Buffering... Buffering... Buffering...

Jeremy Mates

June 27, 2021

To recap `setbuf(3)`, there are three forms of stream buffering available in the standard library: block, line, and unbuffered. Standard error is unbuffered. Standard out is line buffered if the output is to a terminal, or block buffered otherwise. Buffering is most apparent when the expected output does not appear, usually when one has switched from writing to the terminal to a file or pipe. This changes the buffering from line-based to block.

```
$ perl -E 'say "hi"; sleep 7'
hi
^C
$ perl -E 'say "hi"; sleep 7' > out &
[1] 56029
$ cat out
$ cat out
$ sleep 10 && cat out
hi
[1] + Done                    perl -E say "hi"; sleep
```

Where was the output? Stuck in a buffer flushed only when the program exited. Some programs have flags to line or unbuffer their output (`tcpdump -l`); programming language support for what `setbuf(3)` and `fflush(3)` provide varies from complete (TCL) to utterly lacking (the shell). Other options include `unbuffer(1)` or `stdbuf(1)`. Buffered output risks being lost should the buffer not be flushed–so why not unbuffer everything? Because unbuffered output can be inefficient.

Causes of lost output vary from `kill -9`

```
$ python -c 'import time;print "ugh";time.sleep(99)' >out &
[1] 63787
$ kill -9 $!
$ cat out
[1] + Killed                  python -c import time;pr
$ wc -c out
       0 out
```

to more subtle issues that may require `flush` or `close` calls to avoid losing output:

```
$ ruby -e 'puts "hi"; exec "echo hi"'
hi
hi
$ ruby -e 'puts "hi"; exec "echo hi"' | grep hi
hi
$ ruby -e 'puts "hi"; STDOUT.flush; exec "echo hi"' | grep hi
hi
hi
```

The first command uses line-based buffering because output is to the terminal; data is lost in the second case as the block-based buffer piping to `grep` is not flushed prior to the `exec`. The third command avoids this loss. Buffering issues may also occur across `fork(2)` where both the existing parent and new child share the same data.

```
/* buffork - buffer across a fork (bad) */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void) {
    pid_t pid;
    printf("buffered in %d\n", getpid());
    pid = fork();
    if (pid == -1) abort();
    printf("after in %d\n", getpid());
    exit(0);
}
```

The code appears to be correct when the output is to the terminal, yet duplicates data under block-based buffering:

```
$ make buffork
cc     buffork.c   -o buffork
$ ./buffork
buffered in 65741
after in 65741
after in 65742
$ ./buffork > out ; cat out
buffered in 65744
after in 65744
buffered in 65744
after in 65745
```

As with the previous case, a `fflush(3)` prior to the `fork` will avoid this issue. Another design would be to have the parent emit nothing to standard output, or to only emit data after the child processes have all been forked. Or to unbuffer standard output.

Mixed Buffer Calls

A single process can create the same error should buffered and unbuffered system calls be mixed,

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
    puts("duck");
    puts("duck");
    write(1, "goose\n", 6);
    return 0;
}
```

which when compiled and run shows

```
$ make mixbuf
cc     mixbuf.c   -o mixbuf
$ ./mixbuf
duck
duck
goose
$ ./mixbuf > out ; cat out
goose
duck
duck
```

The `puts(3)` call buffers while `write(2)` does not. This can be avoided by not mixing unbuffered and buffered calls (hint: `FILE *` system calls buffer; `int fd` ones do not).

Standard Input

Buffering may also come into play on standard input. Consider the following contrived program which intends to read a total of eight characters, four prior to an `exec` of itself and four more afterwards.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
char buf[5];
int main(int argc, char *argv[]) {
    fgets(buf, 5, stdin);
    printf("%s >%s<\n", *argv, buf);

    /* avoid infinite loop of execs... */
    if (strncmp(*argv, "./replace", 10) == 0) return 0;

    /* replace self under a new name */
    execl("./readfour", "./replace", (char *) 0);
    abort();
}
```

This results in no input to the replacement process.

```
$ make readfour
cc -O2 -pipe    -o readfour readfour.c
$ printf asdfasdf | ./readfour
./readfour >asdf<
./replace ><
```

Buffering is to blame; if run under `strace` the trace can be searched for the string `asdf` (the "known string search pattern" pattern).

```
$ printf asdfasdf | strace -f -o rrr ./readfour
...
$ grep asdf rrr
22059 read(0, "asdfasdf", 4096)           = 8
22059 write(1, "./readfour 'asdf'\n", 18) = 18
```

