C Programming in Plan 9 from Bell Labs

Pietro Gagliardi

ABSTRACT

This paper is an introduction to programming with Plan 9 from Bell Labs with the C language. Plan 9 provides not only a significantly improved version of C, but also a number of programming libraries to simplify complicated tasks. This paper is meant to be a supplement to the manual pages, other documents provided by the system in /sys/doc, and a programmers literature collection.

1. Introduction

Plan 9 from Bell Labs has always been a system above the rest: simple, portable, and feature-complete. It isnt UNIX;® rather, it improves on the basics of UNIX by providing a number of features absent from most other operating systems. One of those features is a great programming environment that rivals UNIXs. Plan 9 is fully Unicode-conformant through its nearly universal use of the UTF-8 encoding, brought to us by two of the people that brought us Plan 9. It not only keeps the C language of old, but through the work of Ken Thompson, it provides a C that makes some otherwise complicated construct straightforward. Backing this new C up is 33 programmability libraries that significantly reduce the amount of code a programmer needs to write. And every single line of this code is fully portable among different Plan 9 installations, even with different architectures: the notion of a configure script has been vanquished at last.

Learning programming with Plan 9 is not something that requires complicated textbooks and four years of college study to master. In fact, with just the manual pages and pages of some documentation in hand, someone can quickly master the core concepts. However, there sometimes is a need of a starters guide or tutorial to start with or to clear up some uncertainty. That task is what this paper aims to do. This paper is *not* a full reference to Plan 9s programming environment: the manual pages do that. Keep this in mind while you read.

You need to know how to use Plan 9 from Bell Labs, rc, an editor such as sam or acme, and the C programming language to start. The official guide to C is Prof. Brian Kernighan and Dennis Ritchies *The C Programming Language*, now in its second edition. Read through it: youll learn quite a lot.

2. Core Concepts

Here is Kernighans "hello, world"-printing program that has become quintessential, in Standard C and with a few differences from the one in Kernighans book (for exposition purposes).

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

Now here it is as a Plan 9 programmer would write it.

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

void
main()
{
    print("hello, world\n");
    exits(0);
}
```

Immediately, expert C programmers will say things like Where did stdio go? and shout at the top of their lungs things like You can't declare main as returning void! If you're one of these guys, then you better get used to it.

The include file u.h, stored in /$objtype/include where $objtype is an environment variable storing the current CPU name, contains CPU-specific definitions. All header files in Plan 9 use this, so it must be included first. Next comes libc.h, stored in /sys/include. libc.h contains the definitions for the C library, which is linked into every Plan 9 program. The C library consists of several parts:

- All the Plan 9 system calls (save for a few that only the library uses)
- A set of subroutines to facilitate using the system calls
- The formatted print routines
- Mathematical functions
- Time functions
- Functions for working with Unicode characters, or Runes

libc must be second; it is used by most, if not all, other libraries.

The print function is a member of the set of formatted print routines; it works just like printf in C, with several minor differences:

- The %u format is gone; it has been replaced with the u modifier to other integer formats. So instead of saying %-3lu, you say %-3uld.
- The %b format is provided for printing binary numbers.
- The %C and %S formats are provided for printing UTF-8 characters, called Runes, and strings of Runes, respecitvely. They are discussed later.
- The ll modifier flag to integral formats prints vlongs, which are described later.
- The %r format prints the error string, which is described next.
- You can create your own formats; that is described later.

Otherwise, print behaves the same as printf.

exits and the void return from main require a bit of explanation. The traditional way of representing errors and status returns in C is with numbers: a return from main or the argument to exit represents a status return from a program, and errno stores information about error returns from functions. The traditional behavior is to have zero mean no error and any other value mean error; ANSI C defines EXIT_SUCCESS and EXIT_FAILURE for status returns from programs.

This gets restricted very quickly. ANSI C only defines three standard values for errno (domain error, range error, and illegal multibyte sequence) and two values for status return. And sometimes an integer won't tell you enough. For example, let's take the UNIX lseek system call, which manipulates the file read/write position:

```
long lseek(int fd, long offset, int from);
```

If any argument is invalid (for example, from not 0, 1, or 2), lseek returns with errno set to EINVAL (specific to UNIX). But this doesn't tell you *which* argument was invalid, or how many; it only says that something was not right. We can add the appropriate errno values to resolve this problem. But what about a library that defines over 1,000 values for errno? On machines with small int sizes, this chokes your program and defeats the purposes of both sides.

A better idea is to give the programmer the ability to handle any error that comes in without worry of losing standards compliance or clarity, and to generate any error without falling into a surfeit of possibility. So the designers of Plan 9 decided to use strings instead of numbers. Each program has an *error string* which is set by routines when an error occurs. And each program returns a string to the host environment with the exits system call. The value given to exits can be accessed from rc through the environment variable $status.

So with a string, how do you represent a lack of error? Why, with a null pointer or null string. Because the constant 0 turns into a null pointer, the statement

