Suffering from Buffering?

My log file has nothing in it!

My output is coming out in the wrong order!

When my program terminates abnormally, the output is incomplete!

My web server says I didn't send the right headers, but I'm sure I did!

I'm trying to send data over the network, but nothing is sent!

I'm afraid you're probably a victim of buffering.

What is Buffering?

I/O is performed by your operating system. When Perl wants to read data from the disk, or to write it to the network, or to read or write data anywhere, Perl has to make a request to the operating system and ask that the data be read or written. This is an example of `making a system call'. (Don't confuse this with Perl's `system` function, which is totally unrelated.)

Making a system call is a relatively slow operation. On top of that, if the data is coming from the disk, you might have to wait for the disk to spin to the right position (that's called the `latency time') and you might have to wait for the disk heads to move over to the right track (that's called the `seek time'), and as computer operations go, that wait is unbearably long---several milliseconds is typical. By contrast, a typical computer operation, such as assigning to a variable or adding two numbers, takes a fraction of a microsecond.

Suppose you're reading a ten-thousand line file line by line:

```
while (<FILE>) {
  print if /treasure/;
}
```

If Perl made a system call for every read operation, that would be 10,001 system calls in all (one extra to detect end-of-file), and if the file was on the disk, it would have to wait for the disk at least 10,000 times. That would be very slow.

For efficiency, Perl uses a trick called *buffering*. The first time you ask to read from the file, Perl has to make a system call. Since system calls are expensive, it plans ahead. If you tried to read a little bit of text, you will probably want to read the rest later. The blocks on your disk are probably about 8K bytes, and your

computer hardware is probably designed to transfer an entire block of data from the disk at once. So instead of asking the system for a little bit of text, Perl actually asks the system for an entire blockful, which hardly takes longer to get than a little bit would have. Then it stores this block of data in a region of memory that is called a *buffer*, and gives you back the one line you asked for. The next time you ask for a line, Perl already has the line you want in memory in the 8K buffer. It doesn't have to make another system call; it just gives you the next line out of the buffer. Eventually you read up to the end of the buffer, and then Perl makes another system call to get another bufferful of data.

If lines typically have about 60 characters each, then the 10,000-line file has about 610,000 characters in it. Reading the file line-by-line with buffering only requires 75 system calls and 75 waits for the disk, instead of 10,001. On my system, a simple program with buffered reading ran about 40% faster than the corresponding program that made a system call for every line.

For writing, Perl uses the same trick. When you write data to a file with `print`, the data doesn't normally go into the file right away. Instead, it goes into a buffer. When the buffer is full, Perl writes all the data in the buffer at once. This is called *flushing the buffer*. Here the performance gain is even bigger than for reading, about 60%. But the buffering can sometimes surprise you.

Surprise!

Suppose you have a program like this:

```
foreach $item (@items) {
  think_for_a_long_time($item);
  print LOG "Finished thinking about $item.\n";
}
```

Suppose there are 1,000 items, and the program will have to think about each one for two minutes. The program will take about 35 hours to complete, and you'd like to be able to peek at the log file to see how it is doing. You start up the program, wait ten minutes, and peek at the log file---but there's nothing there. Disaster! What happened? Is the program stuck? Is it taking five times as long to think about the items as you thought it would?

No, the program is not stuck, or even slow. The problem is that the `print`s to the log file are being buffered. The program has thought about the first five items, and it wrote the log messages for them, but the writes went into the buffer, and Perl isn't going to make a system call to send the buffer to the disk until the buffer is full. The buffer is probably 8K bytes, and the log messages are about 28 bytes each, so what's going to happen is that you won't see anything in the log file until about ten hours from now, when the first 292 messages will appear all at once. After that, you won't get any more messages for another ten hours.

That's a problem, because it's not what you wanted. Here you don't really care about the efficiency gain of buffered writes. On the plus side of buffering, you're saving about four seconds over the 35-hour lifetime of the program. On the minus side, it's making your logging feature useless. You think having the log is worth waiting an extra four seconds, so you'd like to turn off the buffering.