Are there any other `read(2)` calls made?

```
$ fgrep 'read(0,' rrr
22059 read(0, "asdfasdf", 4096)         = 8
22059 read(0, "", 4096)                 = 0
$
```

the first `read` captures the entire eight bytes of input. It actually asked for 4096, but had to settle for less. This leaves nothing for the second `read` in the replacement code. With sufficient padding to fill the buffer of the first `read`, the second `read` will obtain input:

```
$ perl -e 'print "a"x4096; print "bbbb"' | ./readfour
./readfour 'aaaa'
replace 'bbbb'
```

One workaround is to use an unbuffered `read` to only read four bytes. Unbuffered and buffered reads should probably not be mixed on the same stream.

```
    read(STDIN_FILENO, buf, 4);
```

Input may not be of fixed length. In that case the initial code will need to do single character `read` calls until it reaches some record boundary (say, a newline), or it could instead use `fgets` and then call `fseek(3)` to reposition the file pointer back to that record boundary (assuming the file handle is seekable). Let's take a look at how shells use both of these methods to read input.

Seek

With an `input` file containing shell commands

```
$ cat input
echo a
echo bb
echo ccc
```

`strace` will reveal differences in how shells read input. First, the traditional `sh` as provided by `dash` or busybox systems (such as on Alpine Linux but not `bash` pretending to be `sh`):

```
$ strace -o outsh -e trace=desc sh < input >/dev/null
$ egrep 'read|seek' outsh
read(0, "echo a\necho bb\necho ccc\n", 1023) = 24
read(0, "", 1023)                       = 0
```

So, a complete `read` of the `input`, then a second `read` that triggers the `EOF` return value of `0`. Here is `ksh` reading the same file (note that `ksh` may instead be available as `mksh` or `pdksh`).

```
$ strace -o outksh -e trace=desc ksh < input >/dev/null
$ egrep 'read|seek' outksh
read(0, "echo a\necho bb\necho ccc\n", 512) = 24
lseek(0, -17, SEEK_CUR)                 = 7
read(0, "echo bb\necho ccc\n", 512)     = 17
lseek(0, -9, SEEK_CUR)                  = 15
read(0, "echo ccc\n", 512)              = 9
read(0, "", 512)                        = 0
```

Quite different; a 512-byte `read` is attempted, then `ksh` seeks backwards by 17 or then by 9 characters; these place the file pointer at the beginning of the 2nd and 3rd lines, respectively. `ksh` then re-reads from the start of each new line. If the input is unseekable `ksh` will instead read by single characters:

```
$ ruby -e 'puts "echo a\necho bb"' | strace -o qq ksh
a
bb
$ egrep 'read|seek' qq
read(0, "e", 1)                         = 1
read(0, "c", 1)                         = 1
read(0, "h", 1)                         = 1
...
```

By the way, the `strace ... < input` form works because `strace` ignores standard input and passes it down to the traced program. `strace` is thus a (mostly) transparent wrapper around the subsequent command. This is not the case for all commands. `ssh` in particular will consume standard input by default, which is why loops such as

```
while read host; do ssh -- "$host" "hostname -f"; done
```

fail,[1] as the first `ssh` call will consume the remainder of the hosts listed on standard input. There are various workarounds.

```
# close stdin
ssh -- "$host" ... <&-

# or instruct ssh not to read from stdin
ssh -n -- "$host" ...
```

Another issue with `seek` is that file descriptors in a child process may point to the same underlying object used by the parent. This means an `lseek(2)` call in a child can affect subsequent IO operations by the parent. It would be best to avoid such designs (outside of trivia competitions) and to either have the parent or the child assume sole responsibility for IO that involves seek calls. However,

[1] Please do not run `hostname -f` as `root` on Solaris systems. Thank you.

