2 Programs and Processes

2.1. Processes

A running program is called a **process**. The name *program* is not used to refer to a running program because both concepts differ. The difference is the same that you may find between a cookie recipe and a cookie. A program is just a bunch of data, and not something alive. On the other hand, a process is a living program. It has a set of registers including a program counter and a stack. This means that it has a *flow of control* that executes one instruction after another as you know.

The difference is quite clear if you consider that you may execute simultaneously the same program more than once. For example, figure 2.1 shows a window system with three windows. Each one has its own shell. This means that we have three processes running `/bin/rc`, although there is only a single program for those processes. Namely, that kept stored in the file `/bin/rc`. Furthermore, if we change the working directory in a shell, the other two ones remain unaffected. Try it! Each shell process has its own *current working directory* variable. However, the program had only one such variable declared.

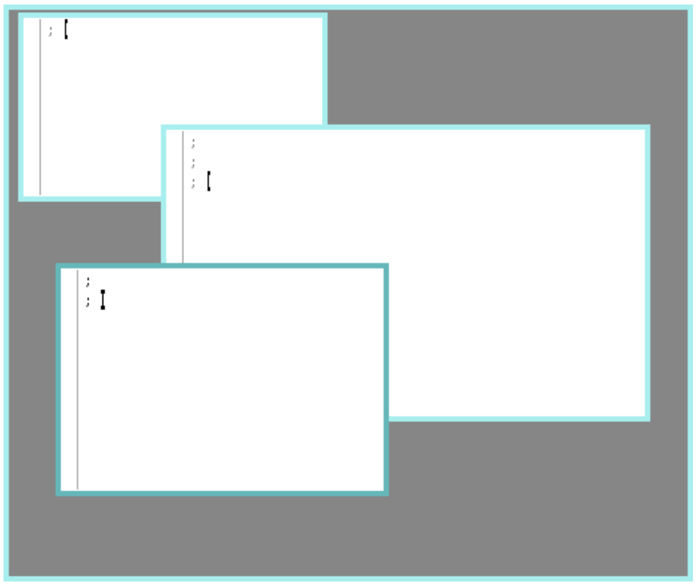

Figure 2.1: *Three* `/bin/rc` *processes. But just one* `/bin/rc`.

So, what is a process? Consider all the programs you made. Pick one of them. When you execute your program and it starts execution, it can run **independently** of all other programs in the computer. Did you have to take into account other programs like the window system, the system shell, a clock, a web navigator, or any other just to write your own (independent) program and execute it? Of course not. A brain with the size of the moon would be needed to be able to take all that into account. Because no such brains exist, operating systems provide the process abstraction. To let you write and run one program and *forget* about other running programs.

Each process gets the *illusion* of having its own processor. When you write programs, you think that the machine executes one instruction after another. But you always think that all the instructions belong to your program. The implementation of the process abstraction included in your system provides this fantasy.

When machines have several processors, multiple programs can be executed in **parallel**, i.e., at the same time. Although this is becoming common, many machines have just one processor. In some cases we can find machines with two or four ones. But in any case, you run many more programs than processors are installed. Count the number of windows at your terminal. There is at least one program per window. You do not have that many processors.

What happens is that the operating system makes arrangements to let each program execute for just some time. Figure 2.2 depicts the memory for a system with three processes running. Each process gets its own set of registers, including the program counter. The figure is just an snapshot made at a point in time. During some time, the process 1 running `rio` may be allowed to proceed, and it would execute its code. Later, a hardware timer set by the system may expire, to let the operating system know that the time for this process is over. At this point, the system may *jump* to continue the execution of process 2, running `rc`. After the time for this process expires, the system would jump to continue execution for process 3, running `rio`. When time for this process expires, the system may jump back to process 1, to continue where it was left at.