Disabling Inappropriate Buffering

In Perl, you can't turn the buffering off, but you can get the same benefits by making the filehandle *hot*. Whenever you print to a hot filehandle, Perl flushes the buffer immediately. In our log file example, it will flush the buffer every time you write another line to the log file, so the log file will always be up-to-date.

Here's how to make the filehandle hot:

```
{ my $ofh = select LOG;
  $| = 1;
  select $ofh;
}
```

The key item here is the `$|` variable. If you set it to a true value, it makes the current filehandle hot. What's the current filehandle? It's the one last selected with the `select` operator. So to make `LOG` hot, we `select' it, set `$|` to a true value, and then we re-`select' whatever filehandle was selected before we selected `LOG`.

Now that `LOG` is hot, Perl will flush the buffer every time we print to `LOG`, so messages will appear in the log file as soon as we print them.

Sometimes you might see code like this:

```
select((select(LOG), $|=1)[0]);
```

That's a compressed way of writing the code above that makes `LOG` hot.

If you happen to be using the `FileHandle` or `IO` modules, there's a nicer way to write this:

```
use FileHandle;          # Or `IO::Handle' or `IO::'-anything-else

...

LOG->autoflush(1);       # Make LOG hot.

...
```

Hot and Not Hot

If Perl is really buffering output, how is it you didn't notice it before? For example, run this program, `favorite`:

```
print "What is your favorite number?   ";
$num = <STDIN>;
$mine = $num + 1;
print "Well, my favorite is $mine, which is a much better number.\n";

% ./favorite
What is your favorite number?     119
Well, my favorite is 120, which is a much better number.
```

If you run this, you find that it works the way you expect: The prompt appears on the screen right away. Where's the buffering? Why didn't Perl save up the output until it had a full buffer? Because that's almost never what you want when you're writing to a terminal, the standard I/O library that Perl uses takes care of it for you. When a filehandle is attached to the terminal, as STDOUT is here, it is in *line buffered mode* by default. A filehandle in line buffered mode has two special properties: It's flushed automatically whenever you print a newline character to it, and it's flushed automatically whenever you read from the terminal. The second property is at work here: STDOUT is flushed automatically when we read from STDIN.

But now let's try it with STDOUT attached to a file instead of to the terminal:

```
% ./favorite > OUTPUT
```

Here the STDOUT filehandle has been attached to the file OUTPUT. The program has printed the prompt to the file, and is waiting for you to enter your favorite number. But if you open another window, and look into OUTPUT, you'll see that the prompt that favorite printed isn't in the file yet; it's still in the buffer. STDOUT is attached to a file, rather than to a terminal, so it isn't line-buffered; only filehandles attached to the terminal are line-buffered by default.

When the program finishes, it flushes all its buffers, so after you enter your favorite number, all the output, including the prompt, appears in the file at the same time.

There's one other exception to the rule that says that filehandles are cold unless they're attached to the terminal: The filehandle STDERR, which is normally used for error logging, is always in line buffered mode by default. If our original example had used STDERR instead of LOG we wouldn't have had the buffering problem.

Other Perils of Buffering

``My output is coming out in the wrong order!"

Here's a typical program that exhibits this common problem:

```
print "FILE LISTING OF DIRECTORY $dir:\n";
print "--------------------------------\n";
system("ls -l $dir");
print "--------------------------------\n";
```

Run this on a terminal, and it comes out OK. But if you run it with STDOUT redirected to a file, it doesn't work: the header appears after the file listing, instead of at the top.

So why didn't it work? Standard output is buffered, so the header lines are saved in the buffer and don't get to the file just yet. Then you run ls, which has its own buffer, and ls's buffer is flushed when ls exits, so ls's output goes into the file. Then you print the footer line and it goes into your programs buffer. Finally, your program finishes and flushes its buffer, and all three lines go into the output file, after the output from ls. To fix this, make STDOUT hot.

Now that we know why the data got into the file in the wrong order, that raises another question: If we have it print to the terminal, why does the output come out in the right order instead of the wrong order? Because when STDOUT is attached to the terminal, it is in line buffered mode, and is flushed automatically

whenever we print a newline character to it. The two header lines are flushed immediately, because they end with newlines; then comes the output of `ls`, and finally the footer line.