Advanced Seek

it may be instructive to show a pair of processes that interact with the same file descriptor to prove that both use the same underlying object. This adds a wrinkle in that the two processes will need to synchronize their actions; by default, either the parent or child could begin execution first following a `fork` depending on the whims of the process scheduler.

```
#include <err.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
char buf[8];
int main(int argc, char *argv[]) {
    int ch, fd[2], rv;

    signal(SIGPIPE, SIG_IGN);
    pipe(fd);

    switch (fork()) {
    case -1: err(1, "could not fork");
    case 0:                             /* child */
        rv = lseek(STDIN_FILENO, -7, SEEK_END);
        if (rv == -1) err(1, "could not seek");
        sleep(3);
        close(fd[1]);
        break;
    default:                            /* parent */
        close(fd[1]);
        read(fd[0], &ch, 1);
        read(STDIN_FILENO, buf, 7);
        printf("read '%s'\n", buf);
    }
    return 0;
}
```

This code uses `pipe(2)` with a blocking read in the parent until the child closes its side. At this point the parent should read from the input file from where the child moved the file pointer to with `lseek(2)`.

```
$ make advancedseek
cc     advancedseek.c   -o advancedseek
$ printf 'more than seven characters' > input
$ ./advancedseek < input
read 'racters'
```

If the child and parent file descriptors were independent the parent would instead have read from the beginning of the file. This while perhaps interesting has taken us slightly adrift from buffers.

Language Support for Buffering

Above I mentioned that language support for `setbuf(3)` and `fflush(3)` ranges from complete to not at all, for example

- Perl offers an `$fh->autoflush` toggle to buffer or not; there are also `flush` and `sync` methods that operate at the perlio and lower OS levels; see `IO::Handle` for details.
- Ruby copies Perl with options to flush, flush by default, and `sync` or `fsync` methods for either the Ruby buffers or lower OS levels.
- TCL - `chan(n)` lists `chan $fh configure -buffering ...` as supporting `none`, `line`, or `full`, and also offers `chan flush $fh` for an immediate flush.

Notably absent is buffering support in the shell; this leads to various programs perhaps having a flag to influence buffering–as mentioned before, `tcpdump -l`. Another option, maybe, is thus `stdbuf(1)` which in a shell pipeline may need to be applied to each and every program in that pipeline.

Pitfalls of `stdbuf(1)`

`stdbuf(1)` advertises the ability to "run a command, with modified buffering operations for its standard streams." Sounds great! Language X or program Y lack options to adjust the buffering, and `stdbuf` claims that it can. So what's not to like? How is a parent process even able to influence how a subsequent child process handles buffering? Let's peek at the `stdbuf` source code...

```
...
static void
set_LD_PRELOAD (void)
{
...
```

Support for this method may well depend on your feelings on monkey patching programs with new code that introduces new behavior. And there can be problems regardless; processes that maintain their own buffer will not be influenced by this, or `LD_PRELOAD` hopefully will not be allowed anywhere `sudo` is involved, nor hopefully will the changed behavior run afoul any security policies...

How to Shoot Performance In the Foot

Inefficient code is fortunately for this section quite easy to write; we need only remove buffering and operate byte by byte for the task of printing 9,999 `a` to standard output. Here is `maxcalls.asm` that does that.

```
bits 64
section .data
letter: db "a"
section .text
global  _start
_start: mov r12,9999
        mov rax,1       ; sys_write
        mov rdi,1       ; stdout
        mov rsi,letter
        mov rdx,1       ; length
_again: syscall
        dec r12
        jnz _again      ; GOTO!
        mov rax,60      ; sys_exit
        mov rdi,0       ; exit code
        syscall
```

Linux does not change `R12` nor the other registers used during the `syscall`–see the "System V Application Binary Interface" for details–so we can get away with a fairly tight loop that decrements `R12` to zero. By contrast, the more efficient `mincalls.asm` places all 9999 `a` into the program itself and prints them with a single `sys_write` call.

```
bits 64
section .data
lotsa:  times 9999 db "a"
.len:   equ $-lotsa
section .text
global  _start
_start: mov rax,1       ; sys_write
        mov rdi,1       ; stdout
        mov rsi,lotsa
        mov rdx,lotsa.len
        syscall
```

```
        mov rax,60      ; sys_exit
        mov rdi,0       ; exit code
        syscall
```

How do the benchmarks compare between these two rather extreme cases? Also–before wasting time on benchmarks, do both implementations produce the same and correct output? Comparison of broken or different things will not produce meaningful results.

```
$ nasm -f elf64 -o mincalls.o mincalls.asm
$ ld -o mincalls mincalls.o
$ nasm -f elf64 -o maxcalls.o maxcalls.asm
$ ld -o maxcalls maxcalls.o
$ ./mincalls > a
$ ./maxcalls > b
$ cmp a b; echo $?
0
$ wc -c b
9999 b
$ time for i in `jot 1000`; do ./mincalls > a; done

real    0m8.818s
user    0m0.107s
sys     0m0.841s
$ time for i in `jot 1000`; do ./maxcalls > b; done

real    0m18.898s
user    0m0.457s
sys     0m12.030s
```