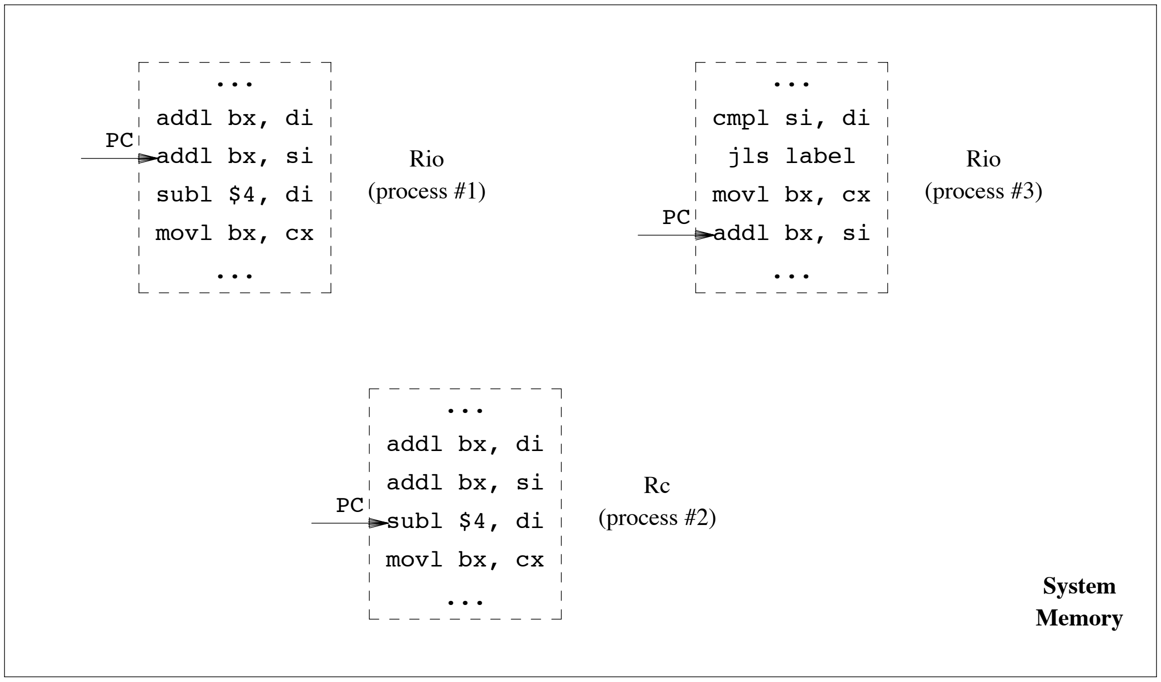

Figure 2.2: *Concurrent execution of multiple programs in the same system.*

All this happens behind the scene. The operating system program knows that there is a single flow of control per processor, and jumps from one place to another to transfer control. For the users of the system, all that matters is that each process executes independently of other ones, as if it had a single processor for it.

Because all the processes appear to execute simultaneously, we say they are **concurrent**. In some cases, they will really execute in **parallel** when each one can get a real processor. In most cases, it would be a **pseudo-parallel** execution. For the programmer, it does not matter. They are just concurrent processes that seem to execute simultaneously.

In this chapter we are going to explore the process we obtain when we execute a program. Before doing so, it is important to know what's in a program and what's in a process.

2.2. Loaded programs

When a program in source form is compiled and linked, a binary file is generated. This file keeps all the information needed to execute the program, i.e., to create a process that runs it. Different parts of the binary file that keep different type of information are called sections. A binary file starts with a few words that describe the following sections. These initial words are called a header, and usually show the architecture where the binary can run, the size and offset in the file for various sections.

One section (i.e., portion) of the file contains the program text (machine instructions). For initialized global variables of the program, another section contains their initial values. Note that the system knows *nothing* about the meaning of these values. For uninitialized variables, only the total memory size required to hold them is kept in the file. Because they have no initial value, it makes no sense to keep that in the file. Usually, some information to help debuggers is kept in the file as well, including the strings with procedure and symbol names and their addresses.

In the last chapter we saw how *nm*(1) can be used to display symbol information in both object and binary files. But it is important to notice that only your program code knows the meaning of the bytes in the program data (i.e., the program knows what a variable is). For the system, your program data has no meaning. **The system knows nothing** about your program. It's you the one who knows. The program *nm* can display information about the object file because it looks at the symbol table stored in the binary for debugging purposes.

We can see this if we remove the symbol table from our binary for the `take.c` program. The command *strip*(1) removes the symbol table. To find the binary file size, we can use option `-l` for *ls*, which lists a long line of information for each file, including the size in bytes.

```
; ls -l 8.take
--rwxr-xr-x M 19 nemo nemo 36348 Jul  6 22:49 8.take
; strip 8.take
; ls -l 8.take
--rwxr-xr-x M 19 nemo nemo 21713 Jul  6 22:49 8.take
```

The number after the user name and before the date is the file size in bytes. The binary file size changed from 36348 bytes down to 21713 bytes. The difference in size is due to the symbol table. And without the symbol table, *nm* knows nothing. Just like the system.

```
; nm 8.take
;
```

A program stored in a file is different from the same program stored in memory while it runs. They are related, but they are not the same. Consider this program. It does nothing, but has a global variable of one megabyte.

global.c

```
#include <u.h>
#include <libc.h>

char global[1 * 1024 * 1024];

void
main(int, char*[])
{
        exits(nil);
}
```

Assuming it is kept at `global.c`, we can compile it and use the linker option `-o` to specify that the binary is to be generated in the new file `8.global`.

```
; 8c -FVw global.c
; 8l -o 8.global global.8

; ls -l 8.global global.8
--rwxr-xr-x M 19 nemo nemo 3380 Jul  6 23:06 8.global
--rw-r--r-- M 19 nemo nemo  328 Jul  6 23:06 global.8
```

Crearly, there is no room in the 371 bytes of the object file for the `global` array, which needs one megabyte of storage. The explanation is that only the size required to hold the (not initialized) array is kept in the file. The binary file does not include the array either (change the array size, and recompile to check that the binary file does not change size).

When the shell asks the system (making a system call) to execute *global*, the system **loads the program** into memory. The part of the system (kernel) doing this is called the **loader**. How can the system load a program? By reading the information kept in the binary:

" The header in the binary file reports the memory size required for the program text, and the file keeps the memory image of that text. Therefore, the system can just copy all this into memory. For a given system and architecture, there is a convention regarding which addresses the program must use. Therefore, the system knows where to load the program.

" The header in the binary reports the memory size required for initialized variables (globals) and the file contains a memory image for them. Thus, the system can copy those bytes to memory. Note that the system has no idea regarding where does one variable start or how big it is. The system only knows how many bytes it has to copy to memory, and at which address should they be copied.

" For uninitialized global variables, the binary header reports their total size. The system allocates that ammount of memory for the program. That is all it has to do. As a courtesy, Plan 9 guarantees that such memory is initialized with all bytes being zero. This means that all your global variables are initialized to null values by default.