```
exits(0);
```

does everything already. Of course you can also say

```
exits(nil);
```

or

```
exits("");
```

nil, in u.h, is Plan 9's NULL.

So how does this explain why main has to return void? You can't return a string placed in automatic storage from a function:

```
char *
f(void)
{
	auto char s[] = "hello";

	return s; /* WRONG */
}
```

But a programmer may store the exit status of a program in this way.

An Aside on Style

Plan 9 programs are usually written to conform to a predefined set of style guidelines, described in the manual page *style*(6), for the sake of uniformity. Here is a taste:

```
static
int
func(int f, char *g[])
{
     int i, j;

     j = 5;
     acquirelock();
     for(i = 0; i < j; i++){
          process(i, &j);
          if((j = g(&i)) == 0 ? h() : i()) /* g() affects h()/i() */
               if(strcmp(s, t) == 0)
                    something();
     }
     return j - i;
}
```

Of course this piece of code doesn't do anything sensible by itself. It was written to show the basics of this style. If you want to contribute to Plan 9, be sure to use this style. Of course, you can still use your favorite style elsewhere.

3. Compiling Programs

UNIX compilers give you the option of compiling a program in one shot:

```
$ cc a.c b.c      # compile and link; creates a.out
$ a.out           # run
```

or in pieces:

```
$ cc -c a.c            # compile; creates a.o
$ cc -c b.c            # compile
$ cc a.o b.o -lS       # link; creates a.out. you can also use ld and omit -lS
$ a.out                # run
```

Plan 9 gives you no choice but to do the latter, but with ld instead of cc for the final stage. On top of that, there is no single C compiler and no single linker; there is one of each for each supported processor architecture.

What are the benefits to this requirement? First, large projects can be built with ease, just like make. (Plan 9 provides an improved variant, called mk, that I describe later.) Second, it removes one possible error: mixing computer architectures. Third, it promotes separation of tasks: the C compiler should not be expected to link.

Using this system is easy. All you have to know is the single character that denotes your processor. For the Intel x86 family that is in most PCs, that character is 8. So I do:

```
% 8c a.c          # compile; creates a.8
% 8c b.c          # compile
% 8l a.8 b.8      # link; creates 8.out
% 8.out           # run
```

A complete list is in the manual page for the C compilers, *2c*(1).

Also note that a special feature of the C compilers allows the linker to detect that libc or another Plan 9 library is to be linked into the program without any extra flags. I will get to that later.

4. Manipulating Files

In Plan 9, absolutely everything is a file: even processes (/proc), environment variables (/env), and file descriptors (/fd)! What is a file descriptor? A file descriptor is an integer that represents an open file. Files are opened with the open system call, which returns one. The syntax of open is

```
int open(char *filename, int openmode);
```

openmode is one of the constants OREAD, OWRITE, or ORDWR, which define what you intend to do with this file (read, write, or both), optionally combined with the constants OTRUNC, OCEXEC, and ORCLOSE via bitwise OR (|). If OTRUNC is given with OWRITE or ORDWR, the file is truncated to zero length. OCEXEC and ORCLOSE are described later. open returns a valid file descriptor @n@ such that @n >= 0@ on success, or -1 on failure.

It is an error to open a file that doesn't exist, so the create system call is used to create one (Ken got his wish). create takes the form

```
int create(char *filename, int createmode, int permissions);
```

If the file already exists, it is truncated to zero length. The permissions are just as in UNIX: a three-digit octal number containing a combination of read, write, or execute bits for the file's owner, the group of the owner, and everyone else. For example, 0644 yields rw-r--r--, and 0750 yields rwxr-x---. createmode is either 0 or a bitwise OR of DMDIR, which creates a directory, DMAPPEND, which makes a file that can only be appended to (i.e. a log file), DMEXCL, which makes the file openable by only one program at a time, or OEXCL, which will cause create to fail if the file exists.

The read and write system calls read and write arbitrary data to the files:

```
long read(int fd, void *buf, long n);
long write(int fd, void *buf, long n);
```

read n bytes from fd into buf and write n bytes from fd into buf, respectively. read returns the number of bytes read, while write returns the number of bytes written.

Why do read and write seem to return their argument n? The truth is, they don't always do so. Lets take read as an example. What if the end of the file is reached before anything was read? Well, you read nothing, so read will appropriately return 0. A write can fail if the disk is full.

Instead of using the low-level write, you can use fprint. fprint, like fprintf, allows formatted output to an open file. It takes, as an extra first argument, the appropriate file descriptor. Note that there are no reading functions like scanf; buffered I/O via libbio, described later, provides the facilities.

The seek system call changes where reads and writes are performed in relation to the file.