A notable and obvious difference is the extremely high system time reported for `maxcalls`. This is because each byte in `maxcalls` triggers a system call–that is, the kernel takes over, performs IO on the byte, and then returns execution back to `maxcalls`. Again and again and again. `mincalls` is faster as it only makes two system calls. `strace -c` will report these numbers.

```
$ strace -c ./mincalls > a
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
 96.30    0.000052          52         1           write
  3.70    0.000002           2         1           execve
------ ----------- ----------- --------- --------- ----------
100.00    0.000054                     2           total
$ strace -c ./maxcalls > b
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
 99.96    0.045516           5      9999           write
  0.04    0.000016          16         1           execve
------ ----------- ----------- --------- --------- ----------
100.00    0.045532                 10000           total
```

`mincalls` spends 5e-5 seconds on `write` while `maxcalls` uses 4e-2 or three orders of magnitude more. Another interesting number is the context switches; `vmstat(8)` on Linux will show this and other useful metrics. Assuming the benchmark takes longer than just a few seconds, in a second terminal window run

```
$ vmstat -w 1
```

and be sure to make the window wide enough to hold the results. Then re-run the benchmarks and observe the `cs`, `us`, `sy`, and `wa` columns. Context switches and IO wait times are higher when `mincalls` is being run–this is because `mincalls` completes its work very quickly so there is more system activity from the shell starting new `mincalls` instances. `maxcalls` shows lower context switches and wait times as fewer instances of it are running per unit time–`maxcalls` takes longer to complete its work. This is why benchmarks need careful consideration of what is being measured and how to collect it–and how long to collect the data for;

a benchmark of a complicated system may require days for the system to settle in, effects of random cron jobs to be seen, etc.

An application may need to Goldilocks how much to buffer (too cold...) versus printing often (too hot!) and may need to balance efficiency versus the risk of information loss should the application or system fail. Latency versus throughput. Unbuffered writes are typically more expensive though this may not matter if the logs are important, if the logging is of negligible overhead, or the logs are large enough such that they do not much benefit from buffering. Specific advice would be to first get the code working, second to profile it and see where it is slow, and third to write benchmarks to test the replacement code against the first draft. I will now ignore this advice.

Buffer Benchmarks

A relevant question is how inefficient unbuffered logging is. Unlike the previous assembly code that used extremes, real application logs run along the lines of

```
fprintf(stderr, "%s: fail connect '%s': ...
```

or instead an entire JSON structure or stack trace.[2] Actual logs should be used, or a reasonable replication thereof. Or borrow logs from production–but note that some regulations frown on production logs being moved to test environments.

I will also ignore this advice, and look specifically at logs of 64 or 4096 characters, buffered or unbuffered.

```
#!/usr/bin/env expect
set logfile  scratch
set buffhows [list none full]
set lengths  [list 64 4096]
set logbytes 2097152
set trials   500
proc writelogs {logfile buffhow logbytes length trials} {
    set statfh  [open stats.$length.$buffhow w]
    set logline [string repeat a [expr $length - 1]]
    while {$trials} {
        set  logfh [open $logfile w]
        chan configure $logfh -buffering $buffhow
        set iterations [expr int($logbytes / $length)]
        set timer [time {
            while {$iterations} {
                puts $logfh $logline
                incr iterations -1
            }
            if {$buffhow eq "full"} {
                chan flush $logfh
            }
        }]
        puts  $statfh [lindex [split $timer " "] 0]
        close $logfh
        incr  trials -1
    }
}
foreach buffhow $buffhows {
    foreach loglength $lengths {
        writelogs $logfile $buffhow $logbytes $loglength $trials
    }
}
```

[2] Java debug logs that fill the entire 80 gigabyte log partition two weeks after the production launch are not unknown in the industry.