We saw how the program *nm* prints addresses for symbols. Those addresses are memory addresses that are only meaningful when the program has been loaded. Using *nm* we can learn more about how the memory of a program looks like. Option `-n` asks *nm* to sort the output by address.

```
; nm -n 8.global
            1020 T main
            1033 T _main
            1073 T atexit
            10e2 T atexitdont
            1124 T exits
            1180 T _exits
            1188 T getpid
            11fb T memset
            122a T lock
            12e7 T canlock
            130a T unlock
            1315 T atol
            1442 T atoi
            1455 T sleep
            145d T open
            1465 T close
            146d T read
            14a0 T _tas
            14ac T pread
            14b4 T etext
            2000 D argv0
            2004 D _tos
            2008 D _nprivates
            200c d onexlock
            2010 D _privates
            2014 d _exits
            2024 B edata
            2024 B onex
            212c B global
          10212c B end
```

Figure 2.3 shows the layout of memory for this program when loaded. Looking at the output of *nm* we can see several things. First, the program code uses addresses starting at 0x1020 up to 0x14c1. We use the C language syntax for hexadecimal literals to avoid confussion.

The last symbol in the code is `etext`, which is a symbol defined by the linker to let you know where the end of text is. Data goes from adress 0x2000 up to address 0x10212c. There is a symbol called `end`, also defined by the linker, at the end fo the data. This symbol lets you know where the end of data is. This symbol is not to be confused with `edata`, which reports the address where initialized data terminates.

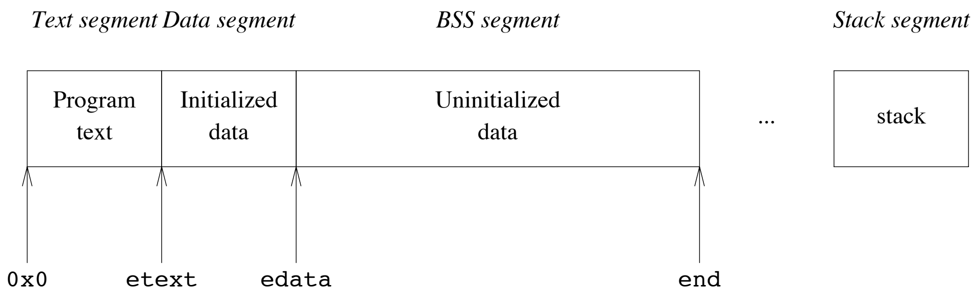

Figure 2.3: *Memory image for the* `global` *program.*

In decimal, the address for `end` is 5.175.468 bytes! That is 4.9 Mbytes, which is a lot of memory for a program that was kept in a binary file of 3 Kbytes. Can you see the difference?

And there is more. We did not take into account the program stack. Therefore, the size of the program when loaded into memory will be even larger. To know how much memory a program will consume, use *nm*, do not list the binary file.

The memory of a process is arranged as shown in figure 2.3. But that is an invention of the operating system. That is the abstraction supplied by the system, implemented using the virtual memory hardware, to make your life easier. This abstraction is called **virtual memory**. A process believes that it is the only program loaded in memory. You can notice by looking at the addresses we did show. All processes for that program will use the same addresses. And more than just one can run at the same time.

The virtual memory of a process in Plan 9 has several, so called, **segments**. This is also an abstraction of the system and has few to do with the segmentation hardware found at some popular processors. A segment is a portion of contiguous memory with some properties. Segments used by a Plan 9 process are:

" The **text segment**. It contains instructions that can be executed but not modified. The hardware is used by the system to enforce these permissions. The memory is initialized by the system using the program text (code) kept within the binary file for the program.

" The **data segment**. It contains the initialized data for the program. Protection is set to allow both read and write operations on it, but you cannot execute instructions on it. The memory is initialized by the system using the initialized data kept within the binary file for the program.

" The uninitialized data segment, called **bss segment** is almost like the data segment. However, this one is initialized by zeroing its memory. The name of the segment comes from an arcane instruction used to implement it on a machine that no longer exists. How much memory is given depends on the size recorded in the binary file. Moreover, this segment can *grow*, by using a system call that allocates more memory for it. Function libraries like *malloc*(2) cause this segment to grow when they consume all the available memory in this segment.

" The **stack segment** is also used for reading and writing memory. Unlike other segments, this segment seems to grow automatically when more space is used. It is used to keep the stack for the executing program.

All this is important to know because it has impact on your programs. For example, because memory is *virtual*, and is only allocated when first used, any unused part of the BSS segment is free! It consumes no memory until you touch it. However, if you initialized it with a loop, all the memory will be allocated. One particular case when this may be useful is when you implement large sparse hash tables, that contain few elements. You might implement them using a huge array, not initialized. If you read one portion of the array for the first time, the system will allocate memory and zero it. Your array entries would be all nulls. The same happens if you write. Therefore, in this example, initializing by hand an array has a big impact on memory consumption.

2.3. Birth and death

Programs are not *called*, they are *executed.* Besides, programs do not *return*, their processes terminate when they want or when they misbehave. Being this said, we can supply arguments to programs we run, to control what they do.

When the shell asks the system to execute a program, after it has been loaded into memory, the system provides a flow of control for it. This means just that a full set of processor registers is initialized for the new running program, including the program counter and stack pointer, along with an initial (almost empty) stack. When we compile a C program, the loader puts `main` at the address where the system will start executing the code. Therefore, our C programs start running at `main`. The arguments supplied to this program (e.g., in the shell command line) are copied by the system to the stack for the new program.

The arguments given to the `main` function of a program are an array of strings and the number of strings kept in the array. We can write a program to print its arguments.

echo.c

```
#include <u.h>
#include <libc.h>

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

        for (i = 0; i < argc; i++)
                print("%d: %s\n ", i, argv[i]);
        exits(nil);
}
```

If we execute it we can see which arguments are given to the program for a particular command line:

```
; 8c -FVw echo.c
; 8l -o 8.echo echo.8
; ./8.echo one little program
0: ./8.echo
1: one
2: little
3: program
;
```

There are several things to note here. First, the first argument supplied to the program is the program name! More precisely, it is the command name as given to the shell. Second, this time we gave a relative path as a command name. Remember, `./8.echo`, is the file `8.echo` within the current working directory for our shell. which is a relative path. And that was the value of `argv[0]` for our program. Programs know their name by looking at `argv[0]`, which is very useful to print diagnostic messages while letting the user know which program was the one that had a problem.

