Bursty writes to SD/USB stalling my time-critical apps on embedded Linux

Asked 16 years, 3 months ago Modified 7 months ago Viewed 4k times

14

I'm working on an embedded Linux project that interfaces an ARM9 to a hardware video encoder chip, and writes the video out to SD card or USB stick. The software architecture involves a kernel driver that reads data into a pool of buffers, and a userland app that writes the data to a file on the mounted removable device.

I am finding that above a certain data rate (around 750kbyte/sec) I start to see the userland video-writing app stalling for maybe half a second, about every 5 seconds. This is enough to cause the kernel driver to run out of buffers - and even if I could increase the number of buffers, the video data has to be synchronised (ideally within 40ms) with other things that are going on in real time. Between these 5 second "lag spikes", the writes complete well within 40ms (as far as the app is concerned - I appreciate they're buffered by the OS)

I think this lag spike is to do with the way Linux is flushing data out to disk - I note that pdflush is designed to wake up every 5s, my understanding is that this would be what does the writing. As soon as the stall is over the userland app is able to quickly service and write the backlog of buffers (that didn't overflow).

I think the device I'm writing to has reasonable ultimate throughput: copying a 15MB file from a memory fs and waiting for sync to complete (and the usb stick's light to stop flashing) gave me a write speed of around 2.7MBytes/sec.

I'm looking for two kinds of clues:

1. How can I stop the bursty writing from stalling my app - perhaps process priorities, realtime patches, or tuning the filesystem code to write continuously rather than burstily?
2. How can I make my app(s) aware of what is going on with the filesystem in terms of write backlog and throughput to the card/stick? I have the ability to change the video bitrate in the hardware codec on the fly which would be much better than dropping frames, or imposing an artificial cap on maximum allowed bitrate.

Some more info: this is a 200MHz ARM9 currently running a Montavista 2.6.10-based kernel.

Updates:

- Mounting the filesystem SYNC causes throughput to be much too poor.

- The removable media is FAT/FAT32 formatted and must be as the purpose of the design is that the media can be plugged into any Windows PC and read.
- Regularly calling sync() or fsync() say, every second causes regular stalls and unacceptably poor throughput
- I am using write() and open(O_WRONLY | O_CREAT | O_TRUNC) rather than fopen() etc.
- I can't immediately find anything online about the mentioned "Linux realtime filesystems". Links?

I hope this makes sense. First embedded Linux question on stackoverflow? :)

linux video embedded filesystems real-time

Share

Improve this question

Follow

edited Sep 17, 2008 at 14:01

asked Sep 16, 2008 at 22:54

blueshift

6,882 ● 2 ● 42 ● 64

FWIW, I find shorter, more concise titles helpful in glancing over questions to answer. YMMV. – JBB Sep 16, 2008 at 23:40

OK, I'll have a crack.. there's quite a bit of info I felt I needed to get across though. – blueshift Sep 17, 2008 at 13:27

10 Answers

Sorted by: Highest score (default)

10

For the record, there turned out to be two main aspects that seem to have eliminated the problem in all but the most extreme cases. This system is still in development and hasn't been thoroughly torture-tested yet but is working fairly well (touch wood).

The big win came from making the userland writer app multi-threaded. It is the calls to write() that block sometimes: other processes and threads still run. So long as I have a thread servicing the device driver and updating frame counts and other data to sychronise with other apps that are running, the data can be buffered and written out a few seconds later without breaking any deadlines. I tried a simple ping-pong double buffer first but that wasn't enough; small buffers would be overwhelmed and big ones just caused bigger pauses while the filesystem digested the writes. A pool of 10 1MB buffers queued between threads is working well now.

The other aspect is keeping an eye on ultimate write throughput to physical media. For this I am keeping an eye on the stat Dirty: reported by /proc/meminfo. I have some rough and ready code to throttle the encoder if Dirty: climbs above a certain threshold, seems to vaguely work. More testing and tuning needed later. Fortunately I have lots of RAM (128M) to play with giving me a few seconds to see my backlog building up and throttle down smoothly.

I'll try to remember to pop back and update this answer if I find I need to do anything else to deal with this issue. Thanks to the other answerers.

Share Improve this answer

Follow

answered Oct 18, 2008 at 20:47

blueshift

6,882 ● 2 ● 42 ● 64

It should be long enough for you to have completely developed the solution. :-) What did you end up with?
– wallyk Jul 24, 2015 at 18:33

Is 6 years a little slow for a response? Anyway, the above fixes worked. I put a basic control loop around the Dirty stat and shipped hundreds (thousands?) of units with no further changes to this needed in this area after extensive customer use. – blueshift Aug 25, 2021 at 6:43

5

I'll throw out some suggestions, advice is cheap.

- make sure you are using a lower level API for writing to the disk, don't use user-mode caching functions like `fopen, fread, fwrite` use the lower level functions `open, read, write`.
- pass the `O_SYNC` flag when you open the file, this will cause each write operation to block until written to disk, which will remove the bursty behavior of your writes...with the expense of each write being slower.

- If you are doing reads/ioctls from a device to grab a chunk of video data, you may want to consider allocating a shared memory region between the application and kernel, otherwise you are getting hit with a bunch of `copy_to_user` calls when transferring video data buffers from kernel space to user space.
- You may need to validate that your USB flash device is fast enough with sustained transfers to write the data.

Just a couple thoughts, hope this helps.

Share Improve this answer Follow

answered Sep 16, 2008 at 23:18
Drew Frezell
2,668 ● 1 ● 21 ● 13

3

Here is some information about tuning pdflush for write-heavy operations.

Share Improve this answer Follow

answered Sep 17, 2008 at 13:51
JBB
4,841 ● 3 ● 26 ● 26

1 This is good and releavant info. I tried some of these things before but will have another fiddle, now my apps have more instrumentation. – blueshift Sep 17, 2008 at 14:02

2

Sounds like you're looking for linux realtime filesystems. Be sure to search Google et al for that.

XFS has a realtime option, though I haven't played with it.

hdparm might let you turn off the caching altogether.

Tuning the filesystem options (turn off all the extra unneeded file attributes) might reduce what you need to flush, thus speeding the flush. I doubt that'd help much, though.

But my suggestion would be to avoid using the stick as a filesystem at all and instead use it as a raw device. Stuff data on it like you would using 'dd'. Then elsewhere read that raw data and write it out after baking.

Of course, I don't know if that's an option for you.

Share Improve this answer Follow

answered Sep 16, 2008 at 23:17

JBB
4,841 ●3 ●26 ●26

1

Has a debugging aid, you could use strace to see what operations is taking time. There might be some surprising thing with the FAT/FAT32.

Do you write into a single file, or in multiple file ?

You can make a reading thread, that will maintain a pool of video buffer ready to be written in a queue. When a frame is received, it is added to the queue, and the writing thread is signaled

Shared data

```
empty_buffer_queue
ready_buffer_queue
video_data_ready_semaphore
```

Reading thread :

```
buf=get_buffer()
bufer_to_write = buf_dequeue(empty_buffer_queue)
memcpy(bufer_to_write, buf)
buf_enqueue(bufer_to_write, ready_buffer_queue)
sem_post(video_data_ready_semaphore)
```

Writing thread

```
sem_wait(vido_data_ready_semaphore)
bufer_to_write = buf_dequeue(ready_buffer_queue)
write_buffer
buf_enqueue(bufer_to_write, empty_buffer_queue)
```

If your writing threaded is blocked waiting for the kernel, this could work. However, if you are blocked inside the kerne space, then thereis nothing much you can do, except looking for a more recent kernel than your 2.6.10

Share Improve this answer Follow

answered Sep 17, 2008 at 15:00

shodanex
15.4k ● 11 ● 62 ● 92

1

Without knowing more about your particular circumstances, I can only offer the following guesses:

Try using fsync()/sync() to force the kernel to flush data to the storage device more frequently. It sounds like the kernel buffers all your writes and then ties up the bus or otherwise stalls your system while performing the actual write. With careful calls to fsync() you can try to schedule writes over the system bus in a more fine grained way.

It might make sense to structure the application in such a way that the encoding/capture (you didn't mention video capture, so I'm making an assumption here - you might want to add more information) task runs in its own thread and buffers its output in userland - then, a second thread can handle writing to the device. This will give you a smoothing buffer to allow the encoder to always finish its writes without blocking.

One thing that sounds suspicious is that you only see this problem at a certain data rate - if this really was a buffering issue, I'd expect the problem to happen less frequently at lower data rates, but I'd still expect to see this issue.

In any case, more information might prove useful. What's your system's architecture? (In very general terms.)

Given the additional information you provided, it sounds like the device's throughput is rather poor for small writes and frequent flushes. If you're sure that for larger writes you can get sufficient throughput (and I'm not sure that's the case, but the file system might be doing something stupid, like updating the FAT after every write) then having an encoding thread piping data to a writing thread with sufficient buffering in the writing thread to avoid stalls. I've used shared memory ring buffers in the past to implement this kind of scheme, but any IPC mechanism that would allow the writer to write to the I/O process without stalling unless the buffer is full should do the trick.

Share Improve this answer Follow

edited Sep 17, 2008 at 16:22

answered Sep 16, 2008 at 23:10

Ori Pessach
6,833 ● 6 ● 38 ● 51

1

A useful Linux function and alternative to sync or fsync is sync_file_range. This lets you schedule data for writing without waiting for the in-kernel buffer system to get around to it.

To avoid long pauses, make sure your IO queue (for example: /sys/block/hda/queue/nr_requests) is large enough. That queue is where data goes in between being flushed from memory and arriving on disk.

Note that sync_file_range isn't portable, and is only available in kernels 2.6.17 and later.

Share Improve this answer Follow

answered Sep 18, 2008 at 14:30

Zan Lynx
54.3k ● 11 ● 81 ● 134

I don't remember seeing this comment before, but thanks! I may be making use of this information in future work.
– blueshift Jun 1, 2010 at 6:48

1

I've been told that after the host sends a command, MMC and SD cards "must respond within 0 to 8 bytes".

However, the spec allows these cards to respond with "busy" until they have finished the operation, and apparently there is no limit to how long a card can claim to be busy (please, please tell me if there is such a limit).

I see that some low-cost flash chips such as the M25P80 have a guaranteed "maximum single-sector erase time" of 3 seconds, although typically it "only" requires 0.6 seconds.

That 0.6 seconds sounds suspiciously similar to your "stalling for maybe half a second".

I suspect the tradeoff between cheap, slow flash chips and expensive, fast flash chips has something to do with the wide variation in USB flash drive results:

- http://www.testfreaks.com/blog/information/16gb-usb-drive-comparison-17-drives-compared/

- http://www.tomshardware.com/reviews/data-transfer-run,1037-10.html

I've heard rumors that every time a flash sector is erased and then re-programmed, it takes a little bit longer than the last time.

So if you have a time-critical application, you may need to (a) test your SD cards and USB sticks to make sure they meet the minimum latency, bandwidth, etc. required by your application, and (b) peridically re-test or pre-emptively replace these memory devices.

Share Improve this answer Follow

answered Nov 18, 2010 at 4:04

David Cary

5,480 ● 6 ● 56 ● 67

0

Well obvious first, have you tried explicitly telling the file to flush? I also think there might be some ioctl you can use to do it, but I honestly haven't done much C/POSIX file programming.

Seeing you're on a Linux kernel you should be able to