``My web server says I didn't send the right headers, but I'm sure I did!"

Here's a typical program that exhibits this common problem:

```
print "Content-type: text/html\n\n";

print "<title>What Time Is It?</title>\n";
print "<h1>The Current Time in Philadelphia is</h1>\n\n";
print "<pre>\n";

system("date");

print "</pre>\n\n";
```

You might think that the output is going to come out in the order you put it in the program, with the `Content-type` header, then the title, and with the date in between the `<pre>` tags. But it isn't. The `print` statements execute, but the output goes into your program's buffer. Then you run `date`, which generates some output, this time into the `date` command's buffer. When `date` exits (almost immediately), this buffer is flushed, and the server (whch is listening to your standard output) gets the date before it gets the output from your `print`s; your `print` data is still in the buffer. Later, when your program exits, its own buffer is flushed and the server gets the output from the `print`s.

The server was expecting to see that `Content-type` line first, but instead it got the date first. It can't proceed without knowing the content-type of the output, so it gives up and sends a message to the browser that says something like `500 Internal Server Error`.

Solution 1: Make `STDOUT` hot. Solution 2: Collect the output from `date` yourself and insert it into your buffer in the appropriate place:

```
...

$the_date = `date`;
print $the_date;

...
```

Here's a similar sort of problem that stems from the program aborting before you wanted it to:

```
print "Content-type: text/html\n\n";

print "<title>Division Table</title>\n";
print "<h1>Division Table</h1>\n\n";

for (i=0; $i<10; $i++) {
  for (j=0; $j<10; $j++) {
```

```
    print $i/$j, "\t";              # Ooops
  }
  print "\n";
}
```

The program is going to abort at the line that says `Ooops', because it divides by zero, and it's going to print the message

```
Illegal division by zero at division.cgi line 8.
```

What you actually see on your web browser depends a lot on the details of the web server, but here's one possible scenario: The server will collect all the output from your program, and send it back to the browser. You might think that the `Content-type` line will come first, followed by the title, and then the `division by zero` message, but you'd be wrong. The content-type and the title are printed to `STDOUT`, which is buffered, but the `divsion by zero` message is printed to `STDERR`, which isn't buffered. Result: The content-type and title are buffered. Then the error message comes out, and then, when the program exits, the `STDOUT` buffer is flushed and the content-type and title come out at last. The server was expecting to see the `Content-type` line right away, gets confused because it appears to be missing, and reports an error.

As usual, you can fix this by making `STDOUT` hot. Another way: Redirect error messages to a separate file, like this:

```
open STDERR, "> /tmp/division.err";
```

A third way:

```
use CGI::Carp 'fatalsToBrowser';
```

The `CGI::Carp` module will arrange that fatal error messages are delivered to the browser with a simple prefabricated HTTP header, so that the browser displays the error message properly.

Making both handles cold won't work, because when the program finishes, there's no telling which of the two will be flushed first.

``I'm trying to send data over the network, but nothing is sent!"

This one is the plague of novice network applications programmers; it bites almost everyone the first time they write a network application.

For concreteness, suppose you're writing a mail client, which will open a connection to the mail server and try to send a mail message to it. Your client will need to use the SMTP (Simple Mail Transfer Protocol) to talk to the server. In SMTP, the client opens a connection and then engages the server in a conversation that goes something like this:

```
Server says: 220 gradin.cis.upenn.edu ESMTP
Client says: HELO plover.com
Server: 250 gradin.cis.upenn.edu Hello mailuser@plover.com
Client: MAIL From: <mjd@plover.com>
Server: 250 <mjd@plover.com>... Sender ok
(And so on...)
```

The usual complaint: ``I opened the connection all right, and I got the greeting from the server, but it isn't responding to my client's commands!" And by now you know the reason why: The client's output to the network socket is being buffered, and Perl is waiting to send the data over the network until there's a whole bufferful. The server hasn't responded because it hasn't received anything to respond to.