There is a standard command in Plan 9 that is almost the same. See *echo*(1). This program prints its arguments separated by white space and a new line. The new line can be supressed with the option `-n`.

```
; echo hi there
hi there
;
; echo -n hi there
hi there;
```

Note the shell prompt right after the output of echo. Despite being simple, echo is invaluable to know which arguments a program would get, and to generate text strings by using echo to print them.

A program terminates by a call to `exits`, see *exits*(2) for the whole story. This system call terminates the calling process. The process may leave a single string as its legacy, reporting what it has to say. If the string is null, it means by convention that everything went well for the dying process, i.e., it could do its job. Otherwise, the convention is that string should report the problem the process had to complete its job. For example,

sic.c

```
#include <u.h>
#include <libc.h>
void
main(int, char*[])
{
        exits("sic!");
}
```

would report `sic!` to the system when `exits` terminates the process. Here is a run that shows that by echoing `$status` we can learn how it went to this depressive program.

```
; 8.sic
; echo $status
8.sic 2046: sic!
;
```

We lied before when we said that a program starts running at `main`, it does not. It starts running at a function that calls `main` and then (when `main` returns), calls `exits` to terminate the execution. That is the reason why a process ceases existing when the main function of the program returns. The process makes a system call to terminate itself. There is no magic here, and a process may not cease existing merely because a function returns. A flow of control does not vanish, the processor always keeps on executing intructions. However, because processes are an invention of the operating system, we can use a system call that destroyes the calling procress. The system deallocates its resources and and the process is history. A process is a data type after all.

2.4. Errors

In this chapter and the following ones we are going to make a lot of system calls from programs writen in C. In many cases, there will be no problem and a system call we make will be performed. But in other cases we will make a mistake and a system call will not be able to do its work. For example, this will happen if we try to change our current working directory and supply a path that does not exist.

Almost any function that we call (and system calls are functions) may have problems to complete its job. In Plan 9, when a system call encounters an error or is not able to do its work, the function returns a value that alerts us of the error condition. Depending on the function, the return value indicating the error may be one or another. In general, absurd return values are used to report errors.

For example, we will see how `open` returns a possitive small integer. However, upon failure, it returns -1. This is the convention for most system calls returning integer values. System calls that return strings will return a null string when they fail, and so on. The manual pages report what a system call does when it fails.

You must **always check out for error conditions**. If you do not check that a system call could do its work, you do not know if it worked. Be warned, not checking for errors is like driving blind, and it will surely put you into a debugging Inferno (limbo didn't seem bad enough).

Besides reporting the error with an absurd return value from the system call, Plan 9 keeps a string describing the error. This **error string** is invaluable information for fixing the problem. You really want to print it out to let the user know what happen.

There are several ways of doing so. The more convenient one is using the format `%r` in `print`. This instructs `print` to ask Plan 9 for the error string and print it along with other output. This program is an example.

err.c

```
#include <u.h>
#include <libc.h>

void
main(int , char* [])
{
        if (chdir("magic") < 0){
                sysfatal("chdir failed: %r\n");
        }
        /* ... do other things ... */
        exits(nil);
}
```

Let's run it now

```
; 8.err
chdir failed: 'magic' file does not exist
```

The program tried to use `chdir` to change its current working directory to `magic`. Becase it did not exist, the system call failed and returned `-1`. A good program would always check for this condition, and then report the error to the user. Note the use of `%r` in `print` and compare to the output produced by the program.

If the program cannot proceed because of the failure, it is sensible to terminate the execution indicating that the program failed. This is so common that there is a function that both prints a message and exits. It is called `sysfatal`, and is used like in this example

```
        if (chdir("magic") < 0)
                sysfatal("chdir failed: %r");
```

In a few cases you will need to obtain the error string for a system call that failed. For example, to modify it and print a customary diagnostic message. The system call `rerrstr` reads the error string. It stores the string at the buffer you supply. Here is an example

```
        char    error[128];
        ...
        rerrstr(error, sizeof error);
```

After the call, `error` contains the error string.

A function implemented to be placed in a library also needs to report errors. If you write such function, you must think how to do that. One way is to use the same mechanism used by Plan 9. This is good because it allows any programmer using your library to do exactly the same to deal with errors, no matter if the error is being reported by your library function or by Plan 9.

The system call `werrstr` writes a new value for the error string. It is used like `print`. Using it, we can implement a function that `pops` an element from a stack and reports errors nicely

```
int
pop(Stack * s)
{
        if (isempty(s)){
                werrstr("pop of an empty stack");
                return -1;
        }
        ... do the pop otherwise ...
}
```

2.5. Environment

Another way to supply arguments to a process is to define **environment variables**. Each process is supplied with a set of *name=value* strings, that are known as environment variables. They are used to customize the behaviour of certain programs, when it is more convenient to define a environment variable than to give a command line argument every time we run a program. Usually, all processes running in the same Rio window share the environment variables.

For example, the variable *home* has the path for your home directory as its value. The command *cd* uses this variable to know where your home is. Otherwise, how could it know what do when given no arguments? Both names and values of environment variables are strings. Remember this.