This may take some time to run (it does emit `2097152*500*2` bytes) though perhaps less for those of you who actually use SSD drives.

```
$ expect benchlogs.tcl
$ ls stats.*
stats.4096.full stats.4096.none stats.64.full stats.64.none
$ R -q --silent --no-save
> smallfull=scan("stats.64.full")
Read 500 items
> smallnone=scan("stats.64.none")
Read 500 items
> bigfull=scan("stats.4096.full")
Read 500 items
> bignone=scan("stats.4096.none")
Read 500 items
```

The TCL `time(n)` command times the execution of the writes; lower numbers are better. With 64 bytes the fully buffered output takes less time and has a lower standard deviation than the unbuffered writes.

```
> summary(smallfull)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  19209   19376   19494   19874   19867   34234
> summary(smallnone)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  78816   79949   80398   80784   80896  118876
> sd(smallfull); sd(smallnone)
[1] 1121.129
[1] 3102.511
```

The 4096 byte unbuffered writes by contrast are only somewhat less efficient than the buffered writes:

```
> summary(bigfull)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6360    6961    7464    7607    8158   11558
> summary(bignone)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6684    6994    7738    7871    8184   12256
> sd(bigfull); sd(bignone)
[1] 887.7779
[1] 1020.918
```

Buffers and Multiple Clients

Socket services suffer no less from buffering than terminal programs do. One example is that of multiple writers whose input is collected by a single listener. In one terminal, start a listener process. `socat` is used as `nc` may variously be called `netcat` or `ncat` and may not support unix domain sockets.

```
$ socat -u UNIX-LISTEN:thesocket,fork ./out
```

In another terminal in the same directory as the above, run this `lobber` script

```
#!/bin/sh
sendaline () {
   echo "pid $$ says $@" | socat - UNIX-CONNECT:thesocket
}
for i in 1 2 3 4 5 6 7 8 9 0; do
   sendaline "$i" &
done
wait
```

which should send 10 lines to the listener, and from there into the `out` file.

```
$ sh lobber
$ cat out
pid 77487 says 5
pid 77487 says 2
pid 77487 says 3
pid 77487 says 1
pid 77487 says 0
pid 77487 says 7
pid 77487 says 4
pid 77487 says 9
pid 77487 says 6
pid 77487 says 8
```

The input is out of order (at the whim of the job scheduler) but all the lines are complete. So where is the buffering problem? We need to send more to the listener. A lot more.

```
#!/usr/bin/env expect
for {set i 0} {$i < 25} {incr i} {
    set pid [fork]
    if {$pid == -1} { puts stderr "fork failed"; exit 1 }
    if {$pid != 0} { continue }
    set fh [open "| socat - UNIX-CONNECT:thesocket" w]
    chan configure $fh -buffering none
    set char    [format %c [expr 97 + $i]]
    set teststr [string repeat $char 9999]
    for {set rep 100} {$rep > 0} {incr rep -1} {
        puts -nonewline $fh $teststr
    }
    exit 0
}
wait -i -1
```

This version forks 25 times and generates strings 9,999 characters long, each child using a unique ASCII letter. The strings are sent 100 times each, no buffering. Be sure to empty the old `out` file.

```
$ :> out
$ expect lobber.tcl
$ wc -c out
 24997500 out
$ echo $((9999*100*25))
24997500
```

That is the expected number of characters. `out` is however rather awkward to parse or display. The expected output is contiguous blocks of 9,999 unique characters in some unknown ordering. What does the file actually contain? Some C can tally the contiguous character counts.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int ch, prev = getchar();
    unsigned long count = 1;
    while ((ch = getchar()) != EOF) {
        if (ch != prev) {
            printf("%lu %c\n", count, prev);
            count = 0;
            prev  = ch;
        }
        count++;
    }
    printf("%lu %c\n", count, prev);
}
```

This reveals runs of 8,192 characters, not 9,999 (and `strace` will confirm this as the buffer size):

```
$ make charcounter
cc     charcounter.c   -o charcounter
$ ./charcounter < out | head -3
8192 a
8192 b
8192 a
$ ./charcounter < out | tail -3
476 n
476 u
476 o
```

The trailing 476 byte counts come from the remainder

```
$ echo $(( 9999 * 100 / 8192 ))
122
$ echo $(( 9999 * 100 - 8192 * 122 ))
476
```

Thus with sufficiently large input otherwise invisible buffers can be overrun and the output corrupted–a different design would be required here to collect the input from multiple sources into a single file.

The above shows the need for careful design and testing of code, especially given that different buffering methods are used depending on the context or system calls involved, or where the file descriptors are shared between processes and used by both, or . . .