Solution: Make the socket filehandle hot. Another solution: Use the `IO::Socket` module; recent versions (since version 1.18) make sockets hot by default.

``When my program terminates abnormally, the output is incomplete!"

When the program exits normally, by executing `die` or `exit` or by reaching the end of the program, it flushes all the buffers. But if the program is killed suddenly, it might exit without getting to flush the buffers, and then the output files will be incomplete. The Unix `kill` command destroys a process in this way and can leave behind incomplete files.

Even worse, a file that exits in this way can leave behind *corrupt* data. For example, imagine a program that writes out a database file. The database file is supposed to contain records of exactly 57 characters each.

Suppose the program has printed out 1,000 records, and then someone kills it and it doesn't have a chance to flush its buffer. It turns out that only 862 complete records have made it into the file, but that's not the worst part. The buffer is flushed every 8,192 bytes, and 57 does not divide 8,192 evenly, so the last record that was flushed to the file is incomplete; only its first 18 bytes appear in the file. The other 39 bytes were still in the buffer, and they're lost. The file is now corrupted, and any program that reads it assuming that each record is exactly 57 bytes long is going to get garbled data and produce the wrong results.

One possible solution to this is to simply make the filehandle hot. Another is to do the buffering yourself: Accumulate 57-byte records into a scalar variable until you have a lot of them, and then write them all at once. A third solution is to use the `setvbuf` method provided by the `Filehandle` and `IO::` modules to make the buffer size an exact multiple of 57 bytes; then it'll never contain any partial records when it's flushed. That looks like this:

```
use IO::Handle '_IOFBF'; # `FBF' means `Fully Buffered'

FH->setvbuf($buffer_var, _IOFBF, 8151);
```

(I picked 8151 here because it's the largest number less than 8K that is a multiple of 57.)

A fourth solution is to manually flush the buffer before it gets completely full. The next section explains how to do this.

Flushing on Command

Sometimes you'd like to have buffering, but you want to control when the buffer is flushed. For example, suppose you're writing a lot of data over the network to a logging service. For efficiency, you'd like buffering, but you don't want the log to get too far out of date. You want to let data accumulate in the buffer for up to ten seconds, and then flush it out, at least six times per minute.

Here's a typical strategy for doing that:

```
if (time > $last_flush_time + 10) {
  my $ofh = select LOG;
  $| = 1;                     # Make LOG socket hot
  print LOG "";               # print nothing
  $| = 0;                     # LOG socket is no longer hot
  select $ofh;
  $last_flush_time = time;
}

... Do something else ...
```

We select the `LOG` filehandle and make it temporarily hot. Then we print the empty string to the filehandle. Because the handle is hot, this flushes the buffer---printing to a hot filehandle always flushes the buffer. Then we return the filehandle to its un-hot state so that future writes to `LOG` will be buffered.

If you're using the `FileHandle` or `IO` modules, there's a simpler interface:

```
$filehandle->flush();         # Flush the buffer
```

It does just the same as the code above.

Other Directions

If for some reason you want to avoid the buffering entirely, you can use Perl's `sysread` and `syswrite` operators. These don't use buffering at all. That makes them slow, but they are often appropriate for tasks like network communications where you don't want buffering anyway. All Perl's other I/O functions, including `write`, `print`, `read`, `<FILEHANDLE>`, and `getc`, are buffered. If you do both buffered and unbuffered I/O on the same filehandle, you're likely to confuse yourself, so beware.

Summary

For efficiency, Perl does not read or write the disk or the network when you ask it to. Instead, it reads and writes large chunks of data to a `buffer' in memory, and does I/O to the buffer; this is much faster than making a request to the operating system for every read or write. Usually this is what you want, but sometimes the buffering causes problems. Typical problems include communicating with conversational network services and writing up-to-date log files. In such circumstances, you would like to disable the buffering. You can do that in Perl by setting `$|=1`. This special variable makes the currently selected filehandle hot, so that the buffer is flushed after every write.

Return to: Universe of Discourse main page | Perl Paraphernalia | Just the FAQs

mjd-perl-faqs@plover.com