We can define environment variables in a shell command line by using an equal sign. Later, we can use the shell to refer to the value of any environment variable. After reading each command line, the shell replaces each word starting with a dollar sign with the value of the environment variable whose name follows the dollar. For example,

```
; dir=/a/very/long/path
; cd $dir
; pwd
/a/very/long/path
;
```

defined the variable *dir*. The second command line used `$dir`, and therefore, the shell replaced the string `$dir` with the string that is the value of the *dir* environment variable: `/a/very/long/path`. Note that *cd* knows nothing about `$dir`. We can see this using *echo*, because we know it prints the arguments received verbatim.

```
; echo $dir
/a/very/long/path
;
```

These two commands do the same. However, one receives one argument, the other does not. The output of `pwd` would be the same after any of them.

```
; cd $home
; cd
```

You know understand what `$status` means. It is the value of the evironment variable *status*. This variable is updated by the shell once it finds out how it went to the last command it executed. This is done before prompting for the next command. As you know, the value of this variable would be the string given to *exits* by the process running the command.

Another interesting variable is **path**. This variable is a list of paths where the shell should look for executable files to run the user commands. When you type a command name that does not start with `/` or `./`, the shell looks for an executable file relative to each one of the directories listed in `$path`, in the same order. If a binary file is found, that is the one executed to run the command. This is the value of the *path* variable in a typical Plan 9 shell:

```
 ; echo $path
 . /bin
 ;
```

It contains the working directory, and `/bin`, in that order. If you type `ls`, the shell tries with `./ls`, and if there is no such file, it tries with `/bin/ls`. If you type `ip/ping`, the shell tries with `./ip/ping`, and then with `/bin/ip/ping`. Simple, isn't it?

Two other useful environment variables are *user*, which contains the user name, and *sysname*, which contains the machine name. You may define as many as you want. But be careful. Environment variables are usually forgotten while debugging a problem. If some program input value should be a command line argument, use a command line argument. Given sensible default values to program parameters can avoid the burden of having to supply always the same arguments. Command line arguments make the program invocation more clear at first sight, and therefore, simpler to grasp and debug.

Because of the syntax in the shell for environment variables, we may have a problem if we want to run *echo*, or any other program, supplying arguments containing either the dollar sign, or the equal sign. Both characters we know are special. This can be done by asking the shell not to do anything with a string we type, and to take it literally. Just type the string into single quotes and the shell will not change anything between them:

```
; echo $user
nemo
; echo '$user' is $user
$user is nemo
;
```

Note also that the shell behaves always the same way regarding command line text. For example, the first word (which is the command name) is not special, and we can do this

```
; cmd=pwd
; $cmd
/usr/nemo
;
```

and use variables whereeever we want in command lines. Also, quoting works always the same way. Let's try with the *echo* program we implemented before:

```
; echo 'this is' weird
0: echo
1: this is
2: weird
;
```

As you may see, `argv[1]` contains the string `this is`, including the white space. The shell did not split the string into two different arguments for the command. Because you quoted it! Even the new line can be quoted.

```
; echo 'how many
;; lines'
how many
lines
```

The prompt changed because the shell had to read more input, to complete the quoted string. That is its way of telling us. Quoting also removes the special meaning of other characters, like the backslash:

```
; echo \
;;          waiting for the continuation of the line
;           ...until we press return
            echo prints the empty line
; echo '\'
\
;
```

To obtain the value for a environment variable, from a C program, we can use the *getenv*(2) system call. An of course, the program must check out for errors. Even `getenv` can fail. Perhaps the variable was not defined. In this case `getenv` returns a null string.

env.c

```
#include <u.h>
#include <libc.h>

void
main(int, char*[])
{
        char*   home;

        home = getenv("home");
        if (home == nil)
                sysfatal("we are homeless");
        print("home is %s\n", home);
        exits(nil);
}
```

Running it yields

```
; 8.env
home is /usr/nemo
```

A related call is *putenv*(2), which accepts a name and a value, and sets the corresponding environment variable accordingly. Both the name and value are strings.

2.6. Process names and states

The name of a process is not the name of the program it runs. That is convenient to know, nevertheless. Each process is given a unique number by the system when it is created. That number is called the **process id**, or the *pid*. The pid identifies, and therefore names, a process.

The pid of a process is a possitive number, and the system tries hard not to reuse them. This number can be used to name a process when asking the system to do things to it. Needless to say that this *name* is also an invention of the operating system. The shell environment varialbe *pid* contains the pid for the shell. Note that its value is a string, not an integer. Useful for creating temporary files that we want to be unique for a given shell.

To know the pid of the process that is executing our program, we can use `getpid`(2):

pid.c

```
#include <u.h>
#include <libc.h>

void
main(int, char*[])
{
        int     pid;

        pid = getpid();
        print("my pid is %d\n", pid);
        exits(nil);
}
```

Executing this program several times may look like this

```
; 8.pid
my pid is 345
; 8.pid
my pid is 372
;
```

The first process was the one with pid 345, but we may say as well that the first process was the 345, for short. The second process started was the 372. Each time we run the program we would get a different one. The command *ps*(1) lists the processes in the system. The second field of each line (there is one per process) is the process id. This is an example

```
; ps
nemo            280    0:00   0:00  13 13    1148K Pread    rio
nemo            281    0:02   0:07  13 13    1148K Pread    rio
nemo            303    0:00   0:00  13 13    1148K Await    rio
nemo            305    0:00   0:00  13 13     248K Await    rc
nemo            306    0:00   0:00  13 13    1148K Await    rio
        ... more output omitted ...
```

The last field is the name of the program being run by the process. The third field going right to left is the size of the (virtual) memory being used by the process. You may now know how much memory a program consumes when loaded.

