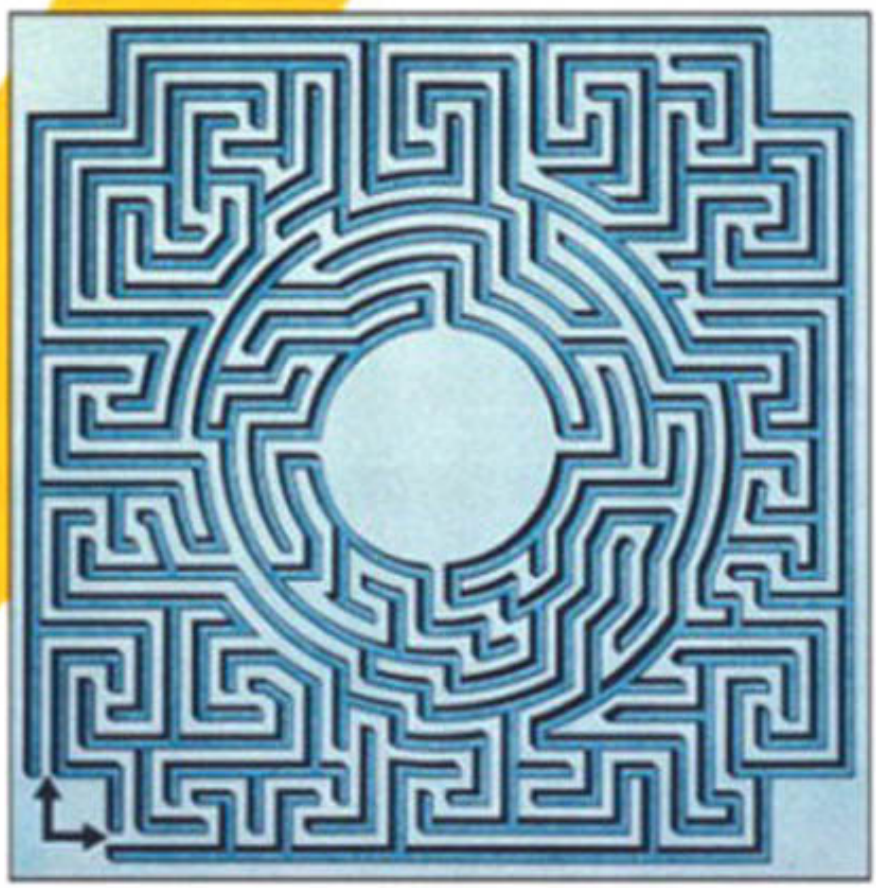
▶Updated Classic!

Advanced UNIX® Programming

SECOND EDITION



MARC J. ROCHKIND

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Advanced UNIX Programming

The return value is used to indicate an error, so an EOF is indicated with the third argument, as in this calling example:

```
ec_false( getln(s, sizeof(s), &iseof) )
if (iseof)
    printf("EOF\n");
else
    printf("Read: %s\n", s);
```

`getln` is efficient for terminals because it reads the entire line with a single system call. Furthermore, it doesn't have to search for the end—it just goes by the count returned by `read`. But it doesn't work on files or pipes at all because, since the one-line limit doesn't apply, `read` will in general read too much. Instead of `getln` reading a line, it will read the next `max - 1` characters (assuming that many characters are present).

A more universal version of `getln` would ignore that unique property of terminals—reading one line at most. It would simply examine each character, looking for a newline:

```
bool getln2(char *s, ssize_t max, bool *iseof)
{
    ssize_t n;
    char c;

    n = 0;
    while (true)
        switch (read(STDIN_FILENO, &c, 1)) {
        case -1:
            EC_FAIL
        case 0:
            s[n] = '\0';
            *iseof = true;
            return true;
        default:
            if (c == '\n') {
                s[n] = '\0';
                *iseof = false;
                return true;
            }
            if (n >= max - 1) {
                errno = E2BIG;
                EC_FAIL
            }
            s[n++] = c;
        }
```

```
EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

This version treats a Ctrl-d typed anywhere as indicating an EOF, which is different from what `getln` did (treating only a Ctrl-d at the beginning of a line as an EOF). With anything other than a terminal, remember, there is no Ctrl-d; an end-of-file is simply the end of the file or a pipe with no open writing file descriptor.

Although `getln2` reads terminals, files, and pipes properly, it reads those sources more slowly than it might because it doesn't buffer the input as described in Section 2.12. This is easily fixed by changing `getln2` to call `Bgetc`, which is part of the `BUFIO` package introduced in that section. `Bopen` is already suitable for opening terminal special files (e.g., /dev/tty). However, to allow us to use `Bgetc` on the standard input, we need to add a function called `Bfdopen` (Exercise 4.2) that initializes a `BUFIO` pointer from an already-open file descriptor instead of from a path. Then we could read a character from the standard input, whether it's a terminal, file, or pipe, like this:

```
ec_null( stin = Bfdopen(STDIN_FILENO, "r") )
while ((c = Bgetc(stin)) != -1)
    /* process character */
```

We're now reading as fast as we can in each case: a block at a time on files and pipes, and a line at a time on terminals.

Our implementation of the `BUFIO` package doesn't allow the same `BUFIO` pointer to be used for both input and output, so if output is to be sent to the terminal, a second `BUFIO` must be opened using file descriptor `STDOUT_FILENO` (defined as 1).

The UNIX standard I/O Library provides three predefined, already-opened `FILE` pointers to access the terminal: `stdin`, `stdout`, and `stderr`, so its function `fdopen`, which is like our `Bfdopen`, usually need not be called for a terminal.

Output to a terminal is more straightforward than input, since nothing like erase and kill processing is done. As many characters as we output with `write` are immediately queued up for sending to the terminal, whether a newline is present or not.

`close` on a file descriptor open to a terminal doesn't do any more than it does for a file. It just makes the file descriptor available for reuse; however, since the file

descriptor is most often 0, 1, or 2, no obvious reuse comes readily to mind. So no one bothers to close these file descriptors at the end of a program.[1]

4.2.2 Nonblocking Input

As I said, if a line of data isn't available when `read` is issued on a terminal, `read` waits for the data before returning. Since the process can do nothing in the meantime, this is called *blocking.* No analogous situation occurs with files: either the data is available or the end-of-file has been reached. The file may be added to later by another process, but what matters is where the end is at the time the `read` is executed.

The `O_NONBLOCK` flag, set with `open` or `fcntl`, makes `read` nonblocking. If a line of data isn't available, `read` returns immediately with a –1 return and `errno` set to `EAGAIN`.[2]

Frequently we want to turn blocking on and off at will, so we'll code a function `setblock` to call `fcntl` appropriately. (The technique I'll use is identical to what I showed in Section 3.8.3 for setting the `O_APPEND` flag.)

```
bool setblock(int fd, bool block)
{
    int flags;

    ec_neg1( flags = fcntl(fd, F_GETFL) )
    if (block)
        flags &= ~O_NONBLOCK;
    else
        flags |= O_NONBLOCK;
    ec_neg1( fcntl(fd, F_SETFL, flags) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Here's a test program for `setblock`. It turns off blocking and then reads lines in a loop. If there's nothing there, it sleeps for 5 seconds before continuing. I include

1. We will be closing them in Chapter 6 when we connect two processes with a pipe.
2. In some versions of UNIX, there is a similar flag called `O_NDELAY`. If set and no data is available, `read` returns with a 0, which is indistinguishable from an end-of-file return. Better to use `O_NONBLOCK`.

in the prompt the time since the loop started, using the `time` system call that I introduced in Section 1.7.1.

```
static void test_setblock(void)
{
    char s[100];
    ssize_t n;
    time_t tstart, tnow;

    ec_neg1( tstart = time(NULL) )
    ec_false( setblock(STDIN_FILENO, false) )
    while (true) {
        ec_neg1( tnow = time(NULL) )
        printf("Waiting for input (%.0f sec.) ...\n",
          difftime(tnow, tstart));
        switch(n = read(STDIN_FILENO, s, sizeof(s) - 1)) {
        case 0:
            printf("EOF\n");
            break;
        case -1:
            if (errno == EAGAIN) {
                sleep(5);
                continue;
            }
            EC_FAIL
        default:
            if (s[n - 1] == '\n')
                n--;
            s[n] = '\0';
            printf("Read \"%s\"\n", s);
            continue;
        }
        break;
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("test_setblock")
EC_CLEANUP_END
}
```

Here's the output from one run. I waited a while before typing "hello," and then waited awhile longer before typing Ctrl-d:

```
Waiting for input (0 sec.) ...
Waiting for input (5 sec.) ...
Waiting for input (10 sec.) ...
hello
Waiting for input (15 sec.) ...
```

```
Read "hello"
Waiting for input (15 sec.) ...
Waiting for input (20 sec.) ...
Waiting for input (25 sec.) ...
^DEOF
```

The approach of sleeping for 5 seconds is a compromise between issuing `read`s so frequently that it wastes CPU time and waiting so long that we don't process the user's input right away. As it is, you can see that several seconds elapsed between when I typed "hello" and when the program finally read the input and echoed it back. Thus, generally speaking, turning off blocking and getting input in a `read`/`sleep` loop is a lousy idea. We can do much better than that, as I'll show in the next section.

As I mentioned, you can also set the `O_NONBLOCK` flag when you open a terminal special file with `open`. In this case `O_NONBLOCK` affects `open` as well as `read`: If there is no connection, `open` returns without waiting for one.

One application for nonblocking input is to monitor several terminals. The terminals might be laboratory instruments that are attached to a UNIX computer through terminal ports. Characters are sent in sporadically, and we want to accumulate them as they arrive, in whatever order they show up. Since there's no way to predict when a given terminal might transmit a character, we can't use blocking I/O, for we might wait for one terminal that has nothing to say while other talkative terminals are being ignored. With nonblocking I/O, however, we can poll each terminal in turn; if a terminal is not ready, `read` will return –1 (`errno` set to `EAGAIN`) and we can just go on. If we make a complete loop without finding any data ready, we sleep for a second before looping again so as not to hog the CPU.

This algorithm is illustrated by the function `readany`. Its first two arguments are an array `fds` of file descriptors and a count `nfds` of the file descriptors in the array. It doesn't return until a `read` of one of those file descriptors returns a character. Then it returns via the third argument (`whichp`) the subscript in `fds` of the file descriptor from which the character was read. The character itself is the value of the function; 0 means end-of-file; –1 means error. The caller of `readany` is presumably accumulating the incoming data in some useful way for later processing.[3]

3. Assume the laboratory instruments are inputting newline-terminated lines. In Section 4.5.9 we'll see how to read data without waiting for a complete line to be ready.

```
int readany(int fds[], int nfds, int *whichp)
{
    int i;
    unsigned char c;

    for (i = 0; i < nfds; i++)
        setblock(fds[i], false); /* inefficient to do this every time */
    i = 0;
    while (true) {
        if (i >= nfds) {
            sleep(1);
            i = 0;
        }
        c = 0; /* return value for EOF */
        if (read(fds[i], &c, 1) == -1) {
            if (errno == EAGAIN) {
                i++;
                continue;
            }
            EC_FAIL
        }
        *whichp = i;
        return c;
    }

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

The comment about the call to `setblock` being inefficient means that we really don't have to do it every time `readany` is called. It would be more efficient, if less modular, to make it the caller's responsibility.

Here's a test function for `readany`. It opens two terminals: /dev/tty is the controlling terminal (a `telnet` application running elsewhere on the network), as we explained in Section 4.2.1, and /dev/pts/3 is an `xterm` window on the screen attached directly to the computer, which is running SuSE Linux. (I discovered the name /dev/pts/3 with the `tty` command.)

```
static void readany_test(void)
{
    int fds[2] = {-1, -1}, which;
    int c;
    bool ok = false;

    ec_neg1( fds[0] = open("/dev/tty", O_RDWR) )
    ec_neg1( fds[1] = open("/dev/pts/3", O_RDWR) )
```