```
vlong seek(int fd, vlong amount, int from);
```

If from is 0, seek amount from the start of the file. If 1, seek from the current position. If 2, seek from the end. Note that amount goes to the right if positive and left if negative regardless of from, so to seek five characters before the end, you say

```
seek(fd, -5, 2);
```

seek returns the position from the start regardless of from. On error, seek seems to succeed; only by examining the error string can you detect an error. seek fails on directories and does nothing on pipes.

What is vlong? It is a typedef-ed alias to long long. The C compilers, as well as C99, provide the long long type, which provides access to very long integer values, often 64 bits. There is also an unsigned variant. u.h provides the terse alias uvlong. On a 32-bit processor like the x86, 64-bit values are simulated. For instance, you can do

```
vlong  v;

switch(v){
case  a:
     /*  ...  */
}
```

However, the mere fact that 64-bit values are available is promising.

Finally, the close system call says that you are done with a file you opened or created. It takes the form

```
int  close(int  fd);
```

close should only fail (return -1) if fd is not really open, so just ignore its return value.

Before I move on, I need to talk about three file descriptors that all programs have when they are created. File descriptor 0 is *standard input*, which is the keyboard by default and changed with rcs <, <<, <{ ...}, and |. File descriptor 1 is *standard output*, which is either the screen or the current rio window by default and changed with rcs >, >>, and |. So

```
print("hello");
```

is the same as

```
fprint(1,  "hello");
```

File descriptor 2 is *standard error*. This allows you to give the user emergency output in the case of an error, without fear of losing the error to redirected output. Standard error can be redirected with the [2] modifier to the output redirection operators in rc.

5. UTF-8 Support

Plan 9 supports Unicode via UTF-8, however you need special provisions for handling the extended characters. The special type Rune is large enough to store a UTF-8 character, which can be embedded into a C program using Standard C s wide character literal format L'*character*'. A string of Rune s can be made in the same way as a string of characters, and has the type "array of Rune s." Most Rune s can be entered directly from the keyboard; see *keyboard* (6) for instructions and the file /lib/keyboard for a complete list and their key codes.

A UTF-8 character or string can be output with the %C and %S formats to the print routines, respectively. For example,

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

void
main()
{
     print("3  %C  4\n",  L'd');
     print("%S\n",   L"ρχιμ®δης");  /*  Archimedes  */
}
```

The codes for capital alpha and lowercase eta with tonos (Unicode 0386 and 03AE, respectively) cannot be entered with the keyboard; they were generated with a simple program:

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

void
main(int argc, char *argv[])
{
	if(argc != 2){
		fprint(2, "usage: %s hex-code\n", argv[0]);
		exits("usage");
	}
	print("%C\n", (Rune)strtol(argv[1], nil, 16));
	exits(0);
}
```

argc, argv, and strtol act as in standard C. If this program is compiled as code2rune, you can say

```
% code2rune 0386

% code2rune 41
A
```

A Rune can be constructed from at least one char. This allows input of Runes by reading a char and seeing if it can be used to begin a Rune. This is a simple multi-step process:

1. Read a character.
2. If that character is less than the constant Runeself, then cast that character to a Rune and return it. Otherwise, store that character in the first position of a buffer.
3. Read the next character into the next buffer position.
4. If the buffer from beginning to the current position is a full Rune, return that Rune. Otherwise, return to step 3.

The function fullrune does the test in step 4.

```
int fullrune(char *buf, int n);
```

returns a nonzero (true) value if the n characters pointed to by buf make up a full Rune. The function chartorune does the actual conversion:

```
int chartorune(Rune *dest, char *src);
```

turns the data pointed to by src into the Rune stored at *dest and returns the number of bytes of src used. On error, it returns 1 and stores the Runeerror constant in *dest. The number of bytes shall never exceed UTFmax, a constant that defines how many possible bytes may be in a Rune.

With all this in mind, we can write a function that uses read to read in a single Rune from a given file descriptor and returns the number of characters read. It behaves similarly to chartorune: on error it returns the number of bytes read, but stores Runeerror.

```
long
readrune(int fd, Rune *r)
{
	char buf[UTFmax];
	char c;
	long nread, n;
	int i;

	if((nread = read(fd, &c, 1)) != 1){
		*r = Runeerror;
		return nread;
	}
	if(c < Runeself){
		*r = (Rune)c;
		return nread;
	}
	buf[0] = c;
	for(i = 1;;){
		if((n = read(fd, &c, 1)) != 1){
			*r = Runeerror;
			return nread;
		}
		nread += n;
		buf[i++] = c;
		if(fullrune(buf, i)){
			chartorune(r, buf);
			return nread;
		}
	}
}
```

We can test this out in a program that reads Rune s and prints them out, buffering the output.

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

void
main()
{
	Rune rs[100];
	int i;

	i = 0;
	while(readrune(0, &rs[i]) > 0)
		if(rs[i] == L'\n'){
			rs[i] = '\0';
			print("%S\n", rs);
			i = 0;
		}else
			i++;
	exits(0);
}
```

Lets try this out:

```
% readrune
a
a
abc
abc
3d4
3???4
d
???
ctl-d%
```

Something seems to be amiss. For every Unicode character I put in, something gets eaten up and a mess of "I don t have that glyph" symbols (Peter Weinberger s famous headshot) comes up. Our problem is declaring c in readrune as a char ; if we change it to uchar (a synonym for unsigned char), then we get this interactive session:

```
% readrune
3d4
3d4
d+-4556
d+-4556
ctl-d%
```

There s still a problem. Consider

```
% xd -c -b bad
0000000   e0  Q  R  S \n
      0   e0 51 52 53 0a
0000005
% cat bad
?QRS
% readrune < bad
?S
%
```

Obviously incorrect. The ? means "this is not a valid Rune ." It turns out that even though fullrune may report that the buffer contains a Rune , it does not say that the Rune is valid. In these situations, chartorune may give up, returning a number of characters converted *less* than the number of characters read! This means we ate too much. Fortunately, and if we re not reading a pipe or directory, we can fix this with the use of seek . Change the last if to

```
        if(fullrune(buf, i)){
            n = chartorune(r, buf);
            while(i > (int)n){
                seek(fd, -1, 1);
                i--;
                nread--;
            }
            return nread;
        }
```

and everything works:

```
term% readrune < bad
?QRS
```

The seek used says to seek -1 characters forward from the current position, or one character back. In effect, this is ungetc from Standard C, except that it doesn t work on pipes or directories. There are other common uses of seek :

```
seek(fd, 0, 0);
```

seeks to the beginning of a file,

```
pos = seek(fd, 0, 1);
```

doesn't change the file position but tells you where, from the beginning, you are, and

```
seek(fd, 0, 2);
```

goes to the end. This is done by default when opening a file that is append-only for writing.

6. Buffered I/O

Let us write a program runecount that counts the number of Runes in a file. The standard wc doesn't do this; it counts the number of bytes. I have omitted the definition of readrune.

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

long readrune(int, Rune *);

uvlong
runecount(int fd, char *filename)
{
	uvlong n;
	Rune r;

	n = 0;
	while(readrune(fd, &r) != 0)
		n++;
	print("%10ulld %s\n", n, filename);
	return n;
}

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

	total = 0;
	if(argc == 1)
		runecount(0, "");
	else{
		for(i = 1; i < argc; i++){
			fd = open(argv[i], OREAD);
			if(fd == -1)
				fprint(2, "can't open %s: %r\n", argv[i]);
			else{
				total += runecount(fd, argv[i]);
				close(fd);
			}
		}
		if(argc > 2)
			print("%10ulld total\n", total);
	}
	exits(0);
}
```

The file /lib/glass is a perfect file to test this program on; it contains translations of the phrase I can eat glass and it doesn't hurt me in many languages and using Unicode characters. For example,

```
% grep '^(French|Russian|Greek):' /lib/glass
Greek: Μπορî να φ¬ω σπασµ-να γυαλι¬ χωρ¯ς να π¬θω τ¯ποτα.
French: Je peux manger du verre, ça ne me fait pas de mal.
Russian: / <>3C 5ABL AB5:;>, >=> <=5 =5 2@548B.
Greek: Μπορî να φ¬ω σπασµ-να γυαλι¬ χωρ¯ς να π¬θω τ¯ποτα.
```

To compare runecount with wc, let's try them out:

```
% runecount /lib/glass
      6715 /lib/glass
% wc -c /lib/glass
   8517 /lib/glass
```

So /lib/glass has 6,715 Runes that fill up 8,517 bytes.

It turns out that when running runecount, I had to wait a while before getting any output, while wc returned immediately. The time program will tell me how long a program runs, so let's try it on runecount:

```
% time runecount /lib/glass
      6715 /lib/glass
0.02u 1.24s 6.74r     runecount /lib/glass
```

This tells that the program took 6.74 seconds to run, with 1.24 seconds in the kernel, 0.02 seconds in user space (that is, main, runecount, and readrune), and the rest doing various other things that I really don't know about (sorry!).

wc is faster than readrune because it buffers its input. A *buffer* is an in-memory array of a number of data objects. When you ask to acquire a character from a character buffer tied to a file, it first sees if there is a character in the buffer. If there is a character, the character is removed from the buffer and returned to the user. If not, then the buffer is filled by reading enough characters to occupy every element of the buffer array, and the first character in the buffer is removed. readrune, however, does no buffering, so a new character has to be read every time.

Fortunately, Plan 9 provides not one but two ways of buffering input and output. The first is libstdio, which works just like in Standard C. But this doesn't support Runes, so we can't use it. It also has several other restrictions that I'll not go into.

The second is libbio, with manual page *bio*(2). libbio is a library for buffering input and output in much the same way as libstdio, but provides a higher level of abstraction and full Rune support. In fact, our readrune function is based on libbio's equivalent function. To put libbio into your program, just do the following:

```
#include <bio.h>
```

This must follow the #include of libc.h.

The next step is to make a new Biobuf, which is the libbio equivalent to FILE. Note that I did not say that it was equivalent to FILE *. This is because there are *two* ways to connect a file to a Biobuf, with each method working differently. The first method actually opens a file:

```
Biobuf *Bopen(char *filename, int openmode);
```

openmode is either OREAD, to indicate reading, or OWRITE, which creates the file with mode 0666 (rw-rw-rw-). It returns a pointer to a dynamically allocated Biobuf, or nil if error.

You can also connect a Biobuf to an already open file:

```
int Binit(Biobuf *bp, int fd, int mode);
```

bp is a pointer to an already allocated Biobuf , either created explicitly by the compiler or dynamically allocated with malloc . The entire malloc family of routines is provided by the C library. In this case, openmode is the same as in Bopen , except OWRITE does not create the file. It returns the constant Beof on error. You can use this function to wrap the standard file descriptors to libbio; this is the only way to do formatted reads from standard input, since Plan 9 does not provide a scanf equivalent.

Once we have a Biobuf open, we can use several functions to read and write to them. But first, a brief note on an extension of Plan 9 s C: basic inheritance is supported. The structure Biobuf has the properties of another structure named Biobufhdr , to the point that all Biobuf needs is to have all the elements of Biobufhdr and the buffer itself. A pointer to a Biobuf can be used as a pointer to a Biobufhdr . This feature will be described in full when we talk about the lock family of routines.

The basic input functions provided by libbio are numerous and useful:

```
long Bread(Biobufhdr *bp, void *buf, long n);
void *Brdline(Biobufhdr *bp, int delim);
char *Brdstr(Biobufhdr *bp, int delim, int nulldelim);
int Blinelen(Biobufhdr *bp);
int Bgetc(Biobufhdr *bp);
long Bgetrune(Biobufhdr *bp);
int Bungetc(Biobufhdr *bp);
int Bungetrune(Biobufhdr *bp);
int Bgetd(Biobufhdr *bp, double *d);
```

Bread behaves just like read . Brdline returns either a full buffer or everything up to the given delimiter. A more useful function is Brdstr , which returns a malloc -ed string consisting of the next full line ending with the given delimiter, or nil on failure. If nulldelim is nonzero, the delimiter is not included in the returned string. This eliminates the need for idioms like

```
s[strlen(s) - 1] = '\0';
```

In both cases, the function Blinelen returns the length of the returned line.

Bgetc and Bgetrune read and return the next character and Rune on the file, respectively. Both return Beof on end of file, hence the long return from Bgetrune . They can be returned to the buffer with the equivalent unget functions. Finally, Bgetd reads in a double , returning -1 on failure or the number of bytes read on success.

The output routines are

```
long Bwrite(Biobufhdr *bp, void *buf, long n);
int Bputc(Biobufhdr *bp, int c);
int Bputrune(Biobufhdr *bp, long r);
int Bprint(Biobufhdr *bp, char *fmt, ...);
int Bvprint(Biobufhdr *bp, char *fmt, va_list v);
int Bflush(Biobufhdr *bp);
```

va_list , and the family of (Standard C) supporting routines, are provided; the standard routines vprint and vfprint are provided. Bflush immediately flushes the buffer; this is usually done when the buffer gets full. Everything else works as expected.

The Bseek function works like seek , but libbio provides an alternative to

```
loc = Bseek(bp, 0, 1);
```

in Boffset, which takes the Biobufhdr * and returns the offset via a long:

```
loc = Boffset(bp);
```

To close an open file, use

```
int Bterm(Biobufhdr *bp);
```

Bterm will not close files opened with Binit; this allows use of the standard file descriptors after Bterm on them.

Lets rewrite runecount to use libbio Note that we no longer need readrune given Bgetrune.

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

uvlong
runecount(Biobuf *f, char *filename)
{
	uvlong n;
	Rune r;

	n = 0;
	while((r = Bgetrune(f)) != (Rune)Beof)
		n++;
	print("%10ulld %s\n", n, filename);
	return n;
}

void
main(int argc, char *argv[])
{
	int i;
	uvlong total;
	Biobuf bstdin, *bfile;

	total = 0;
	if(argc == 1){
		if(Binit(&bstdin, 0, OREAD) == Beof){
			fprint(2, "can't connect stdin to bio: %r");
			exits("Binit");
		}
		runecount(&bstdin, "");
		Bterm(&bstdin);
	}else{
		for(i = 1; i < argc; i++){
			bfile = Bopen(argv[i], OREAD);
			if(bfile == nil)
				fprint(2, "can't open %s: %r\n", argv[i]);
			else{
				total += runecount(bfile, argv[i]);
				Bterm(bfile);
			}
		}
		if(argc > 2)
			print("%10ulld total\n", total);
	}
	exits(0);
}
```

and test it:

```
% 8c runecount.c
% 8l -o runecount runecount.8
% runecount /lib/glass
      6715 /lib/glass
% time runecount /lib/glass
      6715 /lib/glass
0.00u 0.01s 0.02r    runecount /lib/glass
```

Now the program is significantly faster, and it still yields the proper answer.

Given Bgetrune, is there a need for runecount? To be honest, this really depends on taste: one might argue that with libbio, we don't need to use the unbuffered read and we will be just fine with Bgetrune, while another might say that someone may want to use readrune() and therefore it should be preserved. I will kill readrune() in favor of Bgetrune. I am doing this for several reasons:

Most programs use libbio and avoid the low-level system calls altogether.

If a program uses the system calls, it won't poll a byte or a Rune at a time; it will just read an entire line or buffer.

Most functions deal with Runes implicitly, since a set of bytes makes up a Rune, and for those that don't, conversion and handling routines are so straightforward that they are used after input is read.

Feel free to disagree.

An Aside on Linking

The compilation process for runecount in the previous example was shown on purpose: it showed that you did not need an explicit linker flag to link to libbio. Of course, you could supply the libraries as arguments to the linker, in the form -l *ext*, where *ext* is the library name without the lib- prefix (-lbio, for example).

But the C compilers do this for you every time you include the appropriate header file. The C preprocessor reserves a special directive

```
#pragma text
```

where the *text* is implementation-defined. For Plan 9 C compiler, if *text* is of the form

```
lib "library"
```

then the library is automatically linked *exactly once* per program. For example,

```
% grep '^#pragma[ →]lib' /sys/include/libc.h
#pragma lib "libc.a"
```

(A tab in the command line is represented by →.) The file libc.a is part of a collection of library files in /$objtype/lib. The .a means that the library was made with the ar program; see *ar*(1).

7. Processes and Notes

Plan 9's process model, to the programmer, is very similar to UNIX's. You have fork, exec, and wait, but they have changed quite a bit. The system call is no longer fork but rfork, which is much richer and more powerful. And wait is now await, which allows you to get a more precise indication of what happened and how. fork and wait are still there, but wait is quite different. And Plan 9 has no notion of the signal; instead, it uses *notes*, which are strings.

The rfork system call is simple:

```
int rfork(int mode);
```

The `mode` is a bitmask of the following:

RFPROC — Make a new process. If not set, the mode is applied to the parent, allowing it to do things otherwise impossible. Few programs ever need to do so (for example, ar and rio do, for their own reasons).

RFNOWAIT — The parent cannot use the `await` system call or any related routines on the child.

RFNAMEG — The child inherits a copy of the parent's name space (see below). If neither this nor RFCNAMEG, the child shares the parent's name space.

RFCNAMEG — The child has a clean name space to start.

RFNOMNT — DIsallow the `mount` system call (described later) and access to special device directories (#*letter*).

RFENVG — Copy environment variables. Works the same as RFNAMEG.

RFCENVG — Start with no environment variables.

RFNOTEG — Child has its own *note group*, so notes sent to it and its children don't affect the parent.

RFFDG — Child's file descriptors are copied rather than shared.

RFCFDG — Child has no file descriptors, *not even standard ones*.

RFREND — Don't allow the child to rendezvous with the parent or its parents. The `rendezvous` system call is described below.

RFMEM — Child and parent share data and bss segments — that is, global and local variables and function call.

As you can see, `rfork` is a very powerful tool for controlling how a child behaves. (Parents may want to pray for a real-life `rfork`.) But for most purposes, all you want to do is make a child that has its own file descriptors and doesn't communicate with the parent: RFPROC|RFFDG|RFREND and that is what the routine `fork` does. Both return:

- -1 on error
- The child's process ID if the parent
- 0 if the child

and continue execution from where you left off. So you can say

```
switch(pid = rfork(RFPROC | RFFDG | RFNOTEG | RFENVG | RFNOWAIT | RFREND)){
case -1:
    sysfatal("rfork failed: %r");
case 0:
    child();
    exits(0);
}
parent();
exits(0);
```

The `sysfatal` routine, which has the syntax

```
void sysfatal(char *mesg, ...);
```

prints the formatted message on standard error and terminates with that message as the status return. If the global variable `argv0` is set, it will be displayed before the message. `argv0` should be set to `argv[0]` before programs mess with it; the command-line option macros we will see shortly do this for you.

Usually a `rfork` is followed by one of the `exec` routines, which allow a process to be replaced by another. The system call `exec`, which is similar to UNIX's `execv`:

```
void *exec(char *filename, char *argv[]);
```

replaces the current process with the one at `filename`, passing the given vector of arguments to the main routine's `argv`. The first argument (`argv[0]`) is the program's effective name, usually the name without path. The final argument must be a null pointer; this is used to find `argc`. The functions only return on failure and set the error string; the return value is insignificant. Therefore, you can say

```
exec(prog, args);
sysfatal("exec of %s failed: %r", prog);
```

`execl` is a subroutine of the form

```
void *execl(char *filename, ...);
```

It turns each of its optional arguments into a member of an `argv` array until a null pointer is seen, then calls `exec`. Beware:

```
execl(filename, nil);
execl(filename);    /* WRONG */
```

What denotes an executable file? The user must have both execute and read permissions enabled on the file (although the manual page for `exec` only states that execute is required), and the file cannot be a directory. The file is opened with the mode `OEXEC`, which opens to read but requires execute permissions, and the first two bytes are scanned. If the bytes are the characters `#!`, then the file is assumed to be text that is passed to another program. If the first line of `f`'s file is

```
#!/bin/rc
```

and `f` is called by

```
execl("f", "a", nil);
```

then the call to `execl` is, in effect,

```
execl("/bin/rc", "/bin/rc", "f", "a", nil);
```

Otherwise, the two bytes are put back and a `long` is read. If this does not equal the a.out magic number for the current CPU architecture (see *a.out*(6)), an error occurs. Otherwise, the program is executed.

The `await` system call, which has the form

```
int await(char *s, int n);
```

waits for a child that was not `rfork`-ed with the `RFNOWAIT` flag set to terminate. When this happens, the first `n` characters of a special string are stored in `s` and the function returns the length of the special string that was stored (in case `n` was too big), or -1 if there are no children to wait for. The special string is of the form

process-ID user-time system-time real-time 'status-return'

with spaces separating each field. The status return is blank for successful termination; the appearance is `''`. The times are reported in milliseconds. There is *no* `'\0'` at the end of this string, so be sure to add one in your code:

```
char buf[256];
int n;

if((n = await(buf, 255)) >= 0)
    buf[n + 1] = '\0';
```

The tokenize routine can be used to separate the individual fields:

```
int tokenize(char *str, char **array, int max);
```

str is the string to tokenize, wich is split into at most max elements of the array by overwriting certain delimiters with '\0'. The function returns the number of tokens actually split. The splitting rules are simple: split at white space, except treat quoted text as a single token. The quoting rules are the same as in rc:

center; lfCW l rfCW. becomes

So the code to split into the individual fields is simple:

```
char *fields[5], buf[256];
int n;

if((n = await(buf, 255)) < 0)
    sysfatal("await failed: %r");
buf[n + 1] = '\0';
if(buf[n] != '\'')
    sysfatal("buffer was too small to hold await's message");
tokenize(buf, fields, 5);
print("pid %s took %s milliseconds and returned %s\n", fields[0], fields[3],
    *fields[4] == '\0' ? "success" : fields[4]);
```

This is what the wait subroutine does.

```
Waitmsg *wait(void);
```

which waits for a process and returns a malloc-ed structure of type Waitmsg:

```
typedef struct Waitmsg Waitmsg;
struct Waitmsg{
    int pid;
    ulong time[3];
    char *msg;
};
```

where the fields are given in the same order that await does, so time[1] is system time. msg is allocated with malloc, but you can't use free since the malloc that was used is not what you think. You only have to free the Waitmsg, and everything else is fine. If you want to know the magic, see /sys/src/libc/9sys/wait.c. if you want an example of wait and Waitmsg, see the source for the time command at /sys/src/cmd/time.c.

What happens if the command is interrupted (you hit the interrupt key)? An interrupt usually kills the process by sending what's called a *note* to the process and all its children in the same *note group*. Forking the child to have the RFNOTEG flag set allows the child to handle its own notes independently from the parent. If time did this, however, it would be unable to report that the command had been interrupted.

There are many different types of notes. The most common are *interrupt*, *hangup*, which is sent when you disconnect from a CPU server, *alarm*, which is associated with the alarm system call, and *bad address*, which happens when you access invalid memory. If any of these notes are not handled, the program terminates.

How can you handle notes? rc allows you to define functions like sigint that get executed when the specific note gets processed. What really happens is rc registers its *note handler* to execute the function and return when the specific note is issued. The system calls notify and noted do this.

Unlike with UNIX signals, there is only one note handler function, which is registered with the notify system call:

```
int notify(void (*f)(void *, char *));
```

The argument is a pointer to a function defined as

```
void f(void *ureg, char *note)
```

The ureg argument is turned into a pointer to a structure of type Ureg, defined in /$objtype/include/ureg.h. Ureg contains the values of machine registers at the time the note was *posted*, and as such, is nonportable. Few, if any, programs ever need to use this structure and/or this argument to the handler. The second argument is the note string itself. If the function passed to notify is a null pointer, the default handler is restored. The return value is insignificant.

Note handlers follow special rules. They may not use floating-point operations, nor may they call functions that do. A note handler cannot return; it must either exit, use the noted system call, or call the notejmp routine. noted is of the form

```
int noted(int how);
```

how is NDFLT if you want the system to do the default action or NCONT if you want the system to go back to where the program left off. The return value is insignificant, as the note handler doesn't return. Also, jmp_buf, setjmp, and longjmp are provided, but you cannot longjmp from within a note handler. Instead, you use the safer notejmp routine, which works the same as longjmp.

If a note interrupts a system call and the note handler calls noted(NCONT), the system call terminates early with error string interrupted. This is very important, as it can be a cause of errors. Beware.

To send a process a note, use the postnote subroutine:

```
int postnote(int who, int pid, char *note);
```

If who is PNPROC, only the process is killed. But if it is PNGROUP, all the processes in the process group is killed, with the exception of the current process if it is in that group. This is a restriction of the operating system, not of postnote itself. On failure, postnote returns -1. A useful but undocumented note to post is kill, which terminates the process without giving it a fighting chance. This is actually what the kill command does:

```
term% kill rc
echo kill>/proc/2379/note    # rc
echo kill>/proc/4431/note    # rc
echo kill>/proc/5453/note    # rc
echo kill>/proc/6233/note    # rc
echo kill>/proc/6243/note    # rc
echo kill>/proc/6445/note    # rc
echo kill>/proc/6684/note    # rc
echo kill>/proc/7005/note    # rc
```

Piping that to rc will kill every rc, including the one you created in the pipe.

The alarm note involves an *alarm clock* that each process has (and only one per process). The alarm system call is of the form

```
long alarm(ulong ms);
```

ulong is a synonym for unsigned long. If its argument is 0, the alarm clock is cleared. Otherwise, the alarm clock is set to send the note alarm after the given number of milliseconds. The return value is the number of milliseconds left in the previous alarm clock. alarm can be used to write a command timeout which stops a process from running after a given amount of time.

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

int pid;
char *prog;

void
notehandler(void *, char *note)
{
	if(strcmp(note, "alarm") == 0)
		if(postnote(PNGROUP, pid, "kill") < 0)
			sysfatal("could not time out %s: %r\n", prog);
		else{
			fprint(2, "timeout\n");
			exits("timeout");
		}
	else
		noted(NDFLT);
}

int
endswith(char *full, char *what)
{
	int i;
	char *wp = what + strlen(what) - 1;

	for(i = strlen(full) - 1; wp >= what; i--, wp--)
		if(full[i] != *wp)
			return 0;
	return 1;
}

void
main(int argc, char *argv[])
{
	long ms;
	Waitmsg *w;

	if(argc <= 2){
		fprint(2, "usage: %s seconds command-line\n", argv[0]);
		exits("usage");
	}
	ms = strtoul(argv[1], nil, 10) * 1000; /* sec -> ms */
	prog = smprint("/bin/%s", argv[2]);
	switch(pid = rfork(RFPROC | RFFDG | RFENVG | RFREND | RFMEM | RFNOTEG)){
	case -1:
		sysfatal("fork failed: %r");
	case 0:
		exec(prog, &argv[2]);
		prog = smprint("./%s", argv[2]);
		exec(prog, &argv[2]);
		sysfatal("exec failed: %r");
	}
```

```
        notify(notehandler);
        alarm(ms);
        w = wait();
        if(w->msg[0] != '\0'){
            fprint(2, "%s failed with %s\n", prog, w->msg);
            free(prog);
            exits("failed run");
        }
        free(prog);
        exits(0);
    }
```

We have to provide endswith since the C library doesn't provide the similar strrstr (it does provide strstr and other functions). smprint creates, using malloc, a string which contains the fully formatted text. Use this instead of a custom buffer and sprint, as it avoids the risk of truncating or overflow due to an improperly sized buffer. The RFMEM flag is set so the process can change prog at will. We kill with PNGROUP in case the program that you run forks its own processes.

This example shows another feature of the Plan 9 C compilers: an unnamed argument signals that it is not used.