The second field on the right is interesting. You see names like `Pread` and `Await`. Those names reflect the **process state**. The process state indicates what the process is doing. For example, the first two Rios are reading something, and everyone else in the listing above is awaiting for something to happen. To understand this, it is important to get an idea of how the operating system implements processes.

There is only one processor, but there are multiple processes that seem to run simultaneously. That is the process abstraction. Multiple programs that execute independently of each other. None of them transfer control to others. However, the processor implements only a single flow of control. What happens is that when one process enters the kernel because of a system call, or an interrupt, the system may store the process state (its registers mostly) and jump to the previously saved state for another process. Doing this quickly, with the amazingly fast processors we have today, makes it appear that all processes can run at the same time. Each process is given a small ammount of processor time, before the system decides to jump to another. This can be 100ms, which is a very long time regarding the number of machine instructions that you can execute in that time.

Only one process is **running** at a time, and many others may be **ready** to run. These are two process states, see figure 2.4. The running process becomes ready when the system terminates its time in the processor. Then, the system picks up a ready process to become the next running one. States are just constants defined by the system to cope with the process abstraction.

Many times, a process would be reading from a terminal, or from a network connection, or any other device. When this happens, the process has to wait for input to come. The process could wait by using a loop, but that would be a waste of the processor. The idea is that while one process is waiting for input (or output!) to happen, we can put another process to run. Input/output devices are so slow compared with the processor that we could execute a lot of code for other processes while one is waiting. The time the processor needs to execute some instructions, compared to the time needed by I/O devices to perform their job, is like the time you need to move around in your house and the time you need to go to the moon.

This idea is central to the concept of **multiprogramming**, which is the name given to the technique that allows multiple programs to be loaded at the same time on a computer.

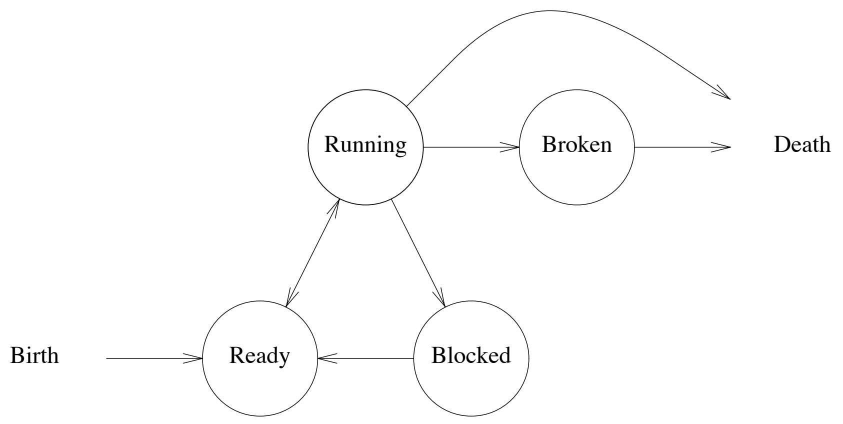

Figure 2.4: *Process states and transitions between them.*

To let one process wait out of the processor, without considering it as a candidate to be put into the running state, the process is flagged as **blocked**. This is yet another process state. All the processes listed above where blocked. When the event a blocked process is waiting for happens, the process state is changed to ready. Sometime in the future it will be selected for execution in the processor.

In Plan 9, the state shown for blocked processes reflects the reason that caused the process to block. That is why *ps* shows many different states. They are a help to let us know what is happening to our processes.

There is one last state, **broken**, which is entered when the process does something illegal (i.e., it suffers an error). For example, dividing by zero or dereferencing a null pointer causes a hardware exception (an error). Exceptions are dealt with by the hardware like interrupts are, and the system is of course the handler for these exceptions. Upon this kind of error, the process enters the broken state. A broken process will never run. But it will be kept hanging around for debugging until it dies upon user request (or because there are too many broken processes).

2.7. Debugging

When we make a mistake, and a running program enters the broken state, it is useful to see what happen. There are several ways of finding out what happen. To see them, let’s write a program that crashes. This program says hello to the name given as an argument, but it does not check that the argument was given, nor does it use the appropriate format string in `print`.

hi.c

```
#include <u.h>
#include <libc.h>

void
main(int, char*argv[])
{
        /* Wrong! */
        print("hi ");
        print(argv[1]);
        exits(nil);
}
```

When we compile this program and execute it, this happens:

```
; 8.hi
8.hi 788: suicide: sys: trap: fault read addr=0x0 pc=0x000016ff
```

The last line is a message printed by the shell. It was waiting for `8.hi` to terminate its execution. When it terminated, the shell saw that something bad happen to the program and printed the diagnostic so we could know. If we print the value of the `status` variable, we see this

```
; echo $status
8.hi 788: sys: trap: fault read addr=0x0 pc=0x000016ff
```

Therefore, the *legacy*, or exit status, of `8.hi` is the string printed by the shell. This status does not proceed from a call to `exits` in `8.hi`, we know that. What happen is that we tried to read the memory address 0x0. That address is not within any valid memory segment for the process, and reading it leads to an error (or exception, or fault). That is why the status string contains `fault read addr=0x0`. The status string starts with the program name and the process pid, so we could know which process had a problem. There is more information, the program counter when the process tried to read 0x0, was 0x000016ff. We do some post-mortem analysis now.

The program `src` knows how to obtain the source file name and line number that corresponds to that program counter.

```
; src -n -s 0x000016ff 8.hi
/sys/src/libc/fmt/dofmt.c:37
```

We gave the name of the binary file as an argument. The option `-n` causes the source file name and line to be printed. Otherwise `src` would ask your editor to display that file and line. Option `-s` permits you to give a memory address or a symbol name to locate its source. By the way, this program is an endless source of wisdom. If you want to know how to implement, say, `cat`, you can run `src /bin/cat`.

