How does stdin buffer mode works?

Asked 9 months ago Modified 9 months ago Viewed 285 times

1

I know how **stdout** buffer mode works,that is:

unbuffered: no buffer.

line buffered: flush the buffer when encounter a newline character.

block buffered: flush when the buffer is full.

I can understand the items above.

but I am confused about the **stdin** buffer works,I cannot imagine what will happen when use one of the three buffer mode.

And,for the three buffer mode,when does the stdin use which mode?

Thanks in advance.

```
// test02.c
int main(int argc, char *argv[]){
  char ptr[20];
  scanf("%s",ptr);
  printf("%s\n",ptr);
  return 0;
}
//./test02 < log

// in this case,scanf() just read the first line of log file
// can anyone explain how the stdin buffer works?
```

c linux systems-programming

Share Edit Follow Flag

edited Dec 1, 2023 at 9:51

asked Dec 1, 2023 at 9:45

hhx
31 3

How would unbuffered input work? You'd need to cyclically check if some keyboard key is pressed. That's what e.g. video games typically do or even simple text editors where characters appear immediately. – Aconcagua Dec 1, 2023 at 10:10

I'm not sure in how far it is meaningful to distinguish between the other two on input. There's no way to automatically flush the buffer, input streams rely on the application to read from (and this doesn't need to occur line by line). Some characteristics might justify considering line buffering (input gets available on pressing enter), but the characters entered before need to be buffered somewhere as well (if I had to implement I'd do in the same buffer and maintain a pointer to the last new-line for not delivering data not yet confirmed by pressing `enter`). – Aconcagua Dec 1, 2023 at 10:14

Yet another interesting aspect: As we *cannot* flush automatically, what happens on the buffer being full? Not accepting any further input (instead of e.g. overwriting and thus losing oldest input) would appear most appropriate to me in *most* cases (but in some applications newest data might be more important – just in general, beyond console input...). – Aconcagua Dec 1, 2023 at 10:18

This question is not really about stdin at all. Consider what happens when stdin is a regular file. I suspect the underlying question is really about reading from non-regular files (eg pipes). – William Pursell Dec 1, 2023 at 11:20

@WilliamPursell,stdin may be regular files(for redirection),terminal,pipes and so on. I just want to clear that how stdin buffer works under these circumstances. – hhx Dec 1, 2023 at 12:12

1 Answer

Sorted by: Highest score (default)

3

It is up to the implementation - how exactly is FILE I/O implemented - what do these modes do, also for stdout. It is nowhere *guaranteed* that setting line buffered will actually be line buffered.

When using glibc on linux when using normal files when underflowing read data to read `_IO_new_file_underflow` is called here https://github.com/lattera/glibc/blob/895ef79e04a953cac1493863bcae29ad85657ee1/libio/fileops.c#L524 `_IO_SYSREAD (fp, fp->_IO_buf_base, fp->_IO_buf_end - fp->_IO_buf_base);`. It does single `read` syscall. You could follow setbuf to how it sets the buffer.

However, practical answer is much more fun. Consider the following C program, that depending on the first argument, which is a number 1, 2 or 3, uses full-, line- or no buffering for stdin stream.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
        assert(argc >= 1);
        const int arg = atoi(argv[1]);
        const int mode = arg == 1 ? _IOFBF : arg == 2 ? _IOLBF : _IONBF;
        printf("Using mode=%d\n", mode);
        char buf[10];
        setvbuf(stdin, buf, mode, sizeof(buf));
        char ptr[20];
        scanf("%19s", ptr);
}
```

The following Makefile compiles the program and runs the program under strace with two lines of input using bash here string. It is so that we can see what `read` system calls are being

made:

```
SHELL = bash
all:
        gcc main.c
        strace -e read ./a.out 1 <<<$$'abc\ndef'
        strace -e read ./a.out 2 <<<$$'abc\ndef'
        strace -e read ./a.out 3 <<<$$'abc\ndef'
```

Make results in:

```
$ make
gcc main.c

strace -e read ./a.out 1 <<<$'abc\ndef'
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220~\2\0\0\0\0\0"...,
832) = 832
Using mode=0
read(0, "abc\ndef\n", 10)               = 8
+++ exited with 0 +++

strace -e read ./a.out 2 <<<$'abc\ndef'
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220~\2\0\0\0\0\0"...,
832) = 832
Using mode=1
read(0, "abc\ndef\n", 10)               = 8
+++ exited with 0 +++

strace -e read ./a.out 3 <<<$'abc\ndef'
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220~\2\0\0\0\0\0"...,
832) = 832
Using mode=2
read(0, "a", 1)                         = 1
read(0, "b", 1)                         = 1
read(0, "c", 1)                         = 1
read(0, "\n", 1)                        = 1
+++ exited with 0 +++
```

Setting `_IOFBF` and `_IOLBF` just does a single `read()` with the size of the buffer `10`. These two work the same when reading. However, `_IONBF` does a single small `read(0, .., 1)` up until a newline.

Share Edit Follow Flag

answered Dec 1, 2023 at 10:04

KamilCuk

137k 8 69 132

It would be interesting to add `strace -e read ./a.out $i < main.c` to the makefile recipe and see the behavior when reading from a regular file. Perhaps more interesting to use a file that is bigger than 10k) – William Pursell Dec 1, 2023 at 11:22