annotation of an `ls` command.

Some fields of the output of `ls` command are straightforward, while some are not. For example, the first line with `total` is the number of blocks in the directory, and the second column in the output indicates the number of hard links. The concept of "blocks" and "hard links" are related to how files are organized on an UNIX file system, which are beyond the scope of this course. We will, however, briefly discuss this later in this chapter.

Note that the output of `ls -l` only shows a portion of file information. Later soon we will learn about all the information about files a system stores.

Now let's focus on the first column, which encodes file types and file permissions. It is a ten character string, where the first character indicates the file type, and the rest nine characters file permissions.

4.1.2 File Types

As mentioned, UNIX system categorizes everything into a type of file, even hardware. Speaking of file types, you might be familiar with something like `pdf`, `jpg`, *etc.* These file types are more on the application level and used typically for choosing correct software to decode and display the content. On a system level, however, there are only seven types of files, as defined by POSIX standards:

3: macOS uses `r` while Linux uses `-`

- `-` or `r`: regular file; [3]
- `d`: directory;
- `b`: block special file (*e.g.,* storage devices);
- `c`: character special file (*e.g.,* graphics cards, keyboards, mouse, main memory);
- `l`: symbolic link;
- `p`: pipe (both named and unnamed); and
- `s`: socket.

When using `ls -l` command, the first character in the first column indicates the file types, based on the list above.

All these types of files, except regular file, have an internal data structure inside the system. For example, a directory can be considered as a list of names of files it contains. Regular files, on the other hand, do not have internal structures, and are simply a sequence of bytes. Therefore, the file types we mentioned such as `pdf` and `jpg` are all considered as regular files.

4.1.3 File Permissions

Now let's move on to the next nine characters in the output of `ls -l`.

Recall in the beginning, we mentioned that UNIX system is a time-sharing operating system, meaning multiple user can share the entire system and the computing resources it provides. Each user just needs to create and log into their own account. While this is convenient, security and privacy problems also become obvious.

File permission is a mechanism in UNIX system to protect files from being read, written, or executed by other users on the system. Typically, there are three different roles in an UNIX system: user, group, others, which you can remember with the acronym **ugo**. Every file has an owner/creator, called its **user**. The file is also associated with one of the groups to which

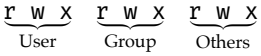
the owner belongs, called its **group**. Lastly, everyone who is neither the user nor a member of the file's group is in the class known as **others**.

For each role mentioned above, a file also has three modes of access: **read**, **write**, and **execute**. [4]

4: For directories, execute access is the ability to `cd` into the directory and as a result, the ability to run programs contained in the directory and run programs that need to access the attributes or contents of files within that directory.

4.1.3.1 Character Representation

Now that we have three roles and three modes of access, each file can be described as a nine-character permission bit string:



If a role doesn't have certain access permission, that permission character will be represented as `-`.

4.1.3.2 Octal Representation

The character representation is user friendly, but let's do the math here: if for each file we use the character representation for file permission, we need nine characters, or nine bytes. The actual information each byte stores, however, is more of a boolean: this role has this permission, or doesn't. Apparently, using nine bytes (72 bits) to store file permission seems like an overkill.

How about this: for each role, let's enumerate all the possible combinations of `rwx`:

rwx	rw-	r-x	r--	-wx	-w-	--x	---
111	110	101	110	011	010	001	000
7	6	5	4	3	2	1	0

We can use the following equation to calculate the octal value of a role's file access permission: [5]

```
value = r?4:0 + w?2:0 + x?1:0
```

5: This is a C syntax:
`a = b ? c : d` is equivalent to:
`if (b == 1) a = c; else a = d;`

As learned in CS-382, to encode eight different information, only $\log_2 8 = 3$ bits is needed. This idea saves lots of space, because now for each role we only need three bits, and for each file, nine bits is enough to represent the complete file permission! For example, a file with `rw-r-----` will have an octal number of `640` stored as the permission.

4.1.3.3 Special Permissions

Sometimes when you look at the permission of some file or directory, you'll notice that in addition to `rwx` there's also some other characters, such as `s` and `t`. They are some special permissions.

► **Letter `s`**

The letter `s` stands for "***s***pecial", which can appear on both the user's and the group's `x` permission. It's also called `setUID`.

When you see `s` on the user, that means this file is executable, but regardless of who actually executes this file (maybe someone else in the group), it always acts as if the user is executing it. One example is the command that resets the password of the user:

```
-rwsr-xr-x 1 root root 63K Nov 29  2022 /usr/bin/passwd
```

When `s` is on the group, similarly, the file will be executed as if invoked by the group it belongs to. When it's present on a directory, additionally, all the files within that directory will have the same group ownership as the directory.

A capital `S` means the user or group ID has been set for the file, but it does not have the permission to execute. Note that `setUID` only works on binary executable. It does not work on bash scripts.

► **Letter `t`**

The letter `t` stands for "s***t***icky bit", which can only be seen in the `x` permission on others (the last permission letter). This letter has no effect on files, but when it's present in the permission of a directory, a file under that directory can only be deleted by the file's owner and the root. For example, `/tmp/` directory's permission looks like this:

```
drwxrwxrwt  15 root root  12K Dec 27 13:13 tmp
```

4.1.4 Index Nodes (inode)

The operating system is responsible for retrieving these information (the "metadata") of each file on the file system. A very straightforward way to do this is to create a struct that has all the metadata of a file, and every time a new file is created on the system, an object of this struct is created, and all the fields of the structs are filled up as well. This struct is called **index node**, or more commonly, **inode**.

In addition to the metadata of a file, inode also stores the physical address of the file on the hard drive. After all, when opening a file, the system needs to know exactly where to find and retrieve the file content on the hard drive. [6] The following list shows all the information an inode stores:

- The type of the file;
- The mode of the file (the set of read-write-execute permissions);
- The number of hard links to the file;

6: If you are interested in how does inode point to physical address of a file on a hard drive, do check out UNIX File System (UFS).

- The user ID of the owner of the file;
- The group ID to which the file belongs;
- The number of bytes in the file;
- An array of 15 disk-block addresses;
- The date and time the file was last accessed;
- The date and time the file was last modified;
- The date and time the inode was changed.

Each file has an inode number associated with it, which is just a positive integer. Technically, an inode number is unique only within a partition: we can have two files that have same inode numbers, which means they reside in two different partitions of the same file system. We can use `ls -il` to also show the inode number of each file.

I encourage you to try to read Linux source to see how inode is declared: https://github.com/torvalds/linux/blob/master/include/linux/fs.h#L620. You'll very likely get lost, but that's a great way to learn. Alternatively, the webpage http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch12lev1sec6.html has a cleaner version that's easier to read and explore.

4.2 Retrieving File Information

4.2.1 The `stat` Command

Even though `ls -l` command is useful to show some information about files, it doesn't show all the data associated with them. A more useful command is `stat`. Figure 4.2 shows an example output of `stat` command on a file `a.out`.

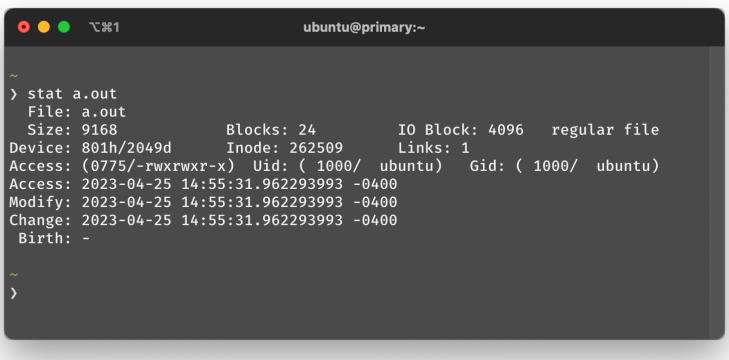


Figure 4.2: Example output of `stat` command.

At this point, we have introduced many fields presented in the output of `stat`. Fields such as device, IO block, *etc.*, are out of the scope of this course.

The `stat` command utilizes system API to retrieve file information. We can also use the interface to implement our own version of `stat` command, and/or to perform file system related tasks in a larger project.

4.2.2 The `stat` Struct

To retrieve all the metadata of a file, the first step is to create an object of `struct stat`, which will be used to fill in the information about a specific file.

The `struct stat` is declared as follows, in `<sys/stat.h>`:[7]

```
struct stat {
   dev_t     st_dev;     /* ID of device containing file */
   ino_t     st_ino;     /* inode number */
   mode_t    st_mode;    /* file type and
                            permission bits */
   nlink_t   st_nlink;   /* number of hard links */
   uid_t     st_uid;     /* user ID of owner */
   gid_t     st_gid;     /* group ID of owner */
   dev_t     st_rdev;    /* device ID (if special file) */
   off_t     st_size;    /* total size, in bytes */
   blksize_t st_blksize; /* the "preferred" block size
                            for FS I/O. */
   blkcnt_t  st_blocks;  /* number of 512B blocks
                            allocated */
   time_t    st_atime;   /* time of last access */
   time_t    st_mtime;   /* time of last modification */
   time_t    st_ctime;   /* time of last status change */
};
```

This struct has been declared in `<sys/stat.h>`, so we do not need to declare it again in our code. Instead, we just need to include the header file `<sys/stat.h>`, and create an object of this struct.

7: The field `st_blocks` represents the number of 512-byte blocks assigned to a file. The number 512 here is chosen because it's the smallest block size of any file system implemented for UNIX.

4.2.3 The `stat()` Function

After creating an empty object of `struct stat`, the function `stat()` can be used to retrieve file information based on the path to the file:

```
int stat(const char* path,  struct stat* buf);
```

where `path` is the pathname of the file, `buf` the `stat` object we just created. Note the second parameter `buf` is a pointer to the `stat` object, because the function needs to modify the values in the `stat` object, upon successful execution. If, say, the file pointed by `path` cannot be found, the function will set `errno`, and return -1. If successful, the function will return 0. This is true for most of the functions from system API — when there's something wrong, it doesn't crash the program; instead it sets `errno`, and returns a non-zero integer (typically -1) as a status code to indicate there's something wrong. Therefore, it is critical to check the return value of all functions from system API.

In the following example, we called `stat()` function, and printed the return value of it.

```c
#include <sys/stat.h>
#include <stdio.h>
int main() {
    struct stat fileinfo;
    char   filename[] =  "src.3e.tar.gz";
    int    status     =  stat(filename, &fileinfo);
    printf("%d\n", status);
    return 0;
}
```

Listing 4.1: Using `stat()` function with `struct stat` to retrieve file information.

Once `stat()` function has been executed successfully, we can print out the member variables in `struct stat` and examine the information we are looking for. Check out the following example, and pay attention to how we handle `stat()` function failure.

```c
/*** filestat.c ***/
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    struct stat fileinfo;
    char filename[] = "filestat.c";
    int status = stat(filename, &fileinfo);

    if (!status) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    printf("Inode number: %llu\n", fileinfo.st_ino);
    printf("User ID: %d\n",        fileinfo.st_uid);
    printf("Total size: %d\n",     fileinfo.st_size);
    printf("Mode: %d\n",           fileinfo.st_mode);

    return 0;
}
```

Listing 4.2: Printing out file information retrieved from `stat()`.

4.2.4 The `st_mode` Variable

When you are trying out Listing 4.2, you probably noticed that when printing `st_mode` as an integer, some large number is printed. From the definition of `struct stat`, `st_mode` encodes file types and file permissions, so to extract the information we are looking for, we have to take a look at how this variable is used by the system.

The `st_mode` variable is a 16-bit integer, where most significant four bits represent the file's type, and the remaining lower 12 bits the access per-

missions and their modifiers:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
File type				Perm changing			User			Group			Others		

The permission changing bits (9th – 11th bits) include `setuid` bit, `setgid` bit, and sticky bit, from most to least significant bits. [8]

8: This is beyond the scope of the course, but take the `setuid` bit as an example. If this bit is set, when anyone in the system executes the file, the privileges of the file's owner is granted.

4.2.4.1 Extracting File Types from `st_mode`

We know `st_mode` contains file types, but it encodes different information into a single variable, so how can we extract the file type from it?

The system API provides a macro test for each type of files:

- `S_ISREG(st_mode)` : regular file;
- `S_ISDIR(st_mode)` : directory;
- `S_ISCHR(st_mode)` : character special file;
- `S_ISBLK(st_mode)` : block special file;
- `S_ISFIFO(st_mode)` : pipe or FIFO;
- `S_ISLNK(st_mode)` : symbolic link;
- `S_ISSOCK(st_mode)` : socket.

All the above macros will return 1 if it's the corresponding file type, and 0 otherwise. This comes in handy when we want to check the file type. See the following example:

Listing 4.3: Checking file types from `st_mode`.

```c
/*** filetype.c ***/
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    struct stat fileinfo;
    char filename[] = "filestat.c";
    stat(filename, &fileinfo);

    if (S_ISREG(fileinfo.st_mode)) puts("Regular file.");
    else if (S_ISDIR(fileinfo.st_mode)) puts("Directory.");
    else puts("It's not a regular file or directory.");

    return 0;
}
```

4.2.4.2 Extracting File Permissions from `st_mode`

It needs a bit more work when extracting file permissions, because there's no testing macros provided as for file types. But the good news is, there's

another set of macros that we can perform bit-wise operations on and get what we want. The macros are defined in `<sys/stat.h>`:

```
#define S_IRWXU 0000700   /* RWX mask for owner */
#define S_IRUSR 0000400   /* R for owner */
#define S_IWUSR 0000200   /* W for owner */
#define S_IXUSR 0000100   /* X for owner */

#define S_IRWXG 0000070   /* RWX mask for group */
#define S_IRGRP 0000040   /* R for group */
#define S_IWGRP 0000020   /* W for group */
#define S_IXGRP 0000010   /* X for group */

#define S_IRWXO 0000007   /* RWX mask for other */
#define S_IROTH 0000004   /* R for other */
#define S_IWOTH 0000002   /* W for other */
#define S_IXOTH 0000001   /* X for other */

#define S_ISUID 0004000   /* set user id on execution */
#define S_ISGID 0002000   /* set group id on execution */
#define S_ISVTX 0001000   /* save swapped text after use */
```

To check if the file has a specific permission for a role, we can simply use AND operation. If the result of AND is zero, the permission does not present; otherwise it does.

The following example is to check if the owner of the file has write permission.

```
/*** checkperm.c ***/
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    struct stat fileinfo;
    char filename[] = "filestat.c";
    stat(filename, &fileinfo);

    if (S_IWUSR & fileinfo.st_mode)
        puts("Owner can write.");
    else puts("Owner cannot write.");

    return 0;
}
```

Listing 4.4: Checking file permission using AND bit-wise operation.

4.3 Reading Directories

When discussing the definition of `struct stat`, you probably noticed that, interestingly, the file name is not a member variable in the struct. In

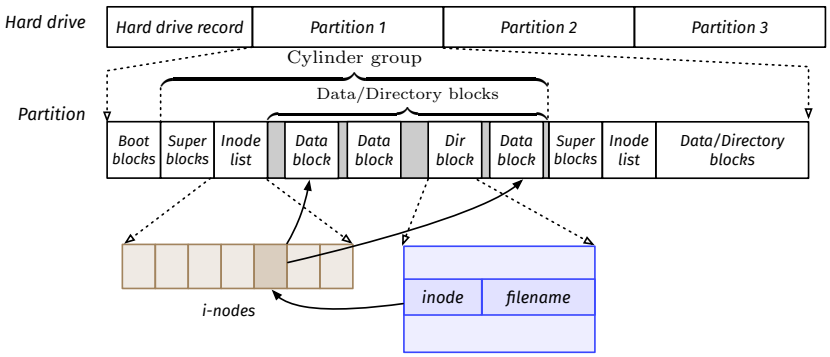


Figure 4.3: Brief overview of an UNIX file system.

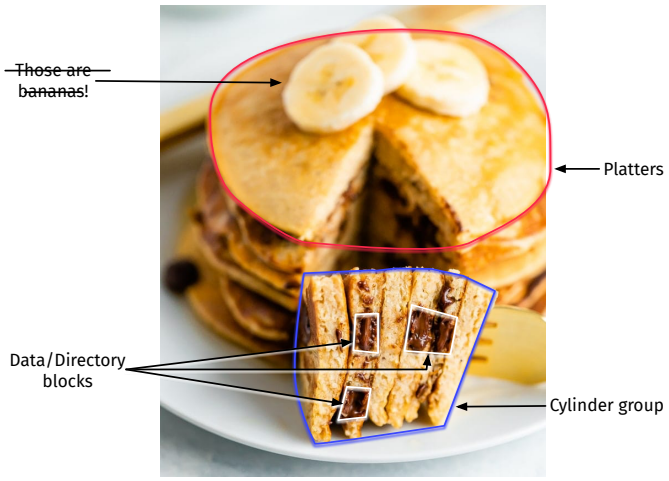


Figure 4.4: Rotating disk organization is like a pancake... mostly.

fact, the file's name is a string stored in the directory. In this section we will learn how to open and read a directory.

4.3.1 Basic Concept of Directories

You are probably more familiar with "folders", and treat directories as folders that can contain files. This type of visualization works well typically, and especially when we were discussing the tree structure of file hierarchy in UNIX. However, the files are not stored as a tree structure physically on a hard drive; instead, all directories and files are simply stored linearly, and all the structures are simply abstract.

Figure 4.3 and Figure 4.4 show us how files and directories are stored on hard drive, where the file content is stored in **data blocks**, while directories are stored in **directory blocks**. It's beyond the scope of our course to discuss this in depth, but we use the figure to show the point that the directories and files physically do not have the tree hierarchy we visualized, and that directories are not technically a "folder" that literally contain multiple files.

In fact, comparing to the name "folder", the name "directory" is more appropriate, because as shown in Figure 4.5, a directory is just a table, where each entry, called **directory entry**, represents a file stored under that directory. As you can see, each directory entry stores its inode number, and the

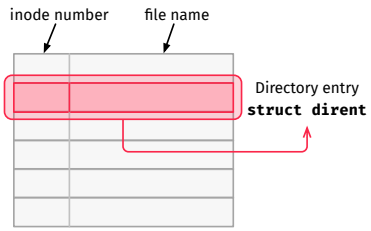


Figure 4.5: The structure of a directory.

corresponding file name. When we try to open a file, the system will use the inode number to index to the inode list on the hard drive, and read the content from the data blocks pointed by the inode objects.

The definition of `struct dirent` is in `<dirent.h>`: have `dirent` to store directory entries:

```
struct dirent {
    ino_t          d_ino;    /* inode number */
    off_t          d_off;    /* offset to the
                                next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char  d_type;   /* type of file;
                                not supported by all
                                file system types */
    char  d_name[256];       /* filename, always
                                ends with '\0' */
};
```

This definition apparently is more comprehensive than the illustration in Figure 4.5, but for us the most useful members are indeed only the inode number `d_ino` and the file name `d_name`.

Now that we know how the entries are stored, how can we use it to list all the files under a directory? In other words, how can we write our own version of `ls` command?

4.3.2 The `DIR` Type

To list all the files under a directory like what `ls` does, we need to iterate over the directory entries, and this is called to read the directory.

There's funny struct called `DIR` in `<dirent.h>`, which represents a directory stream. We say it's funny because we should **never** declare an object of that struct; we instead always declare a pointer of it, and let system functions to deal with it. This is because `DIR` is actually an incomplete type, and therefore can only be accessed by a pointer. [9]

9: Similar to an array without a size.

It might ring a bell at this point — when we tried to a regular file, we also declared a pointer type `FILE*`! In fact there are lots of similarities between reading a regular file and a directory, summarized in Table 4.1.

Table 4.1: Comparison of interfaces between reading regular files and directories.

	Regular Files	Directories
Include	`<stdio.h>`	`<dirent.h>`
Open	`FILE* fopen(char*, char*)`	`DIR* opendir(char*)`
Close	`int close(FILE*)`	`int closedir(DIR*)`
Read	`ssize_t getline(char**, size_t*, FILE*)`	`struct dirent* readdir(DIR*)`

4.3.3 Opening & Closing Directories

To read a directory, we have to open it first, and after done reading we also have to close it. To open a directory, the function is called `opendir()`:

```
DIR* opendir(const char* pathname);
```

where the parameter is the directory's path. Upon success, the function returns a valid `DIR*` which will be used for reading the directory. It is also important to check the return value of `opendir()` is not `NULL` to avoid problems in the code.

Closing a directory is also straightforward:

```
int closedir(DIR* dirp);
```

4.3.4 Opening & Checking Directories

As we said, `DIR` is an incomplete type, and can only be completed by using a function. One function is called `opendir()` which opens a directory, and returns a completed `DIR`:

```
/* Open a directory using pathname */
DIR* opendir(const char* name);
```

Now that we have a pointer to `DIR`, which is the directory we just opened, we can use `readdir()` to check out all files under it:

```
struct dirent* readdir(DIR* dirp);
```

Note that `DIR` is a **stream**.[10] Thus, every time you call `readdir()`, it'll return a pointer to the next directory entry, until `NULL` which is the end of the directory.

10: Recall when you use `fstream` to read a file, each time you call `getline`, you go to the next line, and reach the end of the file if you got a `NULL`.

The following example shows how to list all the files given a pathname.

Listing 4.5: Given a directory path, show all the files under it.

```
/*** ls.c ***/
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char** argv) {
    DIR* dp;
    struct dirent* dirp;

    /* Exit if the pathname was not passed */
    if (argc != 2) exit(EXIT_FAILURE);

    /* Open directory, and exit
```

```
        if DIR object returned is NULL*/
    dp = opendir(argv[1]);
    if (dp == NULL) {
        fprintf(stderr, "Cannot open %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    /* Use readdir in a loop until it returns NULL */
    while ((dirp = readdir(dp)) != NULL) {
        printf("%s\n", dirp->d_name);
    }

    closedir(dp);
    exit(EXIT_SUCCESS);
}
```

Quick Check 4.1

Can you simulate the command `ls -l` or `ll` using a C program? That is, given a pathname passed as command-line argument, print all file names under the directory as well as the owner, permissions, timestamps, and size of those files.

4.3.5 Navigating Directories

When we use terminals, two of the most frequently used commands are `pwd` and `cd`. In this section, we will introduce functions corresponding to these commands.

4.3.5.1 Getting Current Working Directory

We can also use the following functions to get the current directory: [11]

```
#include <unistd.h>
char* getcwd(char* buf, size_t size);
```

Function `getcwd()` will return the absolute pathname as the function value, and will also modify the parameter `buf`. The parameter `size` is the length of the character array `buf`.

The use of `getcwd()` can be a bit tricky though. If you want to store the pathname in an array (called `temp`) with length of say 256, you can do the following:

```
char temp[256];
getcwd(temp, 256);
```

11: You'll probably see there are other functions to do the same thing, such as `getwd()` and `get_current_dir_name()`. They are, however, either almost deprecated, or need additional macros declared, so we don't recommend using these.

and the absolute pathname will be put into `temp`. The return value is exactly the address of `temp`. That is, `temp == getcwd(temp,256)` is true.

The problem with this is if the absolute pathname is longer than 256 characters, you'll get an error, and the function returns `NULL`.

There's another way to work with this. We can simply do the following:

```
char* temp;
temp = getcwd(NULL, 0);
```

because this way `getcwd` will dynamically allocate (`malloc()`) a `char` array to store the absolute pathname, and there's no issue with its length. But also remember it is our responsibility to `free()` the pointer `temp` once it's used.

4.3.5.2 Changing Directory

The following function can be used to change directory:

```
int chdir(const char* path);
```

The only thing that you need to be aware of is these functions change the directory of its own thread, not the shell's. Try the following code:

Listing 4.6: We can use `chdir()` to change an executing program's directory.

```
/*** chdir.c ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char** argv) {

    char* pathname = getcwd(NULL, 0);
    if (pathname == NULL) exit(EXIT_FAILURE);
    printf("C program dir: %s\n", pathname);

    chdir(argv[1]);

    pathname = getcwd(NULL, 0);
    if (pathname == NULL) exit(EXIT_FAILURE);
    printf("C program dir changed to: %s\n", pathname);

    free(pathname);
    exit(EXIT_SUCCESS);
}
```

Assume the executable is called `./a.out`, this is what shows up on the terminal:

```
$ pwd
/home/ubuntu/Downloads

$ ./a.out ../
C program dir: /home/ubuntu/Downloads
C program dir changed to: /home/ubuntu

$ pwd
/home/ubuntu/Downloads
```

We will learn the reason behind this soon in Chapter 5.

4.3.6 Creating & Deleting Directories

The following functions declare operations to create and delete directories:

```
/* Create directories */
int mkdir(const char* pathname, mode_t mode);

/* Delete a directory and it has to be empty */
int rmdir(const char* pathname);
```

4.4 File I/O

Processes deal with files most of the time, performing I/O operations. We will look into detail how files are operated in this section.

4.4.1 File Description

In order to perform I/O operations, a process has to open a file first. What does "opening a file" do actually? Because opened files are used for I/O, the process that opened it needs to keep track of the file's current status. For example, how was this file opened? Is writing to the file allowed? Or what's the current position in the file so far? Thus, when a file is opened, the kernel creates an object of `struct file`, usually called **file description**:

```
struct file {
    struct inode*   f_inode;
    unsigned int    f_flags;
    loff_t          f_pos;
    ...
};
```

As the name suggests, it describes the current status of the *opened* file. There are lots of other fields defined in `struct file`, but most relevant

ones are shown above. [12] The `f_flags` are the file flags indicating the types of I/O operations allowed for this file, *e.g.*, read-only, write-only, *etc*. The `f_pos` indicates current reading or writing position, called **file offset**. Its type, `loff_t`, is a 64-bit value. Then there's the pointer to `struct inode`.

Each process also maintains a **file descriptor table**, which is an array of objects of `struct fd`:

```
struct fd {
    struct file* file;
    unsigned int flags;
};
```

where each entry in this array points to an object of `struct file` that represents a file opened by this process. The index of each individual entry in this array is called **file descriptor**, and that is the main handle we use to perform file I/O.

When a process starts executing, there are three files opened by default: `stdin`, `stdout`, and `stderr`, which occupy the first three entries in the file descriptor table: 0, 1, 2. Whenever we open a new file, the system creates an object of `struct file` connecting with the inode object that represents the actual file on the hard drive; then it creates a new object of `struct fd` and adds this object to the file descriptor table. This new object's index in the table becomes this file's file descriptor.

When a file is closed, `struct fd` object will be deleted, and that file descriptor becomes available for later use again.

We use Figure 4.6 to illustrate the connections between all the objects related to opened files.

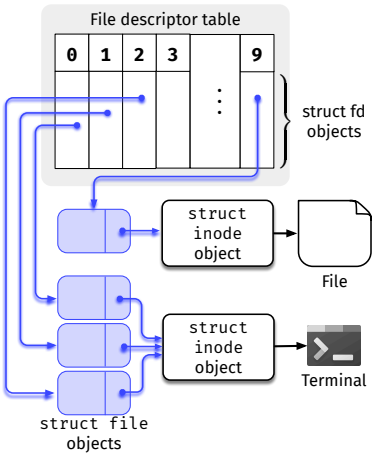


Figure 4.6: By default, every new process has `stdin`, `stdout`, and `stderr` as the first three entries of the file descriptor table, and so they have 0, 1, 2 as their file descriptors. For a new opened file, an object of `struct file` is created, pointed by a new object of `struct fd`, which is maintained in the file descriptor table.

4.4.2 I/O System Calls

File descriptor is an important tool to manage opened files. The path to a file is only used when we are trying to open it. Once it's opened, all the I/O operations are handled by file descriptors.

4.4.2.1 Opening & Closing a File

The prototype of `open()` is as follows:

```
#include <fcntl.h>
int open(const char* pathname, int flags);
```

which returns a file descriptor. When the file can be successfully opened, this file will be assigned a file descriptor in the file descriptor table. **Notice that it is always the lowest available unsigned integer that will be**

12: We won't go into details about the struct here, but in case you're interested, see source at: https://github.com/torvalds/linux/blob/master/include/linux/fs.h#L962.

assigned to this newly opened file. If the file cannot be opened, the function will return -1, and set `errno`.

The `flags` specifies the permission of I/O, and it has to have one of the following macros:

- `O_RDONLY`: read only;
- `O_WRONLY`: write only;
- `O_RDWR`: read and write.

There are also other macros available. [13] For example, if `O_CREAT` is specified, the function will create a new file if the file `pathname` doesn't exist. If `O_APPEND` is used, the function will append content to the end of the file. To combine all these options, we can OR them like this:

13: For other macros, see https://man7.org/linux/man-pages/man2/open.2.html.

```
int fd = open("test.log", O_WRONLY | O_CREAT | O_APPEND);
```

Closing a file is easy:

```
#include <unistd.h>
int close(int fd);
```

After closing the file successfully, the corresponding entry in the file descriptor table will be removed and the file descriptor becomes available. If it happens to be the lowest unsigned integer, a newly opened file will take this spot and use the file descriptor.

4.4.2.2 Reading & Writing a File

The functions to read and write files are as follows:

```
#include <unistd.h>
ssize_t read (int fd,       void* buf, size_t count);
ssize_t write(int fd, const void* buf, size_t count);
```

Function `read()` attempts to read up to `count` bytes from file descriptor `fd` into the buffer starting at `buf`, while `write()` attempts to write `count` bytes starting at `buf` to the file pointed by `fd`. The function returns number of bytes read/write if successful, 0 if it reaches the end of the file, and -1 if there's an error.

Regular files are simply a sequence of bytes. Both functions will start operating at the current file offset indicated by `f_pos` in the file description. After execution, `f_pos` will shift a certain number of bytes towards the end of the file, and this number will be returned by `read()` and `write()`. See Figure 4.7 for an illustration.

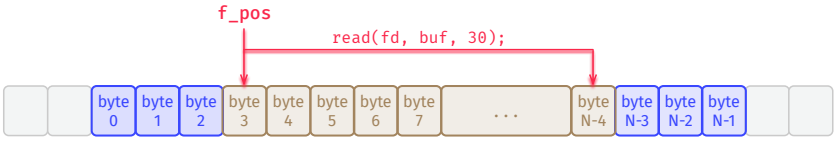


Figure 4.7: Both `read()` and `write()` function update the current position in the file by advancing `f_pos` .

> *From manpage,* https://man7.org/linux/man-pages/man2/read.2.html.
>
> ---
>
> On Linux, `read()` (and similar system calls) will transfer at most `0x7ffff000` (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit systems.)

4.4.2.3 Caution 1: Operating on Bytes

Notice that both functions are operating on bytes. When writing a string to a file, it is not a problem, because each character takes one byte. We have to be careful when we want to write something like integers to a file. For example, we can certainly use the following code to write an integer `0x5678AFFF` to a file:

```
int a = 0x5678AFFF;
write(fd, &a, sizeof(int));
```

However, if you open the file, you'll see garbage instead of the integer. This is because when we `write()` the integer, the four bytes `0x56` , `0x78` , `0xAF` , and `0xFF` are written to the file, but none of them can be used as ASCII value to present the characters.

If we really want to see the integer in the file, we'd have to manually convert the integer to a character array, and `write()` the array to the file. For example,

```
char* arr[] = {"56", "78", "AF", "FF"};
for (int i = 0; i < 4; i ++) {
    write(fd, arr+i, 2);
}
```

4.4.2.4 Caution 2: Expectation vs Reality

There's no guarantee that, especially for low-level system calls, execution of the function will succeed and do what's expected. This is particularly true for `read()` and `write()` . The following example seems correct:

```c
char* str = "Hello!";
write(fd, str, 6);
```

but in fact it is a buggy one, because there's no guarantee that `write()` function will write expected string with six characters to the file. It could write just one, or a few. A better practice is to **always check the return value of `write()` to make sure it matches the expected number**:

```c
char* str = "Hello!";
if (write(fd, str, 6) != 6) {
    fprintf(stderr, "Not complete.\n");
}
```

However, the following code is also wrong:

```c
char* str = "Hello!";
while (write(fd, str, 6) != 6);
```

This is because after the first `write()` call, even if less than six characters have been written, `f_pos` has already been shifted. The file might eventually look like this:

```
HHHellHelHellHellHelloHeHello!
```

Example 4.1: `write()` can be incomplete

Assume we have an empty file called `red.txt`. Consider the following code:

```c
int main() {
    int   red   =   open("red.txt", O_RDWR);
    char* str   =   "blue";
    write(red, str, 4);
    close(red);
}
```

What could be the contents of `red.txt` after we run this code?

- ✓ `blue`
- ✓ `blu`
- ✓ The file is still empty;
- ✗ The program is guaranteed to error;
- ✗ None of the above.

Solution: Remember that there's not guarantee that all characters will be written to the file, so the first three choices are all possible.

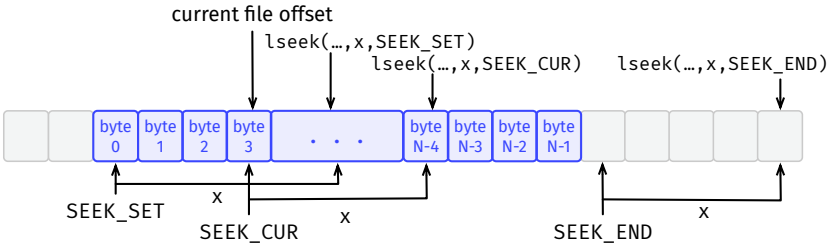


Figure 4.8: `lseek()` changes current file offset based on `offset` and `whence`. The blue boxes indicate the file.

4.4.3 File Reposition

Both `read()` and `write()` implicitly change file offset `f_pos`. We can also explicitly change it by calling `lseek()` function: [14]

```
off_t lseek(int fd, off_t offset, int whence);
```

where `off_t` is offset in bytes, and it can be 32- or 64-bit. `whence` can be one of the three places (declared as macros):

- `SEEK_SET`: relocate to the beginning of the file (offset of zero) + `offset`. So it's basically just `offset`;
- `SEEK_END`: relocated to the end of the file (or, the size of the file) + `offset`;
- `SEEK_CUR`: relocate to its current location within the file + `offset`;

See Figure 4.8 for an illustration.

Quick Check 4.2

See if you can figure out what will the offset be changed to for each of the statements (they are individual statements):

```
lseek(fd, 0,     SEEK_SET);
lseek(fd, 0,     SEEK_END);
lseek(fd, -1,    SEEK_END);
lseek(fd, -10,   SEEK_CUR);
lseek(fd, 10000, SEEK_END);
```

Note that only a few types of files can be used with `lseek()`. For example, `lseek()` will fail with pipe, FIFO, socket, or terminal.

4.5 Buffering

Since everything including hardware in UNIX is considered a file, to utilize those hardware, we have to connect them to the file system. Some devices need immediate access without delay, such as RAM, keyboard, monitor, *etc*. These are called **character special devices**. Some other devices, however, would be better to "cache" some data first in order to avoid

14: The letter `l` stands for `long` integers.

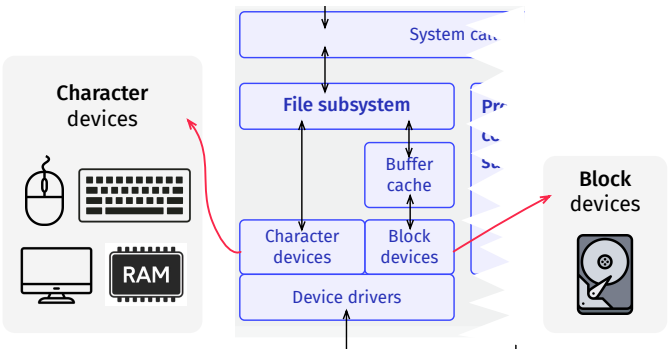


Figure 4.9: Character special devices and block special devices.

longer access time, and these are called **block special devices**, such as your hard drive. See Figure 4.9 for their position in the kernel space.

4.5.1 Kernel Space Buffering

Let's focus on block devices first. Recall that getting access to a file on a hard drive is very very slow. [15] If, say, we directly write or read to a file on a disk, the delay would be intolerable. Therefore, we utilize the idea of "cache", and create a "buffer cache (or just buffer)" between the hard drive and the file system (see Figure 4.9). Whenever we `open()` a file, the data contained in this file will be copied in to the cache. In this way, we directly operate on the copy of the file in the buffer cache, instead of on the hard drive.

15: Refer to memory hierarchy in CS-382.

What is this buffer and where is it? Now think about memory hierarchy again. If the hard drive is too slow, and we want to place the copy of the file to a place where we have faster access, where would you do? It's the main memory. There you go! The buffer cache is not a hardware cache like L1- or L2-cache (SRAM) we talked about before. It's a data structure maintained by the kernel inside the main memory. We often say the buffer has an **in-memory** copy of the file.

During booting of the system, the kernel will allocate a couple of buffers in a certain location of the memory. Each of the buffers include a data area (used to store the copies from the hard drive), and a buffer header to identify this buffer. Part of the buffer header is defined as follows: [16]

16: See https://github.com/torvalds/linux/blob/master/include/linux/buffer_head.h#L60.

```
1  struct buffer_head {
2
3      /* buffer state */
4      unsigned long        b_state;
5
6      /* circular list of page's buffers */
7      struct buffer_head*  b_this_page;
8
9      /* the page this bh is mapped to */
10     struct page*         b_page;
11
12     /* start block number */
```

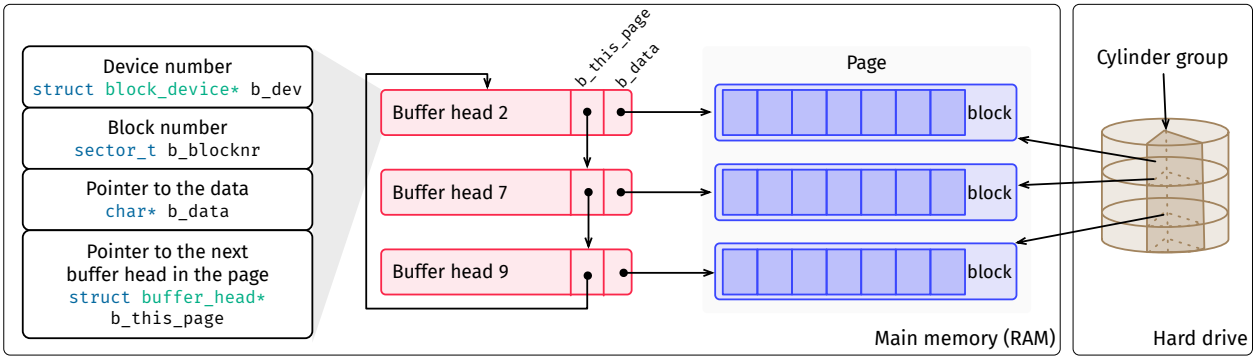


Figure 4.10: Mapping of data blocks between hard drive and buffer in main memory.

```
13      sector_t            b_blocknr;
14
15      /* pointer to data within the page */
16      char*               b_data;
17
18      struct block_device* b_bdev;
19      ...
20  };
```

Let's analyze this following the struct definition and Figure 4.10. When we `open()` a file (for now assume it's a regular file stored on a hard drive), the data blocks used to store this file are copied to main memory. Based on how many blocks are copied, the kernel creates objects of buffer head structs, and use `b_this_page` to link them one after another, and thus form a circular linked list. Each buffer head is pointing to exactly one block of data in main memory, by `b_data`. All the operations, such as `write()` and `read()` are performed in this buffer cache in main memory. When we `close()` the file, the blocks are copied back and replaced the original content on the hard drive.

4.5.2 User Space Buffering

We already talked about buffering in the kernel space, which is a "cache"-like area in main memory connecting the hard drive. In user space, we also need buffered stream between our program and system call. This is because, similarly, we want to avoid using system call too often, so we buffer some data from our program, and only when the buffer is full do we use system call.

4.5.2.1 Streams

To achieve a higher level of buffering in the user space, we operate on an object of `FILE*`, often called **streams**, declared in the standard C library `<stdio.h>`. In summary, there are two ways to get access to a file: file descriptors on lower level, or streams on higher level.

Correspondingly, the "big three": `stdin`, `stdout`, and `stderr` are also predefined and automatically available to a process through global variables. In case you're curious, here's a segment of `<stdio.h>` from Apple's open source C library: [17]

17: See https://opensource.apple.com/source/Libc/Libc-583/include/stdio.h.auto.html.

```
...
#define stdin  (&__sF[0])
#define stdout  (&__sF[1])
#define stderr  (&__sF[2])
...
```

GNU C library also defines them in the following way:

```
FILE* stdin  = (FILE *) &_IO_2_1_stdin_;
FILE* stdout = (FILE *) &_IO_2_1_stdout_;
FILE* stderr = (FILE *) &_IO_2_1_stderr_;
```

You can find the code in `glibc/libio/stdio.c`. [18]

18: Download GNU C library at https://www.gnu.org/software/libc/sources.html.

Recall that to open a file using `<stdio.h>` we use `fopen()` with `FILE*`. This function does system call `open()` for us, and associates this stream with the file descriptor. You can get this number by using `fileno()` function:

Listing 4.7: Using `fileno()` function to get file descriptor of an opened `FILE*` stream.

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv) {
    FILE* stream = fopen(argv[1], "w");
    if (stream == NULL) exit(EXIT_FAILURE);
    int fd = fileno(stream);
    printf("fd = %d\n", fd);
    fclose(stream);
    exit(EXIT_SUCCESS);
}
```

If we want to open a file using an **existing** file descriptor, we can use `fdopen()` function, which will create a `FILE` struct and associate it with the file descriptor. For example:

```
int   fd     =  open(argv[1], O_RDWR);
FILE* stream =  fdopen(fd, "w");
```

We'll use an example to show the idea behind the user space buffering. Say in our program we want to use `fprintf()` to write a string to a file. When our program runs to the line `fprintf()`, from **our** perspective the operation of writing to the file is done, *i.e.*, the string is already written in the file. This is not necessarily true, however, because this only means it's written to the **`stdio` buffer** (or **stream buffer**), not the real file. So when

we say "the actual I/O", we meant the string has been actually, physically put into the file, not just in the buffer.

C standard I/O library `stdio` provides three buffering modes: fully buffered, line buffered, and unbuffered.

4.5.2.2 Fully Buffered

This is typically used with files, such as reading from and writing to a file. The actual I/O happens in two situations:

- Reading: buffer is empty and needs to be filled;
- Writing: buffer is full and needs to be emptied.

Let's look at the following example:

Listing 4.8: An example showing the buffer mechanism in file I/O.

```c
/*** fullybuffered.c ***/
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main (int argc, char** argv) {

    off_t size;
    off_t old_size;
    FILE* stream;
    struct stat info;

    stream   = fopen(argv[1], "w");
    stat(argv[1], &info);
    size     = info.st_size;
    old_size = size;
    printf("Current file size: %lld\n", size);
    getchar();

    while (1) {
        if (size != old_size) {
            printf("File size: %lld\n", size);
            old_size = size;
            getchar();
        }
        fprintf(stream, "0");
        stat(argv[1], &info);
        size = info.st_size;
    }
    exit(EXIT_SUCCESS);
}
```

This program uses an infinite loop to write characters into a file using `fprintf()`. In each iteration, we use `stat()` to gain the current file size, and only pause when the file size changes (the `if` statement). If the file was unbuffered at all, every time we write a character to the file, the file size should've changed. What you see, however, is that the file changes in

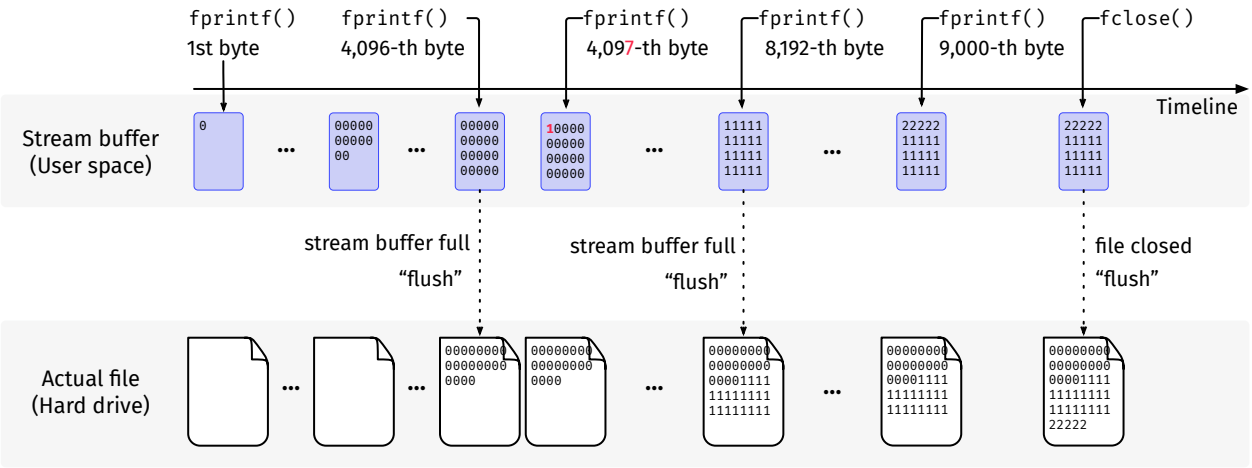


Figure 4.11: Illustration for fully buffered operation.

multiples of 4,096 bytes, which means `stdio` buffers 4,096 bytes when writing, and when the buffer is full, it'll perform the actual I/O, and call `write()` system call to push everything in this buffer to the actual file. See Figure 4.11 for an illustration of this idea.

One thing to notice is that the buffer itself will not be emptied when it's full. The new data will simply overwrite the old data.

Flushing

When the stream buffer is full, the content inside the buffer will be actually written to the disk or the hard drive. This process is called **flushing**. Flushing can be automatic (*e.g.*, when the buffer is full or we closed the file using `fclose()`), but can also be manual, by using `fflush()` function:

```c
int fflush(FILE* stream);
```

Try it

Add the following statement after `fprintf()` in Listing 4.8 and see what happens:

```c
fflush(stream);
```

4.5.2.3 Line Buffered

As the name suggests, line buffered will flush everything from buffer to the destination when a line is finished, *i.e.*, when a newline character `'\n'` is entered. Apparently this is usually used with terminal devices. Similarly, `stdout` is also line buffered but only when it's connected to the terminal; otherwise it's still fully buffered.

Note that line buffered is still buffered, after all, and therefore the buffer has a limit as well. If the buffer is full and there's still no newline character entered, everything has to be flushed, so the final data might be shorter than you actually input.

4.5.2.4 Unbuffered

Unbuffered means I/O takes place immediately. One example is `stderr`, which is never buffered. This means whenever we write something to `stderr`, it'll be put into an actual file on the drive immediately.

Example 4.2: What you wrote is probably NOT what you get

In the following C code, we want to print `"Hello World"` on the screen, but we print `"Hello "` through `stdout` while `"World!\n "` through `stderr`. What's the output? How would you explain it? And how do you get the correct result?

```
1 printf("Hello ");
2 fprintf(stderr, "World!\n");
```

Solution: Surprisingly, or not so surprisingly, the output is:

```
1 World!
2 Hello
```

Try it on your own and you'll see I ain't no lying! Remember, `stdout` is line buffered, which means it'll actually perform I/O to the real destination device only when there's a newline character at the end. When we finished the `printf()`, the string is actually in the stream buffer due to the lack of a newline character.

However, `stderr` is never buffered, meaning calling `fprintf()` with `stderr` will result in immediate I/O operation, so `"World!"` is printed first.

When the program finishes, everything that's still in the stream buffer will be flushed out to the destination, and that's why we saw `"Hello"` appeared last.

There are two solutions to address this. The first one is to call `fflush(stdout)` after `printf()`. Another one is to simply add a new line character at the end of `printf()` (although the output will be a little bit different than expected).

4.5.3 Summary

The buffering mechanism from user space `stdio` to the actual hardware can be depicted as in Figure 4.12.

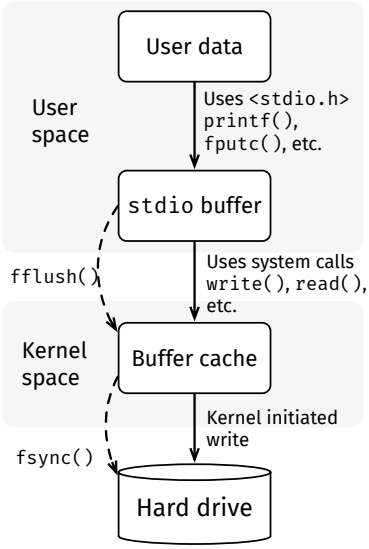


Figure 4.12: Buffering mechanism from user space to hard drive. Dotted lines indicate manual flushing/updating. The picture was in fact from Kerrisk, p.224. Some details are also removed.

Process Control Subsystem 5

5.1 Introduction

When we invoke a program, the actual instance of that program becomes a **process**. A process usually starts when it's executed and ends when `_exit()` is called and returns to the kernel (See Figure 2.2 for a refresher). In addition to the programs we run, there are in fact tons of background processes running as well, including the operating system itself.

5.1.1 Process Image

Through CS-382 we are fairly familiar with the virtual memory space of every process, which includes read-only, read/write, heap, and stack segments. In fact there's more to that. See Figure 5.1.

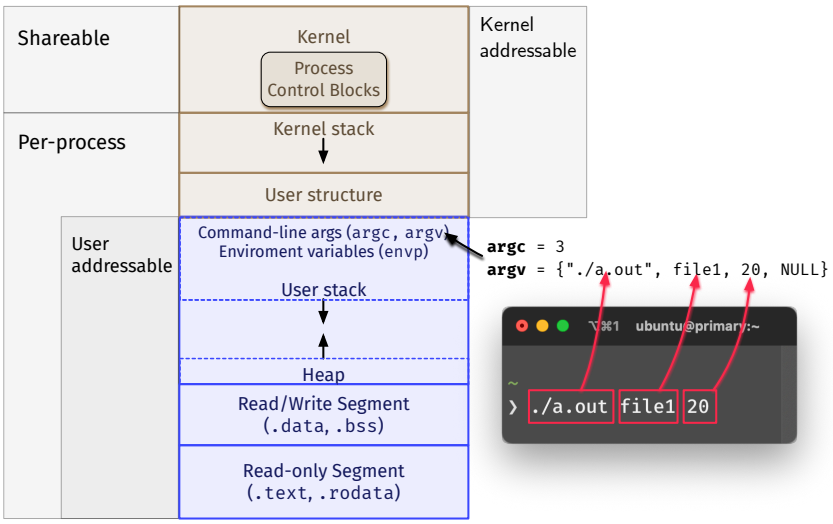


Figure 5.1: A complete diagram of a process image. Command-line arguments are saved as space-separated strings to `char**` array.

The layout of a process, as in Figure 5.1 is called a **process image**. Notice for each process image has been split into two portions: the user-addressable, and kernel-addressable.

As shown in Figure 5.2, the kernel-addressable portions are all maintained in the kernel space of the memory, while user-addressable portions the user space. In addition, each process also has a process structure (*i.e.*, PCB, which will be introduced next) to store all its status and information.

The kernel maintains a process structure for every running process, usually called **process control block (PCB)**, meaning whenever we start running a program, the kernel creates an object of the structure. This structure contains the information the kernel needs to manage the process. In Linux,

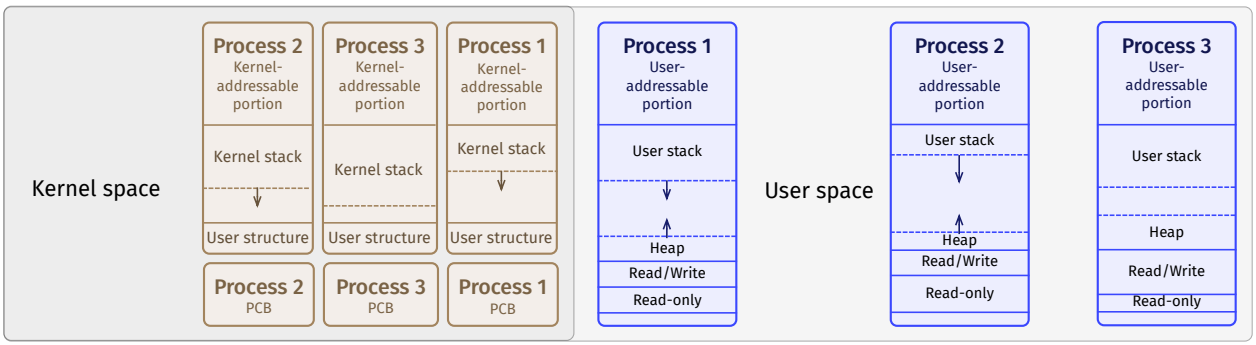


Figure 5.2: A RAM virtual address space is split into two parts: kernel space and user space. Users do not have privileges to get access (*e.g.*, dereference a pointer pointing to a location in the kernel space) to the kernel space.

this structure is called task structure (or processor descriptor), and is declared as `struct task_struct` in `<linux/sched.h>`.[1] The struct has around 800 lines and 150+ fields, almost 8KB, so we will not show all of them here; instead we'll only look at some important or essential fields:

1: See https://github.com/torvalds/linux/blob/master/include/linux/sched.h#L728

- Process ID (or PID);
- Parent process ID (or pointer to parent's PCB);
- Pointer to list of children of the process;
- Process priority for scheduling, statistics about CPU usage and last priority;
- Process status (run, wait, *etc.*);
- Signal information (signals pending, signal mask, *etc.*);
- Machine state (the contents of the registers, program counter, *etc.*);
- Timers;
- Current working directory.

Many of the fields are structs as well, and they usually include information such as:

- Process's group id (in `struct task_group`);
- User IDs associated with the process;
- Memory map for the process (where all segments start, and so on);
- File descriptor table (in `struct files_struct* files`);
- Accounting information;
- Other statistics that are reported such as page faults, *etc.*;
- Signal actions;
- Pointer to the user structure.

Among all these fields, one of the most important variable is Process ID, or **PID**. Just like the system uses file descriptors to handle files, it also uses PID — a non-negative integer — to handle processes. Each process has its own unique PID, and the kernel assigns PID to the process when it starts running.

5.1.2 The /proc/ Virtual File System

For every running process, the system also maintains a virtual file system under the directory `/proc/`. We say it's virtual because it doesn't take any physical disk space; it's in the form of a file just for the convenience of inspecting running process status.

> *From The Linux Programming Interface, Kerrisk, pp.224.*
>
> ---
>
> This file system resides under the `/proc` directory and contains various files that expose kernel information, allowing processes to conveniently read that information, and change it in some cases, using normal file I/O system calls. The `/proc` file system is said to be virtual because the files and sub-directories that it contains don't reside on a disk. Instead, the kernel creates them "on the fly" as processes access them.

5.1.3 Process States

For each process, there are five possible status:

- **Running (R):** those who are actively using CPU;
- **Uninterruptible Sleep (D):** they are not doing anything and thus don't use CPU. They do take up memory space, though. For example, when you program is waiting for user input;
- **Interruptible Sleeping (S):** it is similar to uninterruptible sleep state, except that it'll respond to signals;
- **Stopped (T):** it's very similar to sleeping, but it's done manually. We can manually bring a stopped process back to running state;
- **Zombie (Z):** they are finished so they do not take any resources such as CPU and memory, but for some reason they still appear in the process table. It'll eventually go away.

To check the states of all running processes, we can use the following command:

```
1 $ ps -l
```

Under the second column, you will see `S`, which stands for states. The letters indicate the states as listed above.

5.1.4 System Process Hierarchy

To check the status of all currently running processes in the system, we can use `htop` command. [2] Figure 5.3 is an example output of this command.

Take a look at the column `Command` carefully – it looks like the processes are organized in a tree structure. If we consider a process as a node in a

2: Another command we can use is `pstree`.

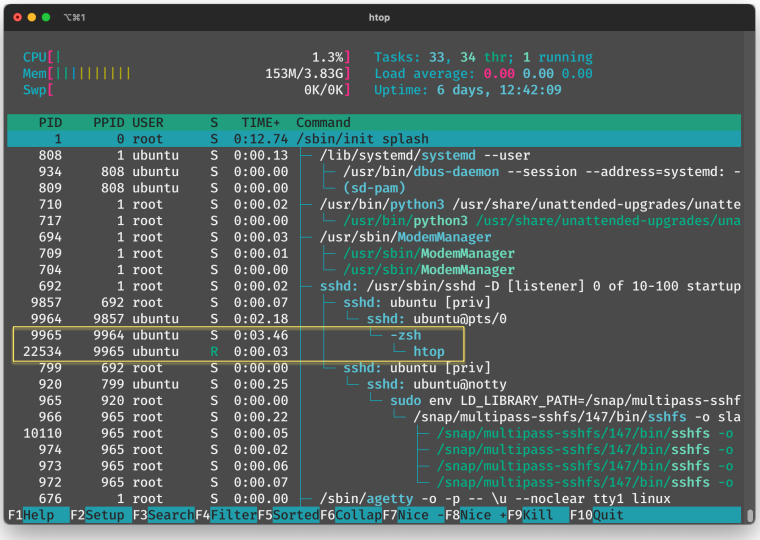


Figure 5.3: An example output from `htop` command. Click on the "Sorted" button at the bottom or press F5 to see the table in a list format. You might also see more columns than what's in the figure – you can click on "Setup" button at the bottom to add/remove columns.

tree, its parent node is called the parent process. This hierarchy stays in the entire system, and no process can be separated from this tree structure.

Every tree structure has a root node. The root in the system process hierarchy is the process with PID of 1, invoked by command `/sbin/init`. This is the very first process when we start the machine. It "coordinates the rest of the boot process and configures the environment for the use." [3] This process will stay in the system for the entire time.

3: See https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/4/html/reference_guide/s2-boot-init-shutdown-init.

Even the command `htop` itself can be found in the process table; itself is also a process, after all. Figure 5.3 highlights the entry for `htop`. Notice it's the child process of `zsh`, the shell interpreter running in the terminal.

Our question is, after receiving our command `htop`, how does the interpreter actually work to run the program `htop` as a child process?

To fully understand how exactly Linux creates such a hierarchy in the system, we take two steps: in Section 5.2, we see how to create a new process within a process, and in Section 5.3, we will see after creating a new process, how can we run existing executable programs.

5.2 Process Control

As mentioned earlier, each process has a unique PID. In a C program, the PID's type is `pid_t` declared in `<sys/types.h>`. The PID of the current process and that of its parent process can be accessed by the following functions:

```c
#include <sys/types.h>
#include <unistd.h>
pid_t getpid();
```

```
pid_t getppid();
```

5.2.1 Creating a Process

Whenever we invoke a program from terminal, the interpreter actually created process that executes our program. We can also create a child process from our program by using `fork()`:

```
pid_t fork();
```

The program that calls `fork()` is called **parent process**, while the program invoked by `fork()` **child process**.

A child process at the moment of successful call of `fork()` is just a copy of the parent — same code, same data, same everything. [4] When we say "copy", it means the entire process image (both user- and kernel-addressable portions) will be copied. Let's look at a simple example below:

4: Almost. One thing that's definitely different is their PIDs.

Listing 5.1: Creating a child process using `fork()`.

```c
/*** pid1.c ***/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {

    pid_t p = fork();

    if (p < 0) {
        perror("fork");
        return -1;
    }
    else if (p == 0) { /* child */
        printf("I am the child\n");
        return 0;
    }
    else { /* parent */
        printf("I am the parent of %d\n", pid);
        return 0;
    }
}
```

We first start this program (say `a.out`) in the terminal. Once started, `a.out` will run the code from `main()` line by line. When it runs to the function `fork()`, it creates a copy of itself, *i.e.*, the child process, and both will run at the same time. The child process, however, will only start at the line of `fork()`; all the code before `fork()` will not be executed in the child process.

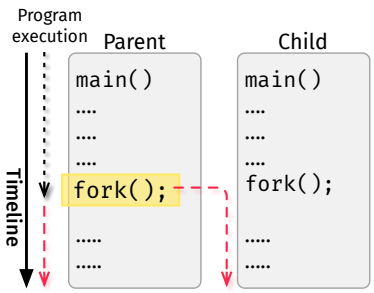


Figure 5.4: Both parent and child processes will start executing at the call of `fork()`.

Now both processes are running. For the parent, `fork()` returns its child

PID; [5] for the child, `fork()` returns 0. Therefore, a successful `fork()` call always returns twice: once in the parent and once in the child. If the integer returned is negative, there's an error. See Figure 5.4 for an illustration.

5: Note this is the only way for a parent to get its child's PID.

5.2.2 Parent vs Child Processes

There are several attributes inherited by the child process from the parent, but also differences. They're summarized as follows.

- **Properties inherited by child**:
  - Real user ID, real group ID, effective user ID, and effective group ID;
  - Supplementary group IDs;
  - Process group ID;
  - Session ID;
  - Controlling terminal;
  - The set-user-ID and set-group-ID flags;
  - Current working directory;
  - Root directory;
  - File mode creation mask;
  - Signal mask and dispositions;
  - The close-on-exec flag for any open file descriptors;
  - Environment;
  - Attached shared memory segments;
  - Memory mappings;
  - Resource limits.
- **Differences between parent and child**:
  - The return values from `fork()` are different;
  - The PIDs are different;
  - The two processes have different PPIDs: the PPID of the child is the parent; the PPID of the parent doesn't change;
  - The child's process timer values are set to 0;
  - File locks set by the parent are not inherited by the child;
  - Pending alarms are cleared for the child;
  - The set of pending signals for the child is set to the empty set.

5.2.3 Orphans and Zombies

5.2.3.1 Orphan Processes

Recall from previous discussion that if a process has terminated, it'll disappear from the process table. We also mentioned that the system has to maintain the process hierarchy all the time. Now we have a problem: say the parent process terminated before the child process and disappeared from the process table. What about the child? It lost its parent process and can't exist without one, so how does the system deal with this issue?

Let's look at a small experiment. In the following code, we let the child process stay for 2 seconds so that the parent process terminates first:

```c
/*** orphan.c ***/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t p;

    if ((p = fork()) < 0) return -1;
    else if (p == 0) sleep(2); /* child */

    printf("PID = %d, PPID = %d\n", getpid(), getppid());
    return 0;
}
```

Listing 5.2: When a parent process terminates first, the child becomes an orphan process

A possible output will look like this:

```
PID = 24083, PPID = 9965
PID = 24084, PPID = 1
```

Apparently, the first line was printed by the parent process whose parent process is the interpreter (PID = 9965). What's interesting is that the child process' PPID is not 24083 as we thought – it's 1. This actually makes sense because at the point where the child printed this line, its parent process had died, and of course no process has a PID of 24083. But why its new parent is the process with PID of 1? [6]

6: This experiment is done on a Multipass VM. On a native Linux machine, the new parent could be `systemd` whose PID is larger. The principle, however, is the same.

Recall that the system has to maintain process hierarchy all the time. When a parent process terminates before the child, the child becomes an **orphan process**, and the system has to find a new parent for it. This procedure is called **adopt** or **re-parent**. Typically, the system process `init` (the very first process), constantly looks for and collects those orphan processes. See Figure 5.5 for an illustration.

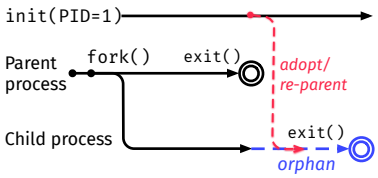


Figure 5.5: A oprhan process will be adopted by `init` process.

5.2.3.2 Zombie Processes

A process becomes an orphan if its parent process terminates first. Now what if we let the child process terminates first? Will the problem be solved?

Now this time, let the parent sleep for a bit longer instead.

```c
/*** zombie.c ***/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t pid;
```

Listing 5.3: Creating a zombie process.

```
7
8       /* error */
9       if ((pid = fork()) < 0) return -1;
10
11      /* child */
12      else if (pid == 0) return 0;
13
14      /* parent */
15      else sleep(50);
16
17      return 0;
18  }
```

In the code above, we let the parent process sleep for 50 seconds, while terminating the child process immediately. Assume the executable is called `a.out`, we start the process:

```
1  $ ./a.out &
```

and then use `ps -l` to examine the process table. See Figure 5.6.

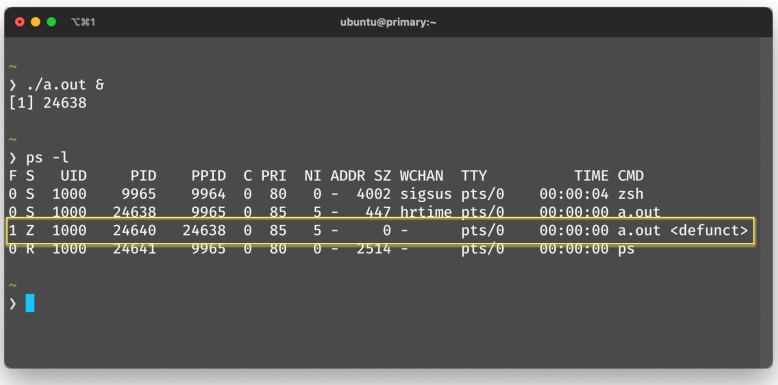


Figure 5.6: A process table that shows the zombie process.

Here we find that we have two processes, both called `a.out`. The one with PID of 10939 is the parent process we started, while the one with 10942 is the child process. This time, the process hierarchy seems correct, but we also noticed something weird there: for the child process, its status is `Z`, and marked as `<defunct>`. Here `Z` stands for zombie, meaning it has become a **zombie process** or **defunct process**. See Figure 5.7 for an illustration.

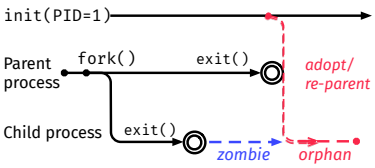


Figure 5.7: A child process becomes a zombie process when it terminates before the parent.

Zombie processes do not take any CPU resources, so basically it looks like it doesn't exist. [7] What's nasty about zombie processes, however, is that it takes an important resource – PID.

Recall that each PID is declared as a `pid_t` type, which is basically `unsigned int`. If you use `sizeof(pid_t)`, it'll return 4, meaning the system only uses 4 bytes to represent PID. What all this means is, PID is a limited resources — there can only be 2^{32} unique PIDs. If currently all the PIDs have

7: In process table you can also see the zombie process' size is 0.

been assigned to some processes, we can't start any new process, because there's simply no representable number to assign to this new process.

This is a predicament now. We have little control over how long the parent and the child processes run, so either we'll end up having an orphan process that puts lots of stress on the system, or a zombie process that's nasty. Is there a way to not leave either processes behind and terminate both parent and child processes cleanly? The answer is `wait()` function.

5.2.3.3 The `wait()` Function & Status Macros

The function we want to use in the *parent process* is:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int* status);
```

When a process starts executing `wait()`, if it has no children, `wait()` returns immediately with a -1. If this process has a child or multiple child processes, it will halt at the line where `wait()` is called, until one of its child processes has finished. Once a child process has finished, the parent process will resume executing from `wait()`, and the return value of `wait()` is the PID of that terminated child process.

The parameter of `wait()` is an integer pointer. When a child process terminates, its value will be modified based on the status of the child process. For example, if the child process terminates normally without error, `status` will be modified to 0; otherwise it'll be modified to a specific error code passed from the child process.

Let's look at the example below:

```
/*** wait.c ***/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    pid_t pid;
    int   stat;
    if ((pid = fork()) < 0) return -1;
    else if (pid == 0) {
        printf("I am %d, child of %d....zzz\n",
                getpid(), getppid());
        sleep(2);
    }
    else {
        printf("I am %d, the parent. I'll wait!\n",
                getpid());
        wait(&stat);
```

Listing 5.4: The parent process should use `wait()` function to wait for the termination of the child process to avoid zombies.

```
20          printf("Wait is over! stat = %d\nI can RIP now!\n",
21                  stat);
22      }
23      return 0;
24  }
```

You can see when the child process is terminated normally, `stat` is equal to 0. Now, let's add the following lines after line 14 in the code above, and see what happens:

```
1  int* ptr;
2  free(ptr);
```

A child process can be terminated differently, with different status. We don't need to remember specific values of the status; there are a couple of macro functions we can use:

```
1  /* Returns TRUE if terminated normally */
2  WIFEXITED(status)
3
4  /* Returns TRUE if terminated by a signal */
5  WIFSIGNALED(status)
6
7  /* Returns TRUE if stopped by delivery of a signal */
8  WIFSTOPPED(status)
9
10 /* Returns TRUE if the child produced a core dump */
11 WCOREDUMP(status)
```

Note that macro `WCOREDUMP()` is not specified in POSIX.1-2001 and is not available on some UNIX implementations. Therefore, when you use it, you should use `#ifdef` like this:

```
1  #ifdef  WCOREDUMP
2      int d = WCOREDUMP(status);
3  #else
4      int d = 0;
5  #endif
```

In addition to predefined error code, we can also use this `status` variable to do something fun:

Listing 5.5: The parameter of `exit()` from child process can be received by parent process' `wait()` function.

```
1  /*** waitexit.c ***/
2  #include <stdio.h>
3  #include <sys/wait.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  int main() {
7      int   stat;
8      pid_t pid = fork();
9
```

```
10      /* child */
11      if (pid == 0) exit(10);
12
13      /* parent */
14      else {
15          wait(&stat);
16          printf("The child said %d!\n", WEXITSTATUS(stat));
17      }
18      return 0;
19  }
```

If we change line 10 in the code above to `exit(392)`, the output won't match what we expected. If you'd like to know why, read manpage: https://man7.org/linux/man-pages/man2/wait.2.html.

5.2.3.4 The `waitpid()` Function

Another function we can use is `waitpid()`:

```
1  #include <sys/types.h>
2  #include <sys/wait.h>
3  pid_t waitpid(pid_t pid, int* status, int options);
```

It is very similar to `wait()`, but it's more flexible. The first argument `pid` can be the following:

- < -1: Wait for any child process whose process group ID is equal to the absolute value of PID. We'll talk about group ID later, but for now you can think it'll wait for all child processes of a parent;
- -1: As long as one of the child processes terminates, the parent stops waiting for others;
- 0: Wait for any child process whose process group ID is equal to that of the calling process;
- > 0: Wait for the child whose process ID is equal to the value of `pid`.

Argument `options` can be one or more (by OR-ing them) of the following macros:

- `WCONTINUED`: returns if a stopped child has been resumed by delivery of `SIGCONT`;
- `WNOHANG`: returns immediately if no child has exited, *i.e.*, no hang-up;
- `WUNTRACED`: returns if a child has stopped.

If some of the descriptions are not clear, I encourage you put a bookmark here, and skip them for now. Once we introduce process groups and signals later, come back and it'll be clearer naturally.

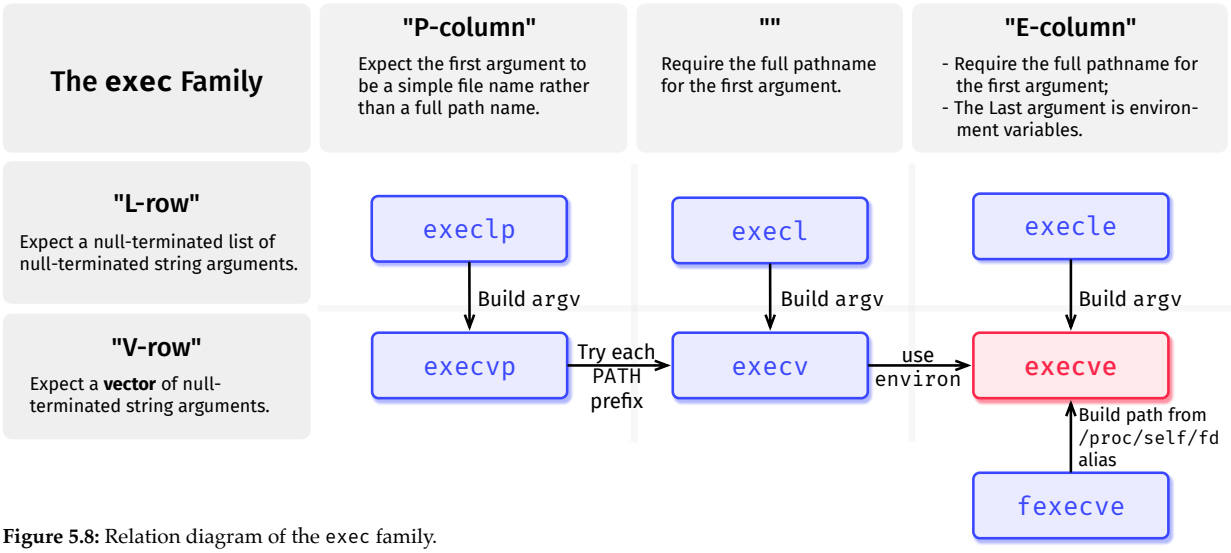


Figure 5.8: Relation diagram of the `exec` family.

5.3 Executing Programs

As mentioned in Section 5.1.4, to create such a hierarchy in the system, the first step is to `fork()` a child or multiple child processes from the very first process `init`. The second step is to start executing different programs within those child processes. Most programs are already compiled and exist as executables, so how do these child processes invoke these programs? For example, if we want to invoke a binary executable `/usr/bin/sort`, in bash or on terminal it's easy: just type `sort` with its arguments. But what if we want to invoke this binary *within* a program?

Linux has a family of functions with `exec` as their prefix, and that's exactly what we need.

Figure 5.8 shows the relations of all functions in the `exec` family. Even if there are so many different variations, they are all just wrappers for one function: `execve()` system call; they all eventually call `execve()` to start executing a program.

The following snippet lists all the function declarations in `exec` family:

```c
#include <unistd.h>
extern char** environ;

/* Requires a pathname */
int execl (const char* path, const char* arg, ...);
int execv (const char* path, char* const argv[]);
int execle(const char* path, const char* arg,...,
           char* const envp[]);
int execve(const char* path, char* const argv[],
           char* const envp[]);

/* Requires a filename */
int execvp(const char* file, char* const argv[]);
```

```
14 int execlp(const char* file, const char* arg, ...);
```

5.3.1 Suffix p

The first parameter for all those functions refers to the binary we want to execute. If the function name ends with suffix `p`, we only need to provide the binary name, such as `echo` or `ls`; the function will look for all directories defined in environment variable `PATH` and see if the binary exists anywhere there.

If the function name doesn't have suffix `p`, we would have to provide a complete path to the binary, such as `/bin/echo`, or `/bin/ls`.

Note the first parameter has to be a binary file (such as `a.out` or a command), or a script starting with a line of the form:

```
1 #!interpreter [optional-arg]
```

For example, the bash script we have learned can also be executed here.

5.3.2 Passing Vector vs Passing List

The next suffix, either `l` or `v`, indicates how we pass the arguments to the binary executable.

- `l`: a list of arguments. That is, put the arguments as individual strings when calling these functions.
- `v`: a vector/array of arguments. When we use these functions, put all argument strings into a `char*[]`, and only pass this array.

Let's look at the following example to see how they are different. Assume the file we want to execute is `/bin/echo`:[8]

```
1 char* arguments[] = {"echo", "hello", "world", NULL};
2 execl("/bin/echo", "echo", "hello", "world", NULL);
3 execv("/bin/echo", arguments);
```

Also notice that when using either `char*[]` or a list as the argument array, we have to put a `NULL` at the end.

The last suffix indicates that if the last argument is an environment variable array or not. If it is, then add "`e`" at the end; otherwise don't.

Example 5.1: Using `execv()`

Let's write a program that invokes `echo`, and its arguments are from the command-line arguments `argv[]`.

8: Notice that the first element in `arguments` by convention should always be the name of the file we want to execute.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[], char* envp[]) {
  if (argc < 2) {
     printf("Usage: %s arg1 [arg2 ...]\n", argv[0]);
     exit(EXIT_FAILURE);
  }
  execv("/bin/echo", argv);
  fprintf(stderr, "execv() failed to run.\n");
  exit(EXIT_FAILURE);
}
```

We chose `execv()` function in the example above. The following table shows similar ways to invoke `echo` as in line 9 of the demo:

Wrapper	Call
`execv()`	`execv("/bin/echo", argv);`
`execvp()`	`execvp("echo", argv);`
`execl()`	`execl("/bin/echo", argv[0], argv[1], argv[2]);`
`execlp()`	`execlp("echo", argv[0], argv[1], argv[2]);`

In the table, `execv()` and `execvp()` are equivalent. The `l`-versions – `execl()` and `execlp()` – are also equivalent, but with limitations: because we need to pass the arguments individually as a list, we have to make sure there are three command-line arguments, *i.e.*, `argc == 3`. When this is true, all four function calls will achieve the same result.

Therefore, which version of the `execxx()` function to use depends on situation: which one is more convenient, and which one will not cause run-time errors.

5.3.3 Process Image Replacement

When running the program in the example above, you must have noticed that after `echo`, the process terminates without calling the function `fprintf()` on line 10. This seems a bit counter-intuitive at first, because we all know that a function is supposed to return back to the calling point, so why doesn't `execv()` return?

This is because when we went to `execv()`, **the process image of our program was completely overwritten by the process image of `/bin/echo`.** Therefore, the termination of `/bin/echo` is also the termination of our own program: it will **not** return to the calling point and keep executing our code.

But what about this? Can you figure out why our program didn't terminate after `execv()`?

Listing 5.6: An exampmle using `fork()` to create a child process to execute binaries.

```c
/*** execdemo.c ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
    pid_t pid;
    if (argc < 2) {
        printf("Usage: %s arg1 [arg2 ...]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((pid = fork()) == 0) {
        execv("/bin/echo", argv);
        fprintf(stderr, "execv() failed to run.\n");
        exit(EXIT_FAILURE);
    }
    printf("Success!\n");
    exit(EXIT_SUCCESS);
}
```

5.3.3.1 Redirection

We have discussed about redirection on terminal level in Chapter 1. For example,

```
$ ls > listing
```

will write all the output from `ls` command to a file called `listing`. But how does the shell make this redirection happen? As we know, which destination to write to—a file or the terminal—is determined completely by the program itself. That is, if the program uses `fprintf()` with file descriptor 1, it will print to `stdout`; if it is with file descriptor X, it prints to the file represented by X. Unless we modify the program itself by changing the destination of a file descriptor, it's impossible to make redirection happen. However, programs such as `ls` have already been compiled, so it's impossible to change their file descriptors either.

The key to a successful redirection relies on two facts:

1. System command programs always print to the file descriptors of 1 and 2, which by default represent `stdout` and `stderr`;
2. When `execxx()` function is called, the process image will be replaced entirely by the binary we're going to execute, but **only** the user-addressable portion. The kernel-addressable portion stays the same.

Now, these two facts give us a "loophole" that we can take advantage of. Recall that file descriptor table is in the kernel-addressable portion. What

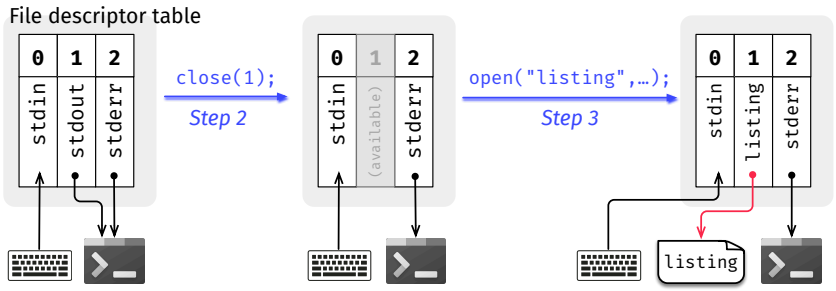


Figure 5.9: Closing `stdout` makes file descriptor 1 available, so `open()` will take over that file descriptor and assign it to the newly opened file.

if before calling `execxx()`, we actually changed the destination of file descriptor 1 from `stdout` to a file?

The following steps exactly describes how the shell implements redirection:

1. `fork()` a new process;
2. In the new process, `close()` file descriptor 1 (`stdout`) so that it becomes available;
3. In the new process, `open()` (with `O_WRONLY | O_CREAT | O_TRUNC` flag) the file named `listing`;
4. Let the new process `exec()` the `ls` program.

Step 2 and 3 are illustrated in Figure 5.9, where we show how the file descriptor table has been changed in `ls` process.

5.3.4 The `system()` Function

If `exec` family seems too confusing, here's an easy solution:

```
#include <stdlib.h>
int system(const char* command);
```

You can just write the command you'd put into the terminal as a string, and pass it to `system()`. To see the difference between `execxx` and `system()`, let's try the following program:

Listing 5.7: An example of using `system()` function.

```
/*** execsystem.c ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
    //system("/usr/bin/ls -a ./");
    execl("/usr/bin/ls", "ls", "-a", "./", NULL);
    fprintf(stderr, "execl() failed to run.\n");
    exit(EXIT_FAILURE);
}
```

Comment out line 6 or 7 at a time to see what output you get. This is the first difference you'd find.

> *From manpage,* `https://man7.org/linux/man-pages/man3/system.3.html`.
>
> ---
>
> The `system()` library function uses `fork(2)` to create a child process that executes the shell command specified in `command` using `execl(3)` as follows:
>
> `execl("/bin/sh", "sh", "-c", command, (char *) NULL);`
>
> `system()` returns after the command has been completed.

Now, let's try the following code and see what we'll get:

```c
/*** execsystem2.c ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
    system("/usr/bin/echo $PATH");
    execl("/usr/bin/echo", "echo", "$PATH", NULL);
    fprintf(stderr, "execl() failed to run.\n");
    exit(EXIT_FAILURE);
}
```

Listing 5.8: An example of executing bash/shell commands.

When we use `system()` to run the command, it outputs the `PATH` correctly, while `execl()` doesn't. This is not because `echo` failed to do its job: the job of `echo` is simply output what you put after it. When we use `system()`, it actually uses an **interpreter** to expand the argument with leading `$`. This interpreter is a program `/bin/sh`, aka the shell, the old plain version of bash we use in terminal by default. Therefore, if we want to use `execl()` to display environment variables, we need to invoke `/bin/bash` instead of `/bin/echo`:

```c
execl("/bin/bash", "bash", "-c", "\"echo $PATH\"", (char*)
 ↪  NULL);
```

5.4 Organization of Processes

5.4.1 Background and Foreground Processes

Most of the programs we run are **foreground processes**, or more accurately, **controlling foreground processes**, meaning we will wait until they finish the task. A simple example is when we use `apt-get` to install a

software in Ubuntu — while the installer is running, we need to wait, and when it has finished we can type next command. These processes **control** the terminal, to simply put.

Sometimes, however, we want to just start a process, and leave it to the background, so that they don't dominate the terminal window and we can continue to work in it. Those are called **background process**.

We can control the processes on terminal: either push them back to the background, or bring them up as a foreground process. See the following example:

```c
int main() {
    while (1);
}
```

Apparently this code has an infinite loop, and if we run it normally, it'll be a foreground process, and will take up the terminal forever. Assume the binary generated is called `a.out`, we can use the following command to start this program as a background process:

```
$ ./a.out &
```

Notice the ampersand after `a.out`. Then you will see an output like this:

```
[1] 109468
```

Here `1` is the job number, while `109468` is the PID. We can use `jobs` command to see all the background jobs.

If later on you want to bring a job back to foreground (maybe to check its status), you can use `fg` command. Assume the job we want to bring back has a job number `10`, this is what we can do:

```
$ fg 10
```

To make it go back to the background, we need to first use `Ctrl+Z` to "stop" it. This doesn't really kill the program; instead, it puts the job on pause and sends it back to the background. [9] Then we can use the `bg` command to resume it in the background:

```
$ bg 10
```

9: To really kill it, use `Ctrl+C`. We will learn this pretty soon in Section 5.6.2.1

5.4.2 Groups and Sessions

For each process, in addition to PID, PPID, *etc.*, there's also a Process Group ID (PGID) indicating the **process group** this process *uniquely* belongs to.

In a process, we can use the following function to retrieve the PGID of the group it belongs to:

```
#include <unistd.h>
pid_t getpgrp();
```

This function will return process group ID of the calling process.

The following example shows us how to get PGID:

```
/*** pgroup.c ***/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t children[5];
    for (size_t i = 0; i < 5; i++) {
        if ((children[i] = fork()) == 0) {
            printf("I'm child %d in group %d.\n",
                   getpid(), getpgrp());
            exit(EXIT_SUCCESS);
        }
    }
    printf("I'm parent %d, in group %d\n",
           getpid(), getpgrp());
    exit(EXIT_SUCCESS);
}
```

Listing 5.9: Using `getpgrp()` to get PGID.

What did you notice from the output of the code above?

We can also use `setpgrp()` function to change the PGID of a process:

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

A group of processes is called a process group; then a group of process group is called a **session**. Of course, each session also has a unique ID called session ID (SID), which can be retrieved by:

```
#include <unistd.h>
pid_t getsid(pid_t pid);
```

Usually when we start a terminal, the process of this terminal creates a session. [10]

10: We can surely create sessions inside a program, but it's beyond the scope of our course.

> *From The Linux Programming Interface, Kerrisk, pp.700.*
>
> ---
>
> All of the processes in a session share a single **controlling terminal**. The controlling terminal is established when the session leader first

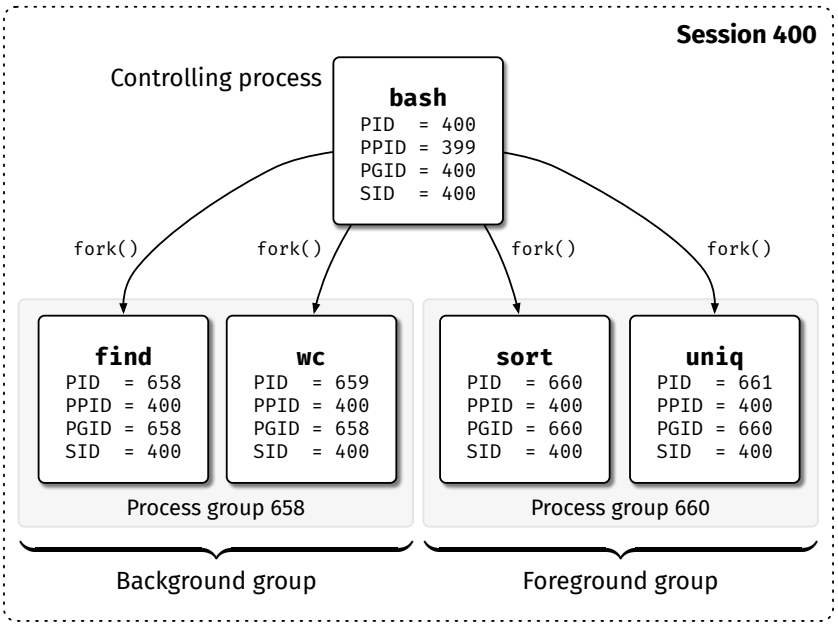


Figure 5.10: The relations among background/foreground processes, sessions, and groups.

opens a terminal device. A terminal may be the controlling terminal of at most one session.

In other words, when we open a new terminal window, it automatically launches bash. This bash process becomes the session leader, and its PID becomes the session ID.

5.4.3 Relations

This section is borrowed from *The Linux Programming Interface, Kerrisk, pp.701*. We use Figure 5.10 to show the relations among background/foreground processes, sessions, and groups. Assume we have the following sequence of commands entered to a terminal:

```
$ echo $$
400

$ find / 2> /dev/null | wc -l &
[1] 659

$ sort < longlist | uniq -c
```

The first command is used to show the PID of the current bash running in the terminal. This bash is the controlling process of the session and the session leader, and its PID (assume `400`) is also the session ID.

We then entered our second line of command, which used a `|`: a pipe operator. The standard output of `find` will be sent to `wc`, while the standard error will be discarded (because of the use of `2> /dev/null`). Notice at the end we have an ampersand, meaning both programs will run

as background processes. Since these two processes are related through the pipe operator, both form a process group, and its leader is the program from the first command `find`. Notice both of them have PPID of `400`, meaning they are both child processes from the bash. After the second command, because we made them background processes, they gave up controlling of the terminal, and handed the control back to bash.

The third command also creates a group, but this time we don't have `&` at the end, so they become the foreground processes, and take up the terminal.

5.5 Processes & File Descriptions

The relations between file descriptors and processes can be a bit tricky and confusing, so we'll use this section to sort them out.

For all the opened files, the kernel maintains a **system-wide** open file table, where each element is a file description object. When a new file is opened, a new object is created and added to the table; when a file is closed, it'll be destroyed.

This open file table is system-wide, meaning all the processes share this table. However, for each process, we have also learned that there's a file descriptor table that keeps track of all opened files in that process. We use Figure 5.11 to describe the relations between all these things.

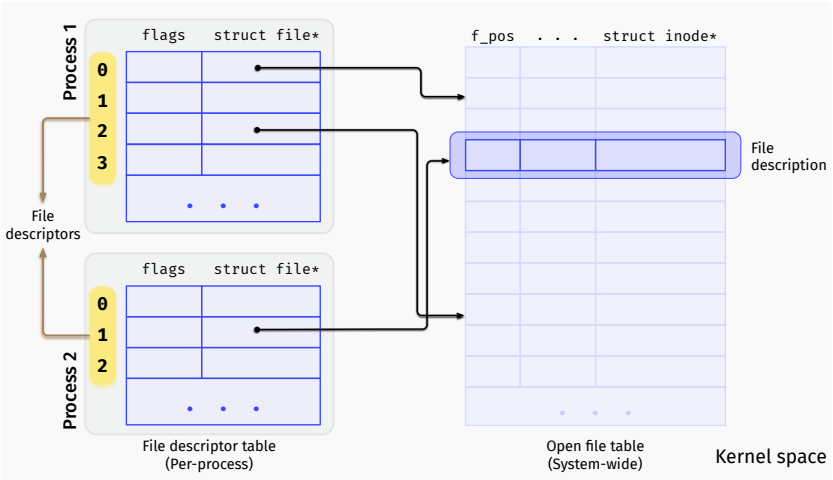


Figure 5.11: Relations between file descriptor tables and open file table.

You can examine the file descriptor table by checking out `/proc/<PID>/fd/`.

5.5.1 Single Process

In a single process it's possible that we open the same file multiple times. Each time we call `open()` function to open the same file, the open file table will create a new file description, and a new file descriptor will be

returned. Thus, even if there are multiple file descriptors working on the same file, each of them tracks the file offset `f_pos` individually. See Figure 5.12 for an illustration.

What this implies is, when we invoke any system call that can change `f_pos` (such as `read()`, or `lseek()`, *etc*) on a file with different file descriptors, they will have their own `f_pos`. See the following example.

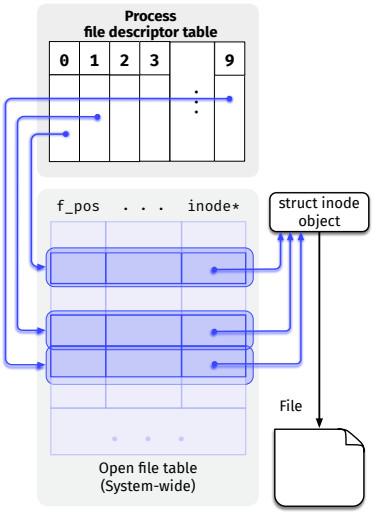


Figure 5.12: When a process opens the same file multiple times, the system will create a separate file description for each of them. Each returned file description will keep track its own offset in the file and will not affect others.

Example 5.2: Same file opened twice

Assume all the function calls succeed and are executed as expected. Suppose that `montymole.txt` exists and was empty before running this block of code. The following code was run:

```
1  int main(int argc, char** argv) {
2      int fd1 = open("montymole.txt", O_WRONLY);
3      int fd2 = open("montymole.txt", O_WRONLY);
4      write(fd1, "mole",  4);
5      write(fd2, "whack", 5);
6      write(fd2, "mole",  4);
7      write(fd1, "mole",  4);
8      write(fd1, "mole",  4);
9      close(fd1);
10     close(fd2);
11 }
```

Which of the following could be the content of `montymole.txt`?

- ✓ **`whacmolemole`**
- ✗ `whacmolek`
- ✗ `molewhackmolemolemole`
- ✗ `whackmolemole`
- ✗ Nothing

Solution: In this example, we opened the file twice, and therefore have two file descriptors to perform I/O on the same file. As discussed before, each file descriptor tracks its own file offset, and `write()` starts operating on the offset of the file descriptor we passed as the first argument.

5.5.2 Unrelated Processes

We know that the same file can be opened by unrelated processes. If each of the two processes called `open()` on the same file, the open file table will create two entries for each of them. What this implies is each process will have its own file description, and so more importantly, `f_pos`, the file offset. When one process calls `write()` on the file, only `f_pos` that belongs to that process will advance, while `f_pos` in another process stays.

One thing to be careful about in this case is that remember processes are scheduled by the kernel individually. If two processes are writing to the same file, there'll be uncertainty about what the file looks like in the end because we can't predict which process will execute `write()` first. We will show an example soon.

5.5.3 Parent-Child Processes

We learned that calling `fork()` will create child process which is a complete copy of the parent process. What's more important is that the child process also copies the file descriptor table.

This is all true, but we need to be careful: If a file is opened **before** creating a child process, both the parent and the child share one file description, and therefore `f_pos`. It is not difficult to understand this, based on the structure of file descriptor tables. The child copies the table from the parent, meaning it also copies the pointer to the file description, so naturally both parent and the child are using the same file description. See Figure 5.13 for an illustration.

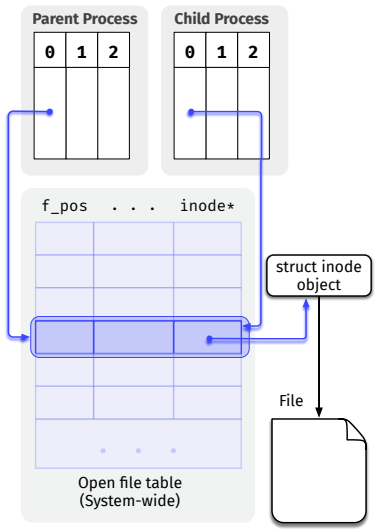


Figure 5.13: Because child process copies the file descriptor table from the parent, it'll share the file description and therefore file offset as well.

However, if both the parent and the child opened the same file **after** the child process is created, each of them will have a separate file description, because now they are basically two separate processes, and one process' `open()` does not affect the other one's.

What this implies is we need to be careful about all the functions that change `f_pos`: `read()`, `write()`, and `lseek()`, *etc.*

Example 5.3: Same file opened by multiple processes

Assume all the function calls succeed and are executed as expected. Suppose that `montymole.txt` exists and was empty before running this block of code. The following code was run:

```c
int main(int argc, char** argv) {
    int   fd1 =  open("montymole.txt", O_WRONLY);
    pid_t p   =  fork();
    if (p == 0) write(fd1, "whack", 5);
    else write(fd1, "mole", 4);
    write(fd1, "mole", 4);
    close(fd1);
}
```

What are the possible content of `montymole.txt`?

Solution:
As discussed above, `fork()` function is called *after* opening the file, so both parent and child will share file description and file offset. To make it clear, we write out the I/O operations performed by both parent and child and label them with P or C:

	Parent		Child
P1	`write(fd1,"mole",4);`	C1	`write(fd1,"whack",5);`
P2	`write(fd1,"mole",4);`	C2	`write(fd1,"mole", 4);`

These two processes will run individually, so all the following outcomes of execution are possible:

- P1 → P2 → C1 → C2: `molemolewhackmole`
- P1 → C1 → C2 → P2: `molewhackmolemole`
- P1 → C1 → P2 → C2: `molewhackmolemole`
- C1 → C2 → P1 → P2: `whackmolemolemole`
- C1 → P1 → P2 → C2: `whackmolemolemole`
- C1 → P1 → C2 → P2: `whackmolemolemole`

Now what if we swap line 2 and 3? That is:

```
1 pid_t p   = fork();
2 int   fd1 = open("montymole.txt", O_WRONLY);
```

Note that in this case, the I/O operations performed by the parent and child are still the same, as well as the execution sequence we enumerated above. However, since `fork()` is called *before* we open the same file, the parent and the child will use two separate file descriptions, and they do not share file offset anymore. What could be the content then?

5.6 Signals

Think signals as the notification on your phone. Say you want to upload a video to YouTube ▶. You don't want to wait for it to complete the whole time, right? So you put it to the background and hand it to the "kernel". When the job is done, or something wrong with the upload, or any other incidents, you will receive a notification (a signal).

5.6.1 General Concepts of Signal

To put it formally, a **signal** is a small message that notifies a process that an event of some type has occurred in the system. It's similar to exceptions and interrupts. The signals can be sent from the kernel to a process, sometimes at the request of another process.

5.6.1.1 System Standard Signals

There's a set of 32 standard signals defined in `<signals.h>`.[11] Each signal is an integer between 1 and 32, indicating a specific and pre-defined system event. For example, when we're trying to dereference an invalid

11: Modern systems extend this to 64, but we don't need to consider those.

pointer, the signal `SIGSEGV` (segmentation violation) will be generated and delivered to our process, causing segmentation fault.

Each signal has been defined with a macro name, all starting with `SIG`. For the same signal macro, different systems might use different integers, so you should **always** use signal macro instead of integer value, for the purpose of portability and readability.

5.6.1.2 Sending & Receiving Signals

Kernel **sends** (delivers) a signal to a **destination process** by updating some state in the context of the destination process. It sends a signal for one of the following reasons:

- Kernel has detected a system event such as divide-by-zero or the termination of a child process;
- Another process has invoked the `kill()` system call to explicitly request the kernel to send a signal to the destination process.

5.6.1.3 Signal Disposition

When a signal has been delivered to the destination process, the process has to handle it. Each signal has a pre-defined action called **disposition**, *i.e.,* how the process will handle the signal by default. There are five default dispositions:

- ignore;
- terminate;
- terminate and dump core;
- stop or pause the program;
- resume a program paused earlier.

5.6.1.4 Pending & Blocked Signals

A signal is **pending** if sent but not yet received. There can be at most one pending signal of any particular type, because signals are not queued. What this means is, if a process has a pending signal of type, say `SIGINT`, then subsequent signals of type `SIGINT` that are sent to that process are discarded.

A process can also **block** the receipt of certain signals. Blocked signals can be delivered, but will not be received until the signal is unblocked.

The rest of the section will visit those general concepts in details/ To thoroughly understand signals, manpage (`https://man7.org/linux/man-pages/man7/signal.7.html`) is always a good and informative read.

Table 5.1: Default keybindings for sending signals.

Keybinding	Signal	Default action (Disposition)
`Ctrl+C`	`SIGINT`	terminate
`Ctrl+Z`	`SIGTSTP`	stop or pause the program
`Ctrl+\`	`SIGQUIT`	terminate and dump core

5.6.2 Sending Signals

At this point, we have briefly mentioned that some keybindings can actually send certain signals to a process. For example, `Ctrl+C` sends `SIGINT`. In this section we will discuss in detail how to send signals.

5.6.2.1 Key-Binded Signals

Say you accidentally wrote a program that has an infinite loop. If you remember, what you did in this case is to press `Ctrl` key and `C` key together, and then the program was then terminated. In fact, when we are pressing `Ctrl+C`, it sends signal `SIGINT` to the process, and the default action of `SIGINT` is to terminate the program. That's why `Ctrl+C` terminates the program.

There are three keybindings that can send signals to the process as shown in Table 5.1. You should be fairly familiar with this table.

Try it

Here's one small experiment worth trying: write a C code with an infinite loop, and start executing it. Then try different keybindings listed in Table 5.1 and see what terminal outputs.

5.6.2.2 Using `htop`

The keybinding method for sending signals is very limited to the three signals mentioned above, so how can we send arbitrary signals to a process?

On a TUI, we can use the interface provided by `htop`. After invoking `htop` (Figure 5.3), choose a destination process we want to send a signal to, and click on the button "Kill" at the bottom, or press F9, and we will see a list of signals we can send on the left, as in Figure 5.14.

In the left panel, the integer represents the signal's actual value, and the macros defined in the system are next to the numbers. On different systems you might notice different integer values for the same macro.

5.6.2.3 Using `kill` Command

Without TUI, we can use the `kill` command to send a signal. Assume the PID of a process is `1648`, and we want to send `SIGKILL` signal to it. We can use the following command:

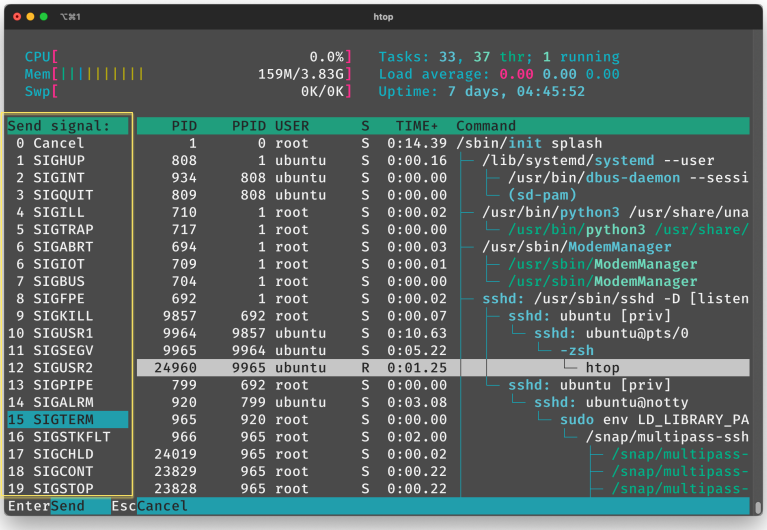


Figure 5.14: Sending signals through `htop` command.

```
$ kill -SIGKILL 1648
$ kill -KILL    1648 # both SIGKILL and KILL work!
$ kill -9       1648 # can also use the integer number
```

As mentioned earlier, the same integer on different systems might represent different signals, so it is good to check before using the command, or just use the macro names for portability and readability.

Now you should know that the command `kill` is **not** really to *kill* programs, although most of the time the signal it sends terminates the program by default. It simply sends a signal to a process. In fact, you can also send signals in your process by calling `kill()`.

5.6.2.4 Using `kill()` Function

Just like other commands, there's a corresponding C function called `kill()` that we can use to send signals in our program:

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

On success (at least one signal was sent), 0 is returned. On error, -1 is returned, and `errno` is set to indicate the error.

Another function is called `raise()`:

```
#include <signal.h>
int raise(int sig);
```

which sends a signal to the *executing* process (*i.e.*, the current process). Therefore, the following two statements are equivalent:

```c
raise(SIGINT);
kill(getpid(), SIGINT);
```

5.6.3 Altering Default Actions

Default actions of most signals can be altered by us, except `SIGKILL` and `SIGSTOP` which cannot be caught, blocked, or ignored. Before we go into the details, let's have a quick look at a simple example.

Example 5.4

In this example, we have an infinite loop that prints an integer at a time. When pressing `Ctrl+C`, we want the program print "Ouch! Stop it!" and then go back to the infinite loop.

Listing 5.10: Altering default action of signals.

```c
/*** sig1.c ***/
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void ouch(int sig) {
    printf("\nOuch!\n");
    sleep(1);
    fflush(stdout);
    printf("Stop it!\n");
}

int main() {
    /* declare a struct sigaction */
    struct sigaction action = {0};
    /* set the handler */
    action.sa_handler = ouch;
    /* Install the signal handler */
    sigaction(SIGINT, &action, NULL);

    for (size_t i = 0; ; i++) {
        printf("%d\n", i);
        sleep(1);
    }
    return 0;
}
```

Normally, when we press `Ctrl+C` to a running program, `SIGINT` will be sent to the process from the kernel and our process will be terminated. In the demo code above, however, we choose not to terminate the program; instead we created a function to respond to the signal, *i.e.*, `ouch()`. A possible output might look like this:

```
0
1
^C
Ouch!
Stop it!
2
^\\[1]    2017 quit       ./a.out
```

We use `Ctrl+C` to send the signal, and use `Ctrl+\` to terminate the terminal output.

5.6.3.1 Installing Signal Handlers

Notice that in the example, we take several steps before going into the infinite loop. First, we declare an object of `struct sigaction`. This struct contains all the information about how we want to change the action. The definition of `struct sigaction` is as follows:

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t*, void*);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

The member variable `sa_handler` is the most important one: it's a function pointer pointing to a **signal handler**, which is the function we want to invoke upon receiving the signal we specify later. Based on our knowledge about function pointers, we have to define the signal handler as a function that receives one integer and returns void. Function `ouch()` in our example is the signal handler, so on line 17 we set up the signal handler. Other member variables are beyond the scope of our course, but feel free to explore if you're interested. [12]

Once `struct sigaction` object has been set up, we can call `sigaction()` to **install the signal handler**. What this means is the function that handles the signal, *i.e.*, signal handler, will be registered in the kernel, so that later on when the corresponding signal was received, the kernel can call the signal handler, and pass the received signal as the parameter to the signal handler. The function prototype is declared as follows:

```
#include <signal.h>
int sigaction(int   signum,
              const struct sigaction* restrict act,
              struct sigaction* restrict oldact);
```

12: Because on some versions of UNIX `sa_handler()` and `sa_sigaction()` are defined as an `union` type, do not assign both of them at the same time. Just choose one you'd like to use. But how does the kernel know which one you choose to use? It looks for `sa_flags`. Setting `SA_SIGINFO` in `sa_flags` will cause the use `sa_sigaction()` instead of `sa_handler()`. The `sa_flags` can also be OR'ed with other macros to specify how we want to handle the signal. Those options can be found at https://man7.org/linux/man-pages/man2/sigaction.2.html. Lastly, `sa_mask` specifies a set of signals which should be blocked while the signal handler is executing.

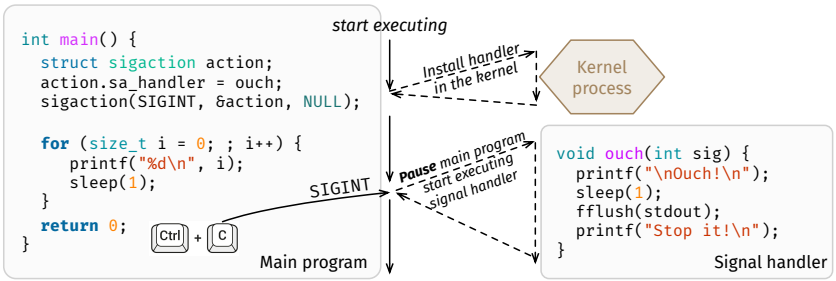


Figure 5.15: Program flow of Listing 5.10.

`signum` specifies the signal we want to deal with, and can be any valid signal except `SIGKILL` and `SIGSTOP`. If `act` is non-NULL, the new action for signal `signum` is installed from `act`. If `oldact` is non-NULL, the previous action is saved in `oldact`.

Now that we have a signal handler, when we press `Ctrl+C` again, we see our process was not terminated; instead it triggered and executed the signal handler `ouch()`, and then kept running the loop. We depict the flow of Listing 5.10 in Figure 5.15.

5.6.3.2 Restoring Signals

When introducing `sigaction()` above, we saw that the last argument of the function is where we save the old action. The following example shows us how to use this feature. What it does is to alter the default action of `SIGINT` by using `sig_handler()` for the first 10 seconds. Afterwards, we still want use `Ctrl+C` to end the program, so we use `sigaction()` to install the old handler stored in `old`.

Listing 5.11: An example of restoring old signal handler.

```c
/*** sigrestore.c ***/
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
void sig_handler(int sig) {
    printf("\nOuch!\n");
    sleep(1);
    fflush(stdout);
    printf("Stop it!\n");
}

int main() {

    /* declare a struct sigaction */
    struct sigaction new;
    struct sigaction old;
    /* set the handler */
    new.sa_handler = sig_handler;
    /* Install the signal handler */
    sigaction(SIGINT, &new, &old);
```

```
22
23      for (size_t i = 0; i < 10; i++) {
24          printf("%d\n", i);
25          sleep(1);
26      }
27
28      /* Restore old handler */
29      sigaction(SIGINT, &old, NULL);
30
31      while(1);
32      return 0;
33  }
```

5.6.4 Properties of Signals

5.6.4.1 Parent and Child

If we'd like to catch different types of signals, we only need one signal handler function, but need to install all of them. Note that it is not possible to install multiple handlers for the same signal. If this happens, the last installed handler for that signal will be registered.

Let's look at the following example.

```
1   /*** sigchild.c ***/
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <signal.h>
5   #include <unistd.h>
6   void ouch(int signum) {
7       if (signum == SIGINT) {
8           printf("Ouch!\n");
9       }
10      else if (signum == SIGCHLD) {
11          printf("Woof!\n");
12          exit(EXIT_SUCCESS);
13      }
14  }
15
16  int main() {
17      pid_t pid;
18
19      /* Installing signal handler */
20      struct sigaction setup_action = {0};
21      setup_action.sa_handler =  ouch;
22      sigaction (SIGINT, &setup_action, NULL);
23      sigaction (SIGCHLD,&setup_action, NULL);
24
25      if ((pid = fork()) == 0) {
26          printf("I'm the child %d\n", pid);
27          sleep(2);
28          printf("I'm done!\n");
```

Listing 5.12: Communiating through signals between parent and child processes.

```
    }
    else while (1);

    return 0;
}
```

You can see when the child process terminates, our main process (the parent) also terminates. This is because the child sends out the signal `SIGCHLD` when it finishes, and our signal handler deals with this signal by calling `exit()`. In fact, child processes inherit all the handlers from the parent. [13]

13: It's a good idea to review Section 5.2.2.

Another interesting experiment you can do is that when the child process is sleeping, press `Ctrl+C` on terminal. You will see that "Ouch!" is printed twice. This is because the signal has been delivered to every process in the same process group (Section 5.4.2). In this case, one "Ouch!" is printed by the parent, while the other by the child.

5.6.4.2 Pending Signals

When main process receives a signal, it'll go to signal handler to deal with it. But what if during the execution of signal handler another signal arrives?

Listing 5.13: A demo showing the queuing of multiple signals arrived.

```
/*** sigqueue.c ***/
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

void catch_stop(int sig) {
    const char* sname = strsignal(sig);
    printf("\n");
    for (int second = 5; second >= 0; second --) {
        printf("[sig%s] %d\n", sname, second);
        sleep(1);
    }
}

void install_handler() {
    struct sigaction setup_action;
    setup_action.sa_handler =  catch_stop;
    setup_action.sa_flags   =  0;
    sigaction (SIGINT, &setup_action, NULL);
    sigaction (SIGQUIT,&setup_action, NULL);
}

int main() {
    install_handler();
    while (1);
    return 0;
```

```
}
```

While the handler deals with `SIGINT`, we can trigger `SIGQUIT` by pressing `Ctrl+\`, and see what happens. You might get an output like this:

```
^C
[sigInterrupt] 5
[sigInterrupt] 4
[sigInterrupt] 3
[sigInterrupt] 2
[sigInterrupt] 1
[sigInterrupt] 0
^\
[sigQuit] 5
[sigQuit] 4
[sigQuit] 3
[sigQuit] 2
[sigQuit] 1
[sigQuit] 0
```

Now, what I'd like you to try out is, while the process is dealing with `SIGINT`, press `Ctrl+C` as much as you can, and see what happens. What you'll see here is after the current handler for `SIGINT` is done, it'll deal with `SIGINT` again, but that's it. No matter how many times you pressed `Ctrl+C`, during the first handling of `SIGINT`, it won't process it. Our conclusion: if during the handling of signal `SIGx` multiple same signals arrived, the system will only handle one more, while the rest of them will be discarded.

What if we press `Ctrl+\` while it's handling `SIGINT`? What you'll see is, once the current handling of `SIGINT` has finished, the system will deal with `Ctrl+\` immediately.

5.6.4.3 Blocking Signals

Following the above example (Listing 5.13): what if we don't want the handling of `SIGINT` to be interrupted? We can block other signals then. As in structure `sigaction`, for each process, the kernel keeps record of a `sa_mask`, which indicates what signals should be blocked by the process **while dealing with other signals**.[14] Note that this is not to block the signal forever; once the running handler has finished, the blocked signal will be handled eventually.

14: The type `sigset_t` is simply a 32-bit integer. You can verify it by using `sizeof(sigset_t)`.

Let's replace the `install_handler()` in Listing 5.13 with the following new function:

```c
/*** sigblock.c ***/
void install_handler() {
    struct sigaction setup_action;
    sigset_t block_mask;
```

Listing 5.14: Adding blocked signals to the set using `sigaddset()`.

```
    sigemptyset (&block_mask);
    /* Block other terminal-generated signals
       while handler runs. */
    sigaddset (&block_mask, SIGINT);
    sigaddset (&block_mask, SIGQUIT);
    setup_action.sa_handler =  catch_stop;
    setup_action.sa_mask    =  block_mask;
    setup_action.sa_flags   =  0;
    sigaction (SIGINT, &setup_action, NULL);
    sigaction (SIGQUIT,&setup_action, NULL);
}
```

You can see we used several functions to set up and manipulate `block_mask` (or the signal set). They are declared as follows:

- `int sigemptyset(sigset_t* set)` : create an empty signal set;
- `int sigfillset(sigset_t* set)` : add every signal to the set;
- `int sigaddset(sigset_t* set, int signum)` : add signal `signum` to the set;
- `int sigdelset(sigset_t* set, int signum)` : delete signal `signum` from the set;
- `int sigismember(const sigset_t* set, int signum)` : check if the signal `signum` is in the set.

Inter-Process Communication 6

It is very common that processes need to transfer data between each other. However, when a process starts running, it contains a complete virtual memory address space, and all the processes are isolated in memory. In this chapter, we are going to explore methods that can transfer data among processes.

6.1 Pipes

Let's start with a simple task, where we want to count the number of files in a directory. As a reminder, command `wc` will receive input from `stdin` and print out the statistics: number of lines, number of tokens, and number of bytes. Because the output of `ls` command is separated by spaces, the number of takens is equal to the number of files.

Our first attempt is to redirect the output of `ls` to a file, and then redirect in the input to `wc` from the file:

```
$ ls ./ > temp
$ wc < temp
```

In fact in this example, the file `temp` is what we use to transfer data between processes. It is simple and straightforward, but it has several flaws:

1. Writing and reading a file on the filesystem is very expensive and inefficient. Think about the memory hierarchy we have learned in CS-382: getting access to secondary storage is 1,000 times slower than to main memory;
2. If the first command writes a lot of output, it will be extremely slow to write them to the file, while we are all just waiting for it to finish. On the other hand, if we have very limited space on secondary storage, we won't have enough space to store the file.

Based on the first flaw we mentioned above, let's think about this: what if we can find a region in the main memory where all processes can get access to? That'll make this data transfer much faster, because now the first command just need to write the data to that area in the main memory so that the second command can read from it.

We have learned that there are kernel and user spaces in the main memory. User space only contains isolated process images, so the better candidate to assigning this "common area" is in the kernel space. This buffer area in the kernel space is called a **pipe**.

There are two types of pipes in the system — unnamed pipes and named pipes. In this section we focus on unnamed pipes, and sometimes we just

call it "pipes". Next section we will focus on named pipes, which is also called FIFO.

6.1.1 Pipe Operator

On command level, we can use the pipe operator `'|'`:

```
$ ls ./ | wc
```

which connects the output of `ls` command to the input of `wc`. The `'|'` is a bash operator; it causes bash to start the `ls` command and the `wc` command **simultaneously**, and to re-direct the standard output of `ls` into the standard input of `wc`. [1]

1: `'|'` is a bash operator that uses the lower-level, underlying pipe facility of UNIX, which was invented by Douglas Mcilroy, and was incorporated into UNIX in 1973.

6.1.2 Creating a Pipe

Although a pipe may seem like a file and is even one of the file types on UNIX, it is not an actual file stored on the hard drive or file system. It is conceptually like a conveyor belt consisting of a fixed number of logical blocks that can be filled and emptied.

A pipe has two ends—read end and write end, making it **unidirectional**. We can create a pipe by calling function `pipe()`:

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

If it is successful, it returns a 0, otherwise it returns -1. What it does is creating a pipe in the kernel space, assigning two new file descriptors and connecting the read end with `pipefd[0]` and the write end `pipefd[1]`. Figure 6.1 shows an illustration.

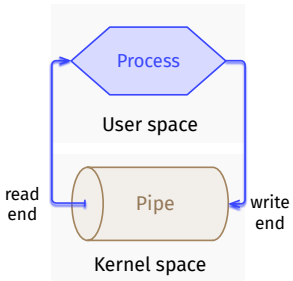


Figure 6.1: A `pipe()` call will create a pipe in the kernel space with read and write ends.

Every process can create its own pipes, one or multiple.

Even though the pipe is just a buffer area in the kernel space, the process treats the pipe as a file, and therefore the common file I/O operations, such as `write()` and `read()`, can also be applied here. However, note that `lseek()` cannot be used for pipes.

Calling `write()` on the write end `pipefd[1]` will write data to the pipe, while `read()` on the read end `pipefd[0]` will empty the data from the pipe in **first-in-first-out (FIFO)** order. The following code demonstrates this simple case.

Listing 6.1: Creating a pipe using `pipe()` function.

```
/*** pipe1.c ***/
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```c
#include <stdlib.h>
#define  READ_END   0
#define  WRITE_END  1
#define  NUM        5
#define  BUFSIZE    32
int main(int argc, char const *argv[]) {

    int  i, nbytes;
    int  fd[2];
    char message[BUFSIZE+1];

    /* Create a pipe */
    if (pipe(fd) == -1) exit(EXIT_FAILURE);

    /* Write NUM strings to the file descriptor */
    for (int i = 1; i <= NUM; i ++) {
        sprintf(message, "Hello #%2d", i);
        write(fd[WRITE_END], message, strlen(message));
    }

    printf("%d messages sent; sleeping a bit...\n", NUM);
    sleep(3);

   /* Read bytes from the pipe */
   while (!(nbytes = read(fd[READ_END],message,BUFSIZE))){
        if (nbytes > 0) {
            message[nbytes] = 0;
            printf("%s\n", message);
            sleep(1);
        }
        else break;
    }
    exit(EXIT_SUCCESS);
}
```

There are two major matters we need to discuss from the demo code above.

1. **Pipes store bytes not strings.** When we write to the pipe, we did write one string at a time. When reading from the pipe, however, as you see in the demo code, we read a chunk of data out of it, instead of a string at a time. This is because regardless of how we put things into the pipe — a string, an object, an integer, *etc* — it's all stored as consecutive bytes in the pipe without any boundary between the data. See the following excerpt from manpage:

   > *From manpage,* `https://man7.org/linux/man-pages/man7/pipe.7.html`.
   >
   > ---
   >
   > The communication channel provided by a pipe is a *byte stream*: there is no concept of message boundaries.

2. **Closing write-end when it's done.** When running the demo code,

you probably noticed that after reading everything from the pipe, the program hangs without terminating. This is because we didn't close the write-end, and so `read()` function is stuck there. If there's nothing more to write to the pipe, we should close the write-end; otherwise the read-end will think there's still more to come in the pipe and so will wait instead of terminating. See the following excerpt from manpage:

> *From manpage,* `https://man7.org/linux/man-pages/man7/pipe.7.html`.
>
> ---
>
> If a process attempts to read from an empty pipe, then `read(2)` will block until data is available. If a process attempts to write to a full pipe (see below), then `write(2)` blocks until sufficient data has been read from the pipe to allow the write to complete.
> ...
> If all file descriptors referring to the write end of a pipe have been closed, then an attempt to `read(2)` from the pipe will see end-of-file (`read(2)` will return 0). If all file descriptors referring to the read end of a pipe have been closed, then a `write(2)` will cause a `SIGPIPE` signal to be generated for the calling process.

6.1.3 Sharing a Pipe

The demo code in Listing 6.1 is only a demonstration to show how to use pipe, but it's not useful to use pipe within one process. After all we want to use pipe to create a channel between separate processes to transfer data.

Recall from Chapter 5.5 we learned that if a process opens a file before forking a child, both the parent and the child will share the same file description entry in the open file table, and thus operate on the same file. Here we can use the same strategy: create a pipe first and then fork a child. This way, the parent and the child can communicate with each other using the pipe. See Figure 6.2.

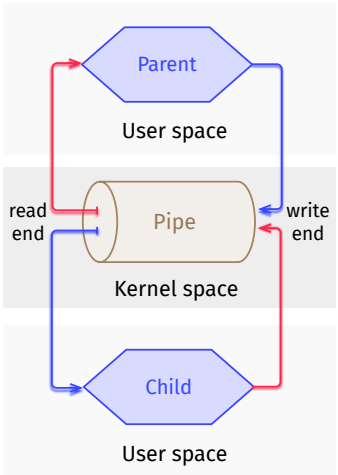


Figure 6.2: Creating a pipe and then forking a child makes both processes share the same pipe and thus enables communication.

In Figure 6.2, we noticed that because the child process copies the parent, now the pipe has two directions: from child to parent (red line), and the opposite (blue line). This is called **full-duplex**, which is not allowed in POSIX standards. Therefore, to make our code portable and stable, we should only implement **half-duplex**, meaning the pipe should only allow **unidirectional** data flow.

To achieve this, we can simply decide the direction of the data flow, and close unrelated file descriptors. For example, if we want the child to read from parent, we should close child's `fd[1]` and parent's `fd[0]`. For a half-duplex, the paradigm of program structure usually looks like this:

```c
#define  WRITE_END   1
#define  READ_END    0

...

if (pipe(fd) == -1) exit(EXIT_FAILURE);

/* child process */
if (fork() == 0) {
    close(fd[WRITE_END]); /* close write-end */
    bytesread = read(fd[READ_END], buffer, BUFSIZE);
    /* check for errors afterwards of course*/
}

/* parent process */
else {
    close(fd[READ_END]); /* close read-end */
    byteswritten = write(fd[WRITE_END], msg, strlen(msg));
}
```

If we want the data flow in both directions, we can create two pipes, and close unrelated file descriptors, as shown in Figure 6.3.

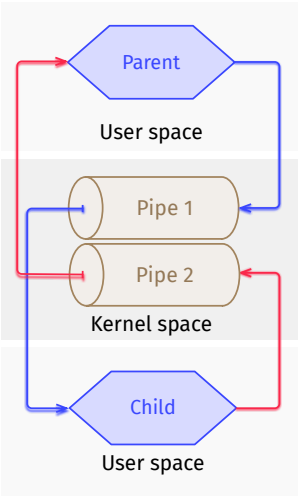


Figure 6.3: A full-duplex can be simulated by creating two unidirectional pipes.

6.1.4 Implementing the Pipe Operator

Now we're ready to implement the pipe operator used in the shell, *e.g.*,

```
$ ls ./ | wc
```

In Chapter 5.3.3.1 we introduced redirection, and how to execute a system utility command with redirected destination. A very simple solution then is to redirect the first command's file descriptor 1 to a file, and the second's file descriptor 0 from that file, as shown in Figure 6.4.

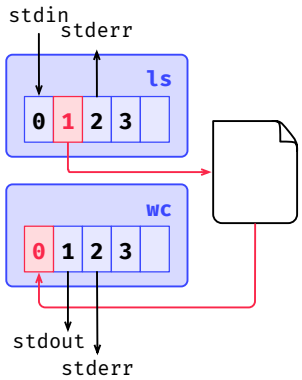


Figure 6.4: A less efficient way of implementing the pipe operator is to do a double-redirection with a file.

We have discussed the disadvantages of using a file for transferring data between processes, so a natural solution here is to replace the file with a pipe, as in Figure 6.5.

To achieve this, we'll follow the steps as below:

1. In the main program, create a pipe by calling `pipe()`;
2. Create two child processes (denoted as `C1` and `C2`), each of which will be used for a command. We assume `C1` is for `ls` and `C2` for `wc`;
3. Close all unrelated file descriptors, including:
   a) Both read- and write-end in the parent;
   b) Read-end in `C1`;
   c) Write-end in `C2`;
4. Let parent wait for both child processes;
5. Re-direct `stdout` in `C1` to the write-end of the pipe;
6. Re-direct `stdin` in `C2` to the read-end of the pipe;

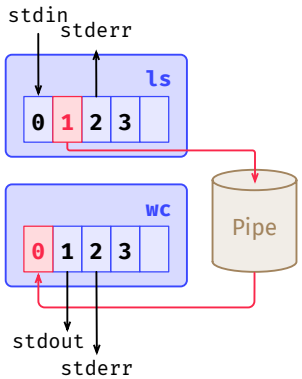


Figure 6.5: Using pipe is a more efficient way to pass day from one process to another.

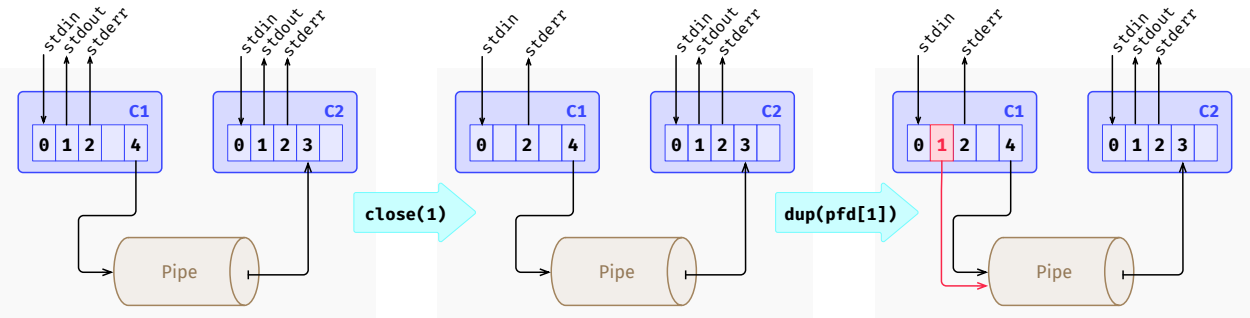


Figure 6.6: An illustration on how pipe can be implemented using `dup()` function.

7. Execute `ls` in `C1` and `wc` in `C2`.

The critical part is step 5 and 6, and we'll use step 5 as an example. We know that after creating the pipe, it already has two file descriptors associated with it. How do we let the write-end of the pipe connect to `stdout`, which itself has a file descriptor of 1? We will introduce an operation called duplicating file descriptors.

6.1.4.1 Duplicating File Descriptors

What we want to achieve here is to let file descriptor 1, which is supposed to point to `stdout`, point to the write-end of the pipe. The function we are going to use is called `dup()`:

```
#include <unistd.h>
int dup(int oldfd);
```

Given a file descriptor `oldfd`, `dup()` assigns a new file descriptor that points to the same file object as the old one pointed by `oldfd`. The critical feature of `dup()` is that it returns the lowest-numbered available file descriptor.

Assume we store the pipe file descriptors in an array called `pfd[2]`. Figure 6.6 shows us how to make this happen. In summary, the following sequence shows us the steps on `C1`'s side:

```
close(1);       // or close(fileno(stdout));
dup(pfd[1]);
close(pfd[1]);
```

After these two steps, we notice that the write-end of the pipe is pointed by two file descriptors: 1 and 3. Theoretically we can operate on both, but it is a good practice to close useless file descriptors, so we usually also close `pfd[1]`, and hence the last line above.

This method works sometimes but it has drawbacks:

- The function `dup(oldfd)` connects the lowest available file descriptor to the object pointed by `oldfd`, **not** the file descriptor we just closed. In our example, the file descriptor 1 we closed before calling `dup()` happens to be the lowest available file descriptor. What if we have the following?

```
close(0);
close(1);
dup(pfd[1]);
```

  In this case, it will be file descriptor **0** connecting to the pipe, not 1, which will not produce the expected result.

- Even if 1 *is* the lowest available file descriptor, we used two function calls to connect 1 with the pipe, which might be impacted by race conditions. We will make a detour on race conditions in the next section first to see why this drawback will cause wrong results.

6.1.4.2 Race Conditions

We know that there are multiple processes running in the system and the kernel is responsible for scheduling them individually. What this means is almost none of the processes is running back to back in one setting — at some point a process can be put on hold to let others run first. This interruption can happen at any point in a process.

The problem here, however, is if a process is currently using a resource or dealing with hardware, such as a file, interrupting it will likely lead to unwanted result or even some risks, called **race conditions**.

Now let's assume in our program we have a signal handler called `erroring()` that'll be respond to `SIGINT`:

```
void erroring(int sig) {
    int fd = open("error", O_WRONLY | O_CREAT | O_TRUNC);
    write(fd, "What's up", 9);
    return;
}
```

What happens if between `close(1)` and `dup(pfd[1])` we pressed `Ctrl+C` and this signal handler is triggered? The main program will be paused and the handler will start executing of course, where a new file is opened. Remember `open()` function also assigns the lowest available file descriptor, so at this point file descriptor 1 will be taken up by this new file. After we return from the signal handler and resume the execution of the main program, the file descriptor 1 is already not available, so `dup(pfd[1])` now will only assign 3 not 1 (as in Figure 6.6). This is certainly not something we expected or wanted.

6.1.4.3 Atomic Operations

To prevent race conditions, all the system calls are designed to be **atomic**, meaning it is not possible to interrupt when a system call is in execution until it's over. This guarantees that the operation is complete and there's no risks associated with it. This attribute is also called **atomicity**.

For example, `open()` is atomic, because until the function has finished—the file is opened, the file descriptor has been returned, *etc*, no other operations can interrupt. An operation such as `a = a + 1` cannot be guaranteed to be atomic, because between `a + 1` and writing it back to memory, CPU might let other processes come in and execute.

Based on our analysis in the previous section, if somehow we can combine `close()` and `dup()` together as one atomic operation, the race conditions will not be a factor anymore. The function `dup2()` is designed exactly for this reason:

```c
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

The function will close `newfd` if it's open, and make `newfd` point to the same file object pointed by `oldfd` pointed, as a **single atomic operation**. Therefore, `dup2()` is preferable over `close()` + `dup()`.

For example, the following two segments are equivalent:

```c
close(1);
dup(pfd[1]);
```

```c
//Using dup2
dup2(pfd[1], 1);
```

From manpage, https://man7.org/linux/man-pages/man2/dup2.2.html.

The `dup2()` system call performs the same task as `dup()`, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in `newfd`. In other words, the file descriptor `newfd` is adjusted so that it now refers to the same open file description as `oldfd`.

If the file descriptor `newfd` was previously open, it is closed before being reused; the close is performed silently (*i.e.*, any errors during the close are not reported by `dup2()`).

The steps of closing and reusing the file descriptor `newfd` are performed *atomically*. This is important, because trying to implement equivalent functionality using `close()` and `dup()` would be subject to race conditions, whereby `newfd` might be reused between the two steps. Such reuse could happen because the main

program is interrupted by a signal handler that allocates a file descriptor, or because a parallel thread allocates a file descriptor.

Note the following points:

- If `oldfd` is not a valid file descriptor, then the call fails, and `newfd` is not closed.
- If `oldfd` is a valid file descriptor, and `newfd` has the same value as `oldfd`, then `dup2()` does nothing, and returns `newfd`.

6.1.5 Pipe Capacity

Like any other I/O operations, pipes also have a capacity, meaning you can only read/write a limited amount of data if you want to make sure the operation is atomic.

On Linux, this is declared as a macro `PIPE_BUF`, which has a value of 4,096 bytes. For more information, see https://man7.org/linux/man-pages/man7/pipe.7.html.

6.2 FIFO

Recall that when we talked about pipes, we refer them to *unnamed* pipe. A FIFO is similar to a pipe, except it has a name within the file system and is opened in the same way as a regular file. This allows a FIFO to be used for communication between **unrelated** processes (*e.g.*, a client and server).

With this, we can figure that FIFO has similar attributes to regular files as well as unnamed pipes. First, the operations on FIFO is the same as regular files, meaning we can use all the functions we learned, `read()`, `write()`, *etc*. Second, it's similar to pipes in a sense that it also has a write end and a read end.

To create an FIFO, we use the following function:

```
#include <sys/stat.h>
int mkfifo(const char* pathname, mode_t mode);
```

The `mode` is the same as we used when creating a new file using `creat()` or `open()`.

6.2.1 A Simple Server-Client Example

We will use a small and simple example to illustrate how to use FIFO. Recall that FIFO allows unrelated processes to send and receive messages. So

in this example, we'll start with two parts: one is a "server" which sends a message, and the other is a "client" which receives the message.

6.2.1.1 Server

The following code shows a server, where we type a message from keyboard, and the server sends out the message.

Listing 6.2: Server code using FIFO.

```c
/*** fifo/server.c ***/
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
int main(int argc, char const *argv[]) {

    /* We usually create FIFOs under
       system directory /tmp/ */
    char* server_path = "/tmp/myfifo";
    mkfifo(server_path, 0666);

    /* Open the FIFO just like opening any regular file */
    int fd = open(server_path, O_WRONLY);
    int i;

    /* Receive a line from stdin */
    char str[1024];
    for (i = 0; read(0, str+i, 1) != -1 && str[i] != '\n';
    ↪ i++);

    /* SEND the line to FIFO */
    write(fd, str, i+1);

    /* Close FIFO */
    close(fd);

    return 0;
}
```

In the code above, we created a FIFO file called `myfifo` under the directory `/tmp/`, with the permission string `0666`. You can certainly use macros if it's more readable to you.

- `open()`
  Just like opening any regular file, we use `open()` function to open it. Notice that the O-flag we used is `O_WRONLY`, because we want to send a message to other processes, or *write* to the FIFO. Note because FIFOs are usually used for connecting different processes, there's no point to open it with `O_RDWR`, or reading and writing at the same time.

- `write()`
  After opening the FIFO, we can send any data to other processes. For the server, sending data is the same as writing data. Think FIFO

as a bulletin board, where the server sends out a message by *writing* it on the board, and the client receives it by *reading* from the board. Simply pass the file descriptor by which you opened the FIFO, and that's it.

Now, if you run this code, you'll realize that the program hangs there. No matter how many times you press enter key. Let's read this passage from man page carefully:

> *From manpage,* `https://man7.org/linux/man-pages/man7/fifo.7.html`.
>
> ---
>
> The kernel maintains exactly one pipe object for each FIFO special file that is opened by at least one process. The FIFO must be opened on both ends (reading and writing) before data can be passed. Normally, opening the FIFO blocks until the other end is opened also.

Clearly, the FIFO we created was only opened by one process, *i.e.*, the server, no data can be processed and therefore the process hangs.

6.2.1.2 Client

The client code is very similar, except that we don't need to call `mkfifo()` to create a FIFO since it's already there.

```c
/*** fifo/client.c ***/
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char const *argv[]) {

    char* server_path = "/tmp/myfifo";
    int fd = open(server_path, O_RDONLY);

    int  len = 0;
    char str[1024];

    /* Read the message sent out by the server */
    for (len = 0; read(fd, str+len,1) != -1 && str[len] !=
    ↪ '\n'; len ++);

    /* Print the message out on the terminal */
    for (size_t j = 0; j < len; j ++) write(1, str+j, 1);

    close(fd);
    return 0;
}
```

Listing 6.3: Client code using FIFO.

Now, you can compile the two C files into two binaries, and run them at the same time from two terminals. When you type something on the server

end and hit enter, you'll see it printed out on the client end on another terminal.

6.2.2 Why Is FIFO an Empty File?

In the example above, we created a FIFO special file at `/tmp/myfifo`. After finish the program, if you use `stat` command to check it out, you'll notice its size is zero. One question you might have is, we did `write()` to the FIFO, so why is it empty?

Now man page answers this question again:

> *From manpage,* `https://man7.org/linux/man-pages/man7/fifo.7.html`.
>
> ---
>
> A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the filesystem. It can be opened by multiple processes for reading or writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the filesystem. Thus, the FIFO special file has no contents on the filesystem; the filesystem entry merely serves as a reference point so that processes can access the pipe using a name in the filesystem.

Imagine we have one server but multiple clients, where the clients might send messages at the same time. How can we make sure the messages from different clients are not mixed up since there's only one FIFO? Recall that pipes, including FIFOs, have a capacity of `PIPE_BUF` which is 4,096 bytes, meaning any read/write that is under this capacity has a guarantee of being atomic. Therefore, as long as the data we send from clients are under `PIPE_BUF` bytes, we can guarantee that the `write()` from a client wouldn't be interrupted by another client's `write()`, and therefore the messages are not mixed.

Try it

We've been working around with character data (strings) for quite a while. Now let's assume we want to send a person's information from one process to another, which includes: name, age, and gender. Could you write a program?

6.3 Sockets

6.3.1 Introduction

Sockets are used like pipes in many ways but, unlike pipes, they can be used across networks. Sockets allow unrelated processes on different computers on a network to exchange data through a channel, using ordinary

`read()` and `write()` system calls. We call this **remote inter-process communication**. Formally, a socket is an endpoint of communication between processes. Although sockets are used primarily across networks, they can also be used on a single host in place of pipes.

Before moving on, we will briefly review some of the concepts in network.

6.3.1.1 Connection Types

- **Connection Oriented Model**
  When we make a telephone call to someone, we have a conversation over a dedicated communication channel—the telephone line, and we stay connected with the person on the other end for the duration of the call. In socket parlance this is called a **connection oriented model**. The connection oriented model uses the **Transmission Control Protocol**, known as TCP;

- **Connection-Less Model**
  When we have an email conversation with someone, the messages are sent to the other person across different paths, and there is no dedicated connection. In fact there is no guarantee that the messages that we send will arrive in the order we send them, and the only way for the person who receives them to know who sent them is for them to have a return address in their message header. In socket parlance this is the **connection-less model**. The connection-less model uses the **User Datagram Protocol**, or UDP.

6.3.1.2 Network Addresses

For two processes to communicate, they need to know each other's network addresses. At the level of socket programming, a network address consists of two parts: **an internet (IP) address and a port number**. The IP address, if 32 bits, consists of four 8-bit octets, and is expressed in the standard dot-notation as in `"146.95.2.131"`. These 32-bit address are known as IPv4 addresses. [2]

2: In 1995, a 128-bit address was developed, known as IPv6. Some computers have multiple network interface cards and therefore may have multiple internet addresses.

6.3.1.3 Port

Each server on a machine has to have a specific port that to use. There are many analogies that we could use, but if you think of an IP address as specifying a specific company's main telephone line, then the port is like a telephone extension within the company. The server uses a specific port for its services and the clients have to know the port number in order to contact the server. A port is a 16-bit integer.

Certain port numbers are well-known and reserved by particular applications and services. For example:

- **7**: echo servers;

- **13**: daytime servers;
- **22**: SSH;
- **25**: SMTP;
- **80**: HTTP.

Port numbers from 1 to 1023 are the well-known ports. To see a list of the port numbers in use, take a look at the file `/etc/services`. Ports 1024 through 49151 are registered ports. These numbers are not controlled and a service can use one if it is not already in use. Ports 49152 through 65536 cannot be used. They are called *ephemeral* ports, which are assigned automatically by TCP or UDP for client use.

The `lsof` command can be used to view the ports that are currently open. The command is actually more general than this: it can be used to view all open files. To see a list of open ports, use either

```
$ lsof -Pnl +M -i4
```

where the `i4` restricts to IPv4, or

```
$ netstat -lptu
```

to see listening sockets for both TCP and UDP. Read manpage for `lsof` and `netstat` to learn more about these commands.

A socket address is a combination of a network address and a port.

6.3.1.4 Domains and Protocol Families

In order for two processes to communicate, they must use the same **protocol**. Part of the procedure for establishing communication involves specifying the **communication domain** and the **protocol family**. For example, the domain `AF_INET` specifies that the protocol family is the IPV4 set of protocols. Within that family there may be a choice of a specific protocol, such as TCP or UDP. The domain might instead be `AF_UNIX`, which specifies that the protocol family is restricted to the local machine. In this case there will not be a choice of protocol. When a socket is created, the domain and protocol are specified as two of the arguments to the function. The manpage on `socket(2)` lists various domains together with possible protocols that can be used with them.

6.3.1.5 Socket Types

When a socket is created, its type must be specified. The type corresponds to the type of connection. A connection oriented communication uses stream sockets, of type `SOCK_STREAM`, whereas a connection-less communication uses datagram sockets, of type `SOCK_DGRAM`. There are also raw sockets, with type `SOCK_RAW`. Linux provides several other socket types, and POSIX requires support for the type, `SOCK_SEQPACKET`.

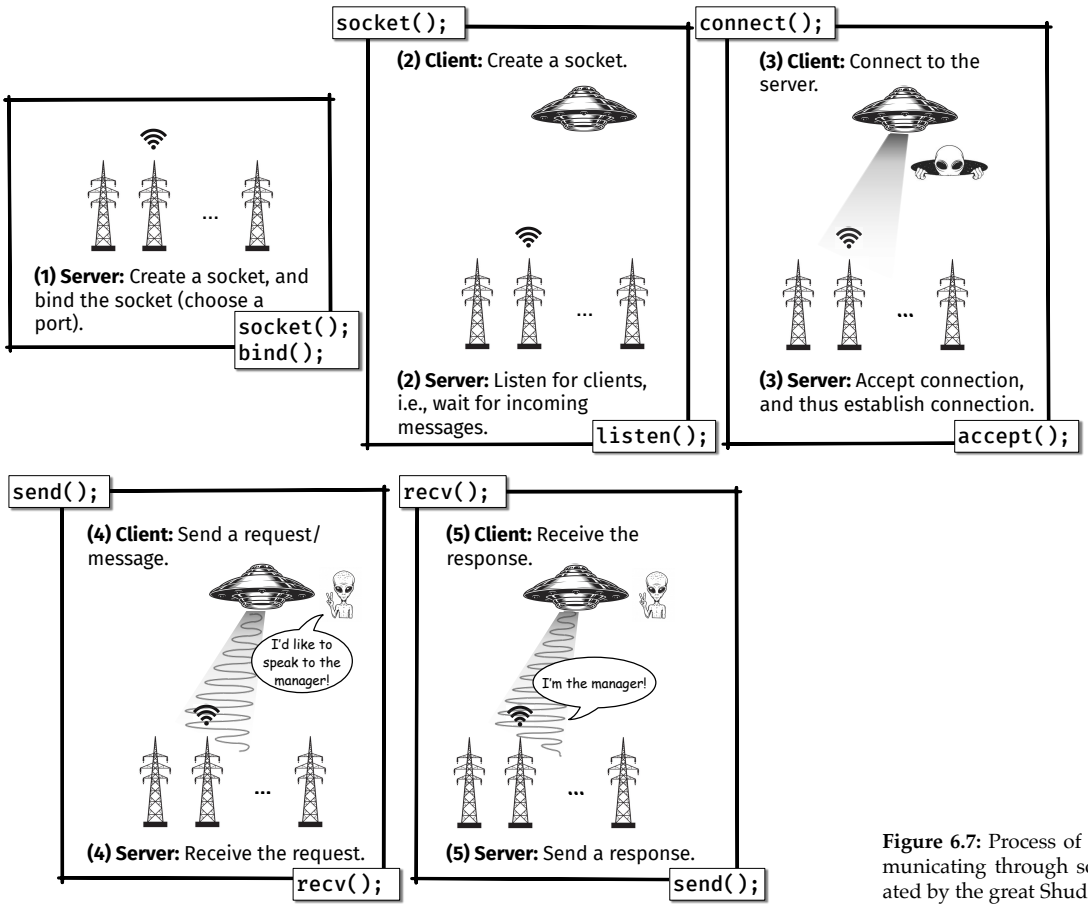


Figure 6.7: Process of creating and communicating through sockets. Comic created by the great Shudong Sensei ;)

6.3.2 Writing a Server

Writing a socket is like creating a channel for communication between two parties. One party is called the **server**, who keeps listening for incoming messages, while the other is the **client**, who sends request to the server. You can think of it this way: the server is our space station on the Earth that tries to keep listening if there's a message from aliens. The client, of course, is the aliens, and they sometimes send messages to us. We'll use this as our working example to see how to program a socket.

6.3.2.1 Typical Steps

There are typically four steps involved to create a server:

1. **Create:** create a socket using system call `socket()`;
2. **Bind:** bind the socket to a local protocol address using `bind()`. This gives the socket a "name". Now our end is the **server**;
3. **Listen:** our server is not ready, so we need to use `listen()` to monitor or listen to incoming messages. This will tell the kernel that we are ready to listen;
4. **Accept:** Enter a loop, where it repeatedly accepts new connections and processes them.

6.3.2.2 Create Sockets

The `socket()` function creates a socket and returns a file descriptor that represents it. [3] The function is declared as follows:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

`domain` is an integer specifying the address family and protocol; `type` is the socket types we mentioned above; and `protocol` can be any specific protocol, but we usually use **0** to let the system choose a proper one. In summary, arguments `domain` and `type` can usually be one of the following values:

- `int domain`
  - `AF_INET` : IPv4 (most commonly used);
  - `AF_INET6` : IPv6 (the future!!is here);
  - `AF_UNIX` : UNIX domain;
  - `AF_UNSPEC` : unspecified;
- `int type`
  - `SOCK_STREAM` : Provides sequenced, reliable, two-way, connection-based byte streams;
  - `SOCK_DGRAM` : Supports datagrams (connectionless, unreliable messages of a fixed maximum length);
  - `SOCK_SEQPACKET` : Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length.

The return value of `socket()` is a file descriptor that can be used to read or write the socket. In our example, we'll declare something like this:

```
int server_fd = socket(AF_INET, SOCK_STREAM, 0);
```

which creates a connection-oriented socket.

6.3.2.3 Bind

Before we look at `bind()` we need to see how these addresses are represented. A generic socket address is defined by the `sockaddr` structure defined in `<sys/socket.h>` :

```
struct sockaddr {
    sa_family_t sa_family; /* address family */
    char        sa_data[]; /* socket address */ };
```

3: We do not have to know how a socket is implemented to use it, however, you should think of a socket as something like the file structure that represents a file in UNIX. It is an internal structure in the kernel, accessed through a file descriptor, representing one end of a communication channel that has a specific network address, family (also called domain), port number, and socket type.

This is a generic socket address structure because it is not specific to any domain. When you call `bind()`, you will be specifying a particular domain, such as `AF_INET` or `AF_UNIX`. For each of these there is a different form of socket address structure. The address structure for `AF_INET`, defined in `<netinet/in.h>`, would be

```
struct sockaddr_in {
    sa_family_t    sin_family;   /* internet addr family */
    in_port_t      sin_port;     /* port number */
    struct in_addr sin_addr;     /* IP address */
    unsigned char  sin_zero[8];  /* padding */
};

struct in_addr {
    unsigned long s_addr;        /* load with inet_aton() */
};
```

► Zero Out

Let's declare an object of `sockaddr_in` and set it up first. To make sure all data in the memory space the object occupies are zeros, we need to use `memset()` to clear it up:

```
struct sockaddr_in server_addr;
memset(&server_addr, 0, sizeof(server_addr));
```

We then need to initialize the values in the object. As discussed before, `sin_family` indicates the domain, and so we'll use the same one we created for the socket, *i.e.*, `AF_INET`:[4]

```
server_addr.sin_family = AF_INET;
```

4: Sometimes you see people use `PF_INET`; they are mostly equivalent.

► Port Number

Another variable we need to set up is `sin_port`, which is the port number. However, notice that Linux uses little-endian to represent this number, but TCP/IP uses big-endian byte ordering. To convert a little-endian number to a big-endian, we can use the following functions:

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

where `l` stands for `long` (as you can see they are used in `uint32_t`, the unsigned 32-bit integers), while `s` for `short`. Letter `h` stands for host, while `n` for network.

In our case, because we want to convert our byte-ordering to the one the network is using, it's from host to network. For endianness, we can simply use long format. Thus, we have the following:

```
server_addr.sin_port = htonl(25555);
```

Notice that we use a port of `25555`, large enough to be safe for our purposes. You can certinaly use other numbers, but do not collide with reserved ports. Refer back to Section 6.3.1.3.

► IP Address

Lastly, we need to set up IP address. In the structure of `sockaddr_in` above, we see the IP address is stored in another struct called `in_addr`. This struct has only one variable called `s_addr`. Thus, we can simply initialize it as follows:

```
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

where `inet_addr()` takes a string that looks like an IP address (in the standard IPv4 dotted decimal notation `xx.xx.xx.xx`), and convert it to an integer value suitable for use as an Internet address.

6.3.2.4 Binding

The `bind()` function takes the socket file descriptor returned by `socket()` and an address structure like the one above, and "binds" them together to form the end of a socket that can now be used by processes living somewhere in the internet to find this server:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr* address,
 ↪  socklen_t addrlen);
```

Putting these few steps above together, we might start out with the following code:

```
struct sockaddr_in server_addr;
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family      = AF_INET;
server_addr.sin_port        = htons(25555);
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

```
bind(server_fd, (struct sockaddr *) &server_addr,
 ↪  sizeof(server_addr));
```

6.3.2.5 Listen

The socket created so far is an active socket, one that can connect to other sockets actively, like dialing another telephone number. Since this is the **server**, the purpose of this socket is not to "dial-out" but to listen for incoming calls. Therefore, the server must now call `listen()` to tell the kernel that all it really wants to do is listen for incoming messages and set a limit on its queue size. This call will basically put the socket into the `LISTEN` state in the TCP protocol.

The `listen()` is defined by

```
#include <sys/socket.h>
int listen(int sockfd, int queue_size);
```

The first argument is the descriptor for the already bound socket, and the second is the maximum size of the queue of pending (incomplete) connections.

After `listen()` has returned, two queues have been created for the server. One queue stores incoming connection requests that have not yet completed the TCP handshake protocol. The other queue stores incoming requests that have completed the handshake. These requests are ready to be serviced.

Continuing with our example, after binding, we can start listening:

```
if (listen(server_fd,5) == 0) printf("Listening\n");
else perror("listen");
```

Here we assume we can receive at most 5 incoming client connection requests.

6.3.2.6 Accept

During this step, we enter a loop in which it repeatedly accepts new connections and processes them. It can accept a new connection by calling `accept()`, defined as follows:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr* addr, socklen_t*
 ↪  addrlen);
```

This function returns a file descriptor, which represents the socket of the incoming message, *i.e.*, the clients! This file descriptor will be used to receive and send messages.

```
struct sockaddr_in in_addr;
socklen_t addr_size;
int       client_fd;
addr_size  =   sizeof(in_addr);
client_fd  =   accept(server_fd,
                      (struct sockaddr *) &in_addr,
                      &addr_size);
```

6.3.3 Writing a Client

Once the server has established, it'll be easier to write a client, since all it needs to do is to connect with the server. We'll create a new file, and write our client code there.

The first few steps are very similar:

```
int       server_fd;
struct    sockaddr_in server_addr;
socklen_t addr_size;

server_fd = socket(AF_INET, SOCK_STREAM, 0);
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family      = AF_INET;
server_addr.sin_port        = htons(25555);
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

Then we simply need to connect to the server:

```
connect(client_fd, (struct sockaddr *) &server_addr,
↪  addr_size);
```

6.3.4 Putting It All Together

Let's put all the stuff above together in a demo code in this section. The IP address we used is a local one so that we can test both server and client locally.

6.3.4.1 Server Code

Listing 6.4: A simple socket server setup code.

```
/*** socket/demo1/server.c ***/

#include <stdio.h>
#include <sys/socket.h>
```

```c
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>

int main() {
    int    server_fd;
    int    client_fd;
    struct sockaddr_in server_addr;
    struct sockaddr_in in_addr;
    socklen_t addr_size = sizeof(in_addr);

    /* STEP 1
        Create and set up a socket
    */
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(25555);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    /* STEP 2
        Bind the file descriptor with address structure
        so that clients can find the address
    */
    bind(server_fd,
            (struct sockaddr *) &server_addr,
            sizeof(server_addr));

    /* STEP 3
        Listen to at most 5 incoming connections
    */
    if (listen(server_fd, 5) == 0)
        printf("Listening\n");
    else perror("listen");

    /* STEP 4
        Accept connections from clients
        to enable communication
    */
    client_fd  =   accept(server_fd,
                          (struct sockaddr*)&in_addr,
                          &addr_size);

    return 0;
}
```

6.3.4.2 Client Code

```c
/*** socket/demo1/client.c ***/

#include <stdio.h>
#include <sys/socket.h>
```

Listing 6.5: A simple socket client setup code.

```c
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>

int main() {
    int    server_fd;
    struct sockaddr_in server_addr;
    socklen_t addr_size = sizeof(server_addr);

    /* STEP 1:
    Create a socket to talk to the server;
    */
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(25555);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    /* STEP 2:
    Try to connect to the server.
    */
    connect(server_fd,
            (struct sockaddr *) &server_addr,
            addr_size);

    return 0;
}
```

6.3.5 Communication

In the previous section we learned how to create sockets and set them up. Now we're going to see how we can send and receive data using sockets. Figure 6.8 is a refresher on the entire process, and we'll focus on the communication part.

6.3.5.1 Sending & Receiving Data

Sockets is one of the file types in UNIX system, and speaking of files, you should probably already know that we can use `read()` and `write()` to receive and send data from/to sockets. However, let's take this opportunity to introduce two socket-specific communications functions: `recv()` and `send()`.

The following are the prototypes of `recv()` and `send()`:

```c
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int socket_fd, void* buf,
             size_t len, int flags);
ssize_t send(int socket_fd, const void* buf,
```

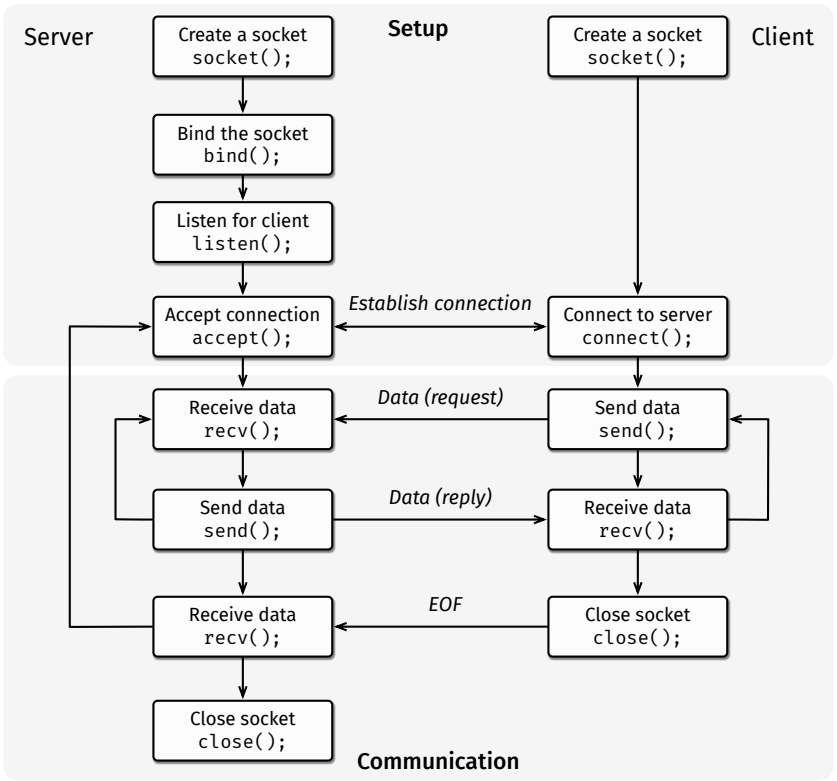


Figure 6.8: Flowchart for socket setup and communication.

```
6                size_t len, int flags);
```

As you can see for both functions, the first three parameters are literally identical to `read()` and `write()`. The only difference is the last parameter, `flags`. The flags can be used to turn on non-blocking operation (`MSG_DONTWAIT`), to notify the kernel that the process wants to receive out-of-band data (`MSG_OOB`), or to peek at the data without reading it (`MSG_PEEK`), to name a few. Our applications for this course can be very simple, so we can just pass `0` to the `flags`, in which case it is identical to `read()` and `write()`.

If you're interested, there are more functions for sending/receiving data for sockets, such as `sendto()` and `recvfrom()`, `sendmsg()` and `recvmsg()`, *etc*. See manpage for more details.

Now let's modify the code from previous section so that the client can send messages to the server, and the server can send back response to the client.

Server Code

We will add the following code to Listing 6.4 after step 4. For complete code, see `socket/demo2/server.c`.

Listing 6.6: A simple socket server code with sending/receiving data.

```
char* receipt   = "Read\n";
int   recvbytes = 0;
char  buffer[1024];

while(1) {
    recvbytes = recv(client_fd, buffer, 1024, 0);
    if (recvbytes == 0) break;
    else {
        buffer[recvbytes] = 0;
        printf("[Client]: %s", buffer);
        send(client_fd, receipt, strlen(receipt), 0);
    }
}

printf("Ending....\n");
close(client_fd);
close(server_fd);
```

Client Code

The following is the code segment we'll add to Listing 6.5 after step 3. As before, the complete code is listed in `socket/demo2/client.c`.

Listing 6.7: A simple socket client code with sending/receiving data.

```
char buffer[1024];

while(1) {

    /* Read a line from terminal
       and send that line to the server
     */
    printf("[Client]: "); fflush(stdout);
    scanf("%s", buffer);
    send(server_fd, buffer, strlen(buffer), 0);

    /* Receive response from the server */
    int recvbytes = recv(server_fd, buffer, 1024, 0);
    if (recvbytes == 0) break;
    else {
        buffer[recvbytes] = 0;
        printf("[Server]: %s", buffer); fflush(stdout);
    }
}

close(server_fd);
```

6.3.6 Multiplexed Server-Client Model

Now let's think about the limitations of the above server-client model. Say we have tons of clients trying to connect to our server, but our server code can only handle one connection— the function `accept()` was only called

once, and as soon as it connects to a client, there's no more clients allowed to be connected.

To fix this, we can simply modify Listing 6.6 as follows:

```
char* receipt   = "Read\n";
int   recvbytes = 0;
char  buffer[1024];
while(1) {

    /* Keep accepting new clients' connection requests */
    client_fd  =   accept(server_fd,
                          (struct sockaddr *) &in_addr,
                          &addr_size);
    if (client_fd != -1) printf("New client detected!\n");

    recvbytes = recv(client_fd, buffer, 1024, 0);
    if (recvbytes == 0) break;
    else {
        buffer[recvbytes] = 0;
        printf("[Client]: %s", buffer);
        send(client_fd, receipt, strlen(receipt), 0);
    }

    /* Close connection, and wait for the next client */
    close(client_fd);
}

printf("Ending....\n");
close(server_fd);
```

Let's say now we have two clients (A and B) trying to connect to us. The problem with the code above is, once a client has been connected (say A), we'll block all other requests while waiting for the input from A. What if A suddenly decides to go to lunch? Clients need to eat too! Then B cannot get its message delivered until A's back from lunch. In this case, we'd have to use some other mechanisms to **not** wait for pending inputs. The function that achieves this is `select()`. Before talking about how to use it, let's look at two concepts: multiplexed I/O model, and file descriptor sets.

6.3.6.1 Multiplexed I/O Model

Most of the time in our programs we deal with single channel I/O. For example, if we call `read()` function, the source of the input comes from only one file descriptor (could be `stdin`, or any other file descriptor). As we see above, however, in more advanced situations, multiple sources might want to send input to our process *concurrently*. Each of those sources has their own file descriptors, but `read()` can only receive one at a time. If one file descriptor is not ready, all others will be blocked.

In Linux, this is called multiplexed I/O. In fact, it's more common than we thought, and occurs not only in socket programming.

> *From The Linux Programming Interface, Kerrisk, pp.1330.*
>
> ---
>
> I/O multiplexing allows us to simultaneously monitor multiple file descriptors to see if I/O is possible on any of them.

In other words, we just need to actively monitor them, and once one of them is ready, we perform I/O on the file descriptor. Let's highlight certain keywords from the previous sentence that will directly correspond to our program structure:

- **Actively.** Think about the shell. The shell can be considered as *actively* waiting for the input. The corresponding program structure is to use an infinite loop. This is also precisely what we are going to use here;
- **Monitor.** How do we actually watch all the file descriptors, and determine which file descriptor is ready? The answer is using a function called `select()`. See next subsection;
- **Ready.** What do we mean when we say a file descriptor is ready? Ready for what? Recall from Chapter 5, under certain circumstances, an I/O operation such as `read()` will be blocked. When in such situations where an I/O operation is blocked, the file descriptor is **not** ready. Once it's unblocked, it becomes ready.

Based on this analysis, we have the following structure of the program:

```
/* The while loop for "actively" monitoring */
while(1) {

    /* Monitor file descriptors */
    select(...);

    /* Perform I/O when a descriptor is ready */
    read(...);

}
```

We will look into detail on these smaller parts.

6.3.6.2 File Descriptor Sets

Linux uses a data structure called `fd_set` to manage all file descriptors that could be used for I/O operations, and this is also the structure we monitor on. Note that there's limit on how many file descriptors we can add to a set, which is 1,024 on Linux, defined by constant `FD_SETSIZE`. The type of `fd_set` is implemented as a bit mask, and we don't really need to know how it's implemented.

To create a file descriptor set, simply write something like `fd_set myfdset;`. Then there are several functions (actually macros) we can use to modify this set:

```
#include <sys/select.h>
/* initializes the set pointed to by fdset to be empty. */
void FD_ZERO(fd_set* fdset);

/* adds the file descriptor fd to the set
   pointed to by fdset. */
void FD_SET(int fd, fd_set* fdset);

/* removes the file descriptor fd from the set
    pointed to by fdset. */
void FD_CLR(int fd, fd_set* fdset);

/* returns true if the file descriptor fd is
   a member of the set pointed to by fdset. */
int  FD_ISSET(int fd, fd_set* fdset);
```

6.3.6.3 Select

As mentioned before, `select()` function will monitor all file descriptors in the `fd_set`, and perform I/O on the ones that are ready. The prototype of `select()` is as follows:

```
#include <sys/select.h>
int select(int nfds, fd_set* readfds, fd_set* writefds,
           fd_set* exceptfds, struct timeval* timeout);
```

which will return the number of ready file descriptors in the set, 0 on timeout, or –1 on error.

Let's look at the arguments of the function. The first parameter `nfds` is set to the highest-numbered file descriptor in any of the three sets, plus 1.

The following three arguments are the `fd_set`: one for read, one for write, and one for exceptional conditions (not errors). `select()` function will check each of the sets for corresponding operations. For example, it'll check all the file descriptors in the set `readfds` to see if any of them is ready for input.

Lastly, `timeout` parameter controls the blocking of `select()` function. The simplest we can pass is `NULL`, which will block `select()` until one of the following situation happens:

- At least one of the file descriptors specified in `readfds`, `writefds`, or `exceptfds` becomes ready;
- The call is interrupted by a signal handler.

If you only want `select()` waits (in other words, be blocked) for a specific amount of time, you can declare a `timeval` struct object, and pass its address to `select()`. We'll skip it here.

The program will halt at `select()` function, until one of the monitored file descriptors becomes ready. When `select()` returns, all the non-ready file descriptors will be removed from the monitored file descriptor set, and only those who are ready will stay. Thus, after `select()` function, we can use `FD_ISSET()` to check on the file descriptor of interest, to see if it's ready. For example,

```
select(nfds, &fds, NULL, NULL, NULL);
if ( FD_ISSET(myfd, &fds) ) { ... }
```

This code calls `select()` to monitor a file descriptor set `fds` which includes `myfd`. When `select()` is unblocked (that is, one of the file descriptors is ready), we can use the next `if`-statement to check if it is `myfd` that is ready.

The following excerpt from manpage is very important that you need to know and remember and be careful about:

> *From manpage,* `https://man7.org/linux/man-pages/man2/select.2.html`.
>
> ---
>
> After `select()` has returned, `readfds` (and/or `writefds`) will be cleared of all file descriptors except for those that are ready for reading (and/or writing).

6.3.6.4 Ready?

We mentioned the word "ready" multiple times. What does that even mean in terms of programming? Recall that we said it is I/O operation that is ready. For different types of files, the word "ready" has different meanings. Since we're only focusing on sockets, we'll omit others.

In sockets, `select()` function will consider a file descriptor ready if:

- Input available for reading, such as hitting the enter key after typing a string; **or**
- Output possible for writing; **or**
- Incoming connection established on listening socket for reading; **or**
- Stream socket peer closed connection or executed.

What I'd like you to pay particular attention to is the third situation, because that's one of the key parts of writing a concurrent server with socket. When `select()` is unblocked, there could be two possible reasons: one is we received an input from a client, and another one is we just established

a connection to a new client. Therefore, we have to differentiate these two situations in our program.

6.3.6.5 Writing a Multiplexed Echo Server

Let's now just write a small segment of the concurrent server. What we want to accomplish here is to check if there's any new connections established to the server, and if so, print a message `"New connection established."`.

► Initialization

The initialization part is the same. We will just list the key steps here:

```c
#define MAX_CONN 5
int  server_fd;
int  in_fd;
struct sockaddr_in server_addr;
struct sockaddr_in in_addr;
socklen_t addr_size = sizeof(struct sockaddr_in);

server_fd = socket(...);     /* Step 1 - socket */
bind(...);                   /* Step 2 - binding */
listen(server_fd, MAX_CONN); /* Step 3 - listening */
```

Next let's think about what other variables we need. First of all, we define an `fd_set` called `active_fds`. We also need an array to store all the file descriptors from the client. We do have `active_fds` to monitor them, but remember, once we are returned `select()` function, `active_fds` will be cleared out, so without an array we will lose all other file descriptors. Thus, we can declare an integer array that stores all client file descriptors, and initialize them with -1:

```c
fd_set active_fds;
int    client_fds[MAX_CONN];
int    nconn = 0;
for (size_t i = 0; i < MAX_CONN; i ++) client_fds[i] = -1;
```

Next, we will write the main structure of the server where we actively monitor file descriptors. As discussed before, we follow the infinite loop pattern:

```c
while (1) {

    /* Part 1: Preparation of the file descriptor set */

    /* Part 2: Monitor the file descriptor set */

    /* Part 3: Accept new connections */
```

```
8
9       /* Part 4: Close connections */
10
11  }
```

and we will look at each part in detail in the following paragraphs.

► Part 1: Prepare the File Descriptor Set

There are three things we have to do at the beginning of each iteration. First, we will have to clear out `active_fds`. This is because, again, once we have passed `select()` function, `active_fds` will only contain the file descriptors that are ready. By clearing out `active_fds`, we have a "fresh start".

The second thing we need to do is to re-add all the file descriptors we want to monitor to `active_fds`. The `server_fd` is one we have to add of course, because that file descriptor is used to establish new connections. For other client file descriptors, remember we have `client_fds[]` that keeps track of all of them, so we can use a loop to iterate over `cliend_fds[]`: if `client_fds[i]` is not -1, we call `FD_SET()` to add it.

Last thing, is to make sure we know the largest file descriptor value among all the ones we are monitoring. This is for the first argument to `select()`.

Putting them together, we will write a function called `prep_fds()` that returns the maximal file descriptor:

```
1   int prep_fds( fd_set* active_fds,
2                 int     server_fd,
3                 int*    client_fds ) {
4
5       FD_ZERO(active_fds);
6       FD_SET(server_fd, active_fds);
7
8       int max_fd = server_fd;
9
10      for (int i = 0; i < MAX_CONN; i ++) {
11          if (client_fds[i] > -1)
12              FD_SET(client_fds[i], active_fds);
13          if (client_fds[i] > max_fd)
14              max_fd = client_fds[i];
15      }
16
17      return max_fd;
18  }
```

► Part 2: Monitor the File Descriptor Set

Monitoring the file descriptor set is fairly easy. We just need to get the maximal file descriptor back from `prep_fds()`:

```
int max_fd = prep_fds(&active_fds, server_fd, client_fds);
select(max_fd + 1, &active_fds, NULL, NULL, NULL);
```

Here we are only monitoring for reading and don't care about timeout, so the last three parameters are all `NULL`.

► Part 3: Accept New Connections

The program will halt at `select()`. In the beginning there's only `server_fd` in `active_fds`, so the program being resumed is a signal for new connection. However, as we add more clients' file descriptors, they can also unblock `select()` function. Thus, we just need to check if it is `server_fd` that's ready. If it is, we need to call `accept()` function to get the new client's file descriptor `in_fd`:

```
if (FD_ISSET(server_fd, &active_fds)) {
    in_fd = accept(server_fd,
                   (struct sockaddr*) &in_addr,
                   &addr_size);

     if (nconn == MAX_CONN) {
        close(in_fd);
        fprintf(stderr, "Max connection reached!\n");
     }
     else {
        for (int i = 0; i < MAX_CONN; i ++) {
            if (client_fds[i] == -1) {
                client_fds[i] = in_fd;
                break;
            }
        }
        printf("New connection detected!\n");
        nconn ++;
     }
}
```

► Part 4: Close Connections

When a client process terminates, it will send an EOF to the server, and correspondingly, the client's file descriptor in the server process also becomes ready for read. Or more specifically, ready for reading the EOF. In this case, we need to find out which process is terminating, and if `read()`

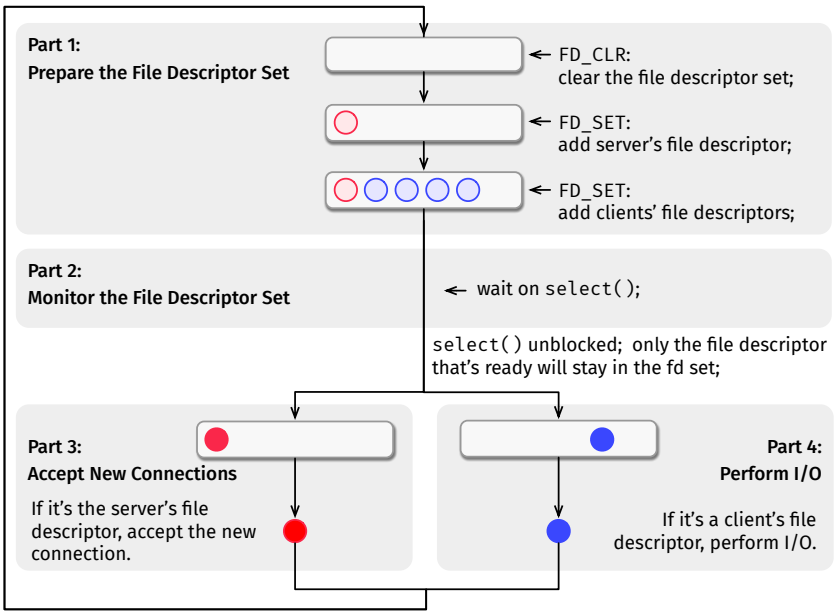


Figure 6.9: In a multiplexed server-client model, all the available file descriptors (including the server's and the connected clients') will be added to the `fd_set` to monitor. Once `select()` function is unblocked, suggesting some file descriptor is ready for I/O, the `fd_set` will only save the one that's ready, and remove all others.

returns 0, we will close that file descriptor, remove it from `active_fds`, and set it to -1 in `client_fds[]`:

```
for (int i = 0; i < MAX_CONN; i ++) {
    if (client_fds[i] > -1 && \
        FD_ISSET(client_fds[i], &active_fds)) {
        char buffer[1024];
        if (read(client_fds[i], buffer, 1024) == 0) {
            close(client_fds[i]);
            FD_CLR(client_fds[i], &active_fds);
            client_fds[i] = -1;
            printf("Lost connection!\n");
            nconn --;
        }
    }
}
```

We use Figure 6.9 to illustrate the four parts above. The entire server program is written in an infinite loop. We first clear out the file descriptor set `active_fds` and add both the server's and the connected clients' file descriptors to it. The `select()` will block the program until some file descriptors are ready for I/O operation. If it is the server that's ready, that means a new connection is detected. If it is the client's, it's ready to perform I/O — either by sending an EOF to cut the connection, or by sending some data through `write()`.

After the corresponding action is taken, at the end of the loop, only the file descriptor we operated on will stay in `active_fds`; all others have been deleted from it by `select()` function. We then go back to the beginning of the loop, and restart the process again.

Alphabetical Index