Because of the source file name printed, we know that the problem seems to be within the C library, in `dofmt.c`. What is more likely? Is there a bug in the C library or did we make a mistake when calling one of its functions? The mistery can be solved by looking at the stack of the broken process. We can read the process stack because the process is still there, in the broken state:

```
; ps
  ...many other processes...
nemo            788    0:00   0:00        24K Broken   8.hi
;
```

To print the stack, we call the debugger, `acid`:

```
; acid 788
/proc/788/text:386 plan 9 executable

/sys/lib/acid/port
/sys/lib/acid/386
acid:
```

and when we get its prompt, we ask for a stack dump:

```
acid: stk()
dofmt(fmt=0x0,f=0xdfffef08)+0x138 /sys/src/libc/fmt/dofmt.c:37
vfprint(fd=0x1,args=0xdfffef60,fmt=0x0)+0x59 /sys/src/libc/fmt/vfprint.c:30
print(fmt=0x0)+0x24 /sys/src/libc/fmt/print.c:13
main(argv=0xdfffefb4)+0x12 /usr/nemo/doc/os/9intro/progs/hi.c:8
_main+0x31 /sys/src/libc/386/main9.s:16
acid:
```

The function `stk()` dumps the stack. The program crashed while executing the function `dofmt`, at file `dofmt.c`. This function was called by `vfprint`, which was called by `print`, which was called by `main`. As you can see, the parameter `fmt` of `print` is zero! That should never happen, because `print` expects its first parameter to be a valid, non-null, string. That was the bug.

We can gather much more information about this program. For example, to obtain the values of the local variables in all functions found in the stack

```
acid: lstk()
dofmt(fmt=0x0,f=0xdfffef08)+0x138 /sys/src/libc/fmt/dofmt.c:37
        nfmt=0x0
        rt=0x0
        rs=0x0
        r=0x0
        rune=0x15320000
        t=0xdfffee08
        s=0xdfffef08
        n=0x0
vfprint(fd=0x1,args=0xdfffef60,fmt=0x0)+0x59 /sys/src/libc/fmt/vfprint.c:30
        f=0x0
        buf=0x0
        n=0x0
print(fmt=0x0)+0x24 /sys/src/libc/fmt/print.c:13
        args=0xdfffef60
main(argv=0xdfffefb4)+0x12 /usr/nemo/doc/os/9intro/progs/hi.c:8
_main+0x31 /sys/src/libc/386/main9.s:16
```

When your program gets broken, using `lstk()` in `acid` is invaluable. Usually, that is all you need to fix your bug. You have all the information about what happen from `main` down to the point where it crashed, and you just have to think a little bit why that could happen. If your program was checking out for errors, things can be even more easy, because in many case the error diagnostic printed by the program may suffice to fix up the problem.

One final note. Can you see how `main` is not the main function in your program? It seems that `_main` in the C library called your suposssedly `main` function.

The last note about debugging is not about what to do after a program crashes, but about what to do *before*. There is a library function called `abort`. This is its code

```
void
abort(void)
{
        while(*(int*)0)
                ;
}
```

This function dereferences a nil pointer! You know what would happen to the miserable program calling `abort`. It gets broken. While you program, it is very sensible to prepare for things that in theory would not happen. In practice they will happen. One tool for doing this is `abort`. You can include code that checks for things that should never happen. Those things that you know in advance that would be very hard to debug. If your code detects that such things happen, it may call `abort`. The process will enter the broken state for you to debug it before things get worse.

2.8. Everything is a file!

Whe have seen two abstractions that are part of the baggage that comes with processes in Plan 9: Processes themselves and environment variables. The way to use these abstractions is to perform system calls that operate on them.

That is nice. But Plan 9 was built considering that it is natural to have the machine connected to the network. We saw how your files are not kept at your terminal, but at a remote machine. The designers of the system noticed that files (another abstraction!) were simple to use. They also noticed that it was well known how to engineer the system to permit one machine use files that were kept at another.

Here comes the idea! For most abstractions provided by Plan 9, to let you use your hardware, a **file interface** is provided. This means that the system lies to you, and makes you believe that many things, that of course are not, are files. The point is that they *appear* to be files, so that you can use them as if that was really the case.

The motivation for doing things this way is that you get simple interfaces to write programs and use the system, and that you can use also these files from remote machines. You can debug programs running at a different machine, you can use (almost) anything from any other computer running Plan 9. All you have to do is to apply the same tools that you are using to use your real files at your terminal, while keeping them at a remote machine (the file server).

Consider the time. Each Plan 9 machine has an idea of what is the time. Internally, it keeps a counter to notice the time passing by and relies on a hardware clock. However, for a Plan 9 user, the time seems to be a file:

```
; cat /dev/time
 1152301434   1152301434554319872        ...
```

Reading `/dev/time` yields a string that contains the time, measured in various forms: Seconds since the epoch (since a particular agreed-upon point in time in the past), nanoseconds since the epoch, and clock ticks since the epoch.

Is `/dev/time` a real file? Does it exist in your disk with rest of the files? Of course not! How can you keep in a disk a file that contains the *current* time? Do you expect a file to change by some black magic so that each different nanosecond it contains the precise value that matches the current time? What happens is that when you read the file the system notices you are reading `/dev/time`, and it knows what to do. To give you the string representing the current system time.

If this seems confussing, think that files are an abstraction. The system can decide what reading a file means, and what writing a file means. For real files sitting on a disk, the meaning is to read and write data from and to the disk storage. However, for `/dev/time`, reading means obtaining the string that represents the system time. Other operating systems provide a `time` system call that returns the time. Plan 9 provides a (fake!) file. The function described in *time*(2), reads this file and returns the integer value that was read.

Consider now processes. How does *ps* know which processes are in the system? Simple. In Plan 9, the `/proc` directory does not exist on disk either. It is a virtual (read: fake) directory that represents the processes running in the system. Listing the directory yields one file per process:

```
; lc /proc
1       1320    2       246     268     30      32      348
10      135     20      247     269     300     320     367
 ...
```

But these files are not real files on a disk. They are the *interface* for handling running processes in Plan 9. Each of the files listed above is a directory, and its name is the process pid. For example, to go to the directory representing the shell we are using we can do this:

```
; echo $pid
938
; cd /proc/938
; lc
args    fd      kregs   note    notepg  proc    regs    status  wait
ctl     fpregs  mem     noteid  ns      profile segment text
```

These files provide the interface for the process with pid 938, which was the shell used. Many of these (fake, virtual) files are provided to permit debuggers to operate on the process, and to permit programs like *ps* gather information about the process. For example, look again at the first lines printed by `acid` when we broke a process in the last section:

```
; acid 788
/proc/788/text:386 plan 9 executable
```

Acid is reading `/proc/788/text`, which *appears to be* a file containing the binary for the program. The debugger also used `/proc/788/regs`, to read the values for the processor registers in the process, and `/proc/788/mem`, to read the stack when we asked for a stack dump.

Besides files intented for debuggers, other files are for you to use (as long as you remember that they are not files, but part of the interface for a process). We are now in position of killing a process. If we write the string `kill` into the file named `ctl`, we kill the process. For example, this command writes the string `kill` into the `ctl` file of the shell where you execute it. The result is that you are killing the shell you are using. You are not writing the string `kill` on a disk file. Nobody would record what you wrote to that file. The more probable result of writting this is that the window where the shell was running will vanish (because no other processes are using it).

```
; echo kill >/proc/$pid/ctl
    ... where is my window? ...
```

We saw the memory layout for a process. It had several segments to keep the process memory. One of the (virtual) files that is part of the process interface can be used to see which segments a process is using, and where do they start and terminate:

```
; cat /proc/$pid/segment
Stack     defff000 dffff000    1
Text   R  00001000 00016000    4
Data      00016000 00019000    1
Bss       00019000 0003f000    1
```

The stack starts at 0xdefff000, which is a big number. It goes up to 0xdffff000. The process is not probably using all of this stack space. You can see how the stack segment does *not* grow. The physical memory actually used for the process stack will be provided by the operating system on demand, as it is referenced. Having virtual memory, there is no need for growing segments. The text segment is read-only (it has an `R` printed). And four processes are using it! There must be four shells running at my system, all of them executing code from `/bin/rc`.

Note how the first few addresses, from 0 to 0x0fff, are not valid. You cannot use the first 4K of your (virtual) address space. That is how the system catches null pointer dereferences.

We have seen most of the file interface provided for processes in Plan 9. Environment variables are not different. The interface for using environment variables in Plan 9 is a file interface. To know which environment variables we have, we can list a (virtual) directory that is invented by Plan 9 to represent the interface for our environment variables. This directory is `/env`.

```
; lc /env
 '*'                cpu               init              planb      sysname
0                   cputype           location          plumbsrv   tabstop
MKFILE              disk              menuitem          prompt     terminal
afont               ether0            monitor           rcname     timezone
apid                facedom           mouseport         role       user
auth                'fn#sigexit'      nobootprompt      rootdir    vgasize
bootdisk            font              objtype           sdC0part   wctl
bootfile            fs                part              sdC1part   wsys
cflag               home              partition         service
cfs                 i                 path              status
cmd                 ifs               pid               sysaddr
;
```

Each one of these (fake!) files represents an environment variable. For you and your programs, these files are as real as those stored in a disk. Because you can list them, read them, and write them. However, do not search for them on a disk. They are not there.

You can see a file named `sysname`, another named `user`, and yet another named `path`. This means that your shell has the environment variables *sysname*, *user*, and *path*. Let's double check:

```
; echo $user
nemo
; cat /env/user
nemo;
```

The *file* `/env/user` appears to contain the string `nemo`, (with no new line at the end). That is precisely the value printed by *echo*, which is the value of the *user* environment variable. The implementation of *getenv*, which we used before to return the value of an environment variable, reads the corresponding file, and returns a C string for the value read.

This simple idea, representing almost everything as a file, is very powerful. It will take some ingenuity on your part to fully exploit it. For example, the file `/dev/text` represents the text shown in the window (when used within that window). To make a copy of your shell session, you already know what to do:

```
; cp /dev/text $home/saved
```

The same can be done for the image shown in the display for your window, which is also represented as a file, `/dev/window`. This is what we did to capture screen images for this book. The same thing works for any program, not just for `cp`, for example, `lp` prints a file, and this command makes a hardcopy of the whole screen.

```
; lp /dev/screen
```

Questions

1 Why was not zero the first address used by the memory image of program `global`?

2 Write a program that behaves like *echo*(1).

3 What would print `/bin/ls /blahblah` (given that `/blahblah` does not exits). Would `ls /blahblah` print the same? Why?

4 What happens when we execute

```
; cd
;
```

after executing this program. Why?

```
#include <u.h>
#include <libc.h>
void
main(int, char*[])
{
        putenv("home", "/tmp");
        exits(nil);
}
```

5 What would do these commands? Why?

```
; cd /
; cd ..
; pwd
```

6 How can we know the arguments given to a process that has been already started?

7 What could we do if we want to debug a broken process tomorrow, and what to power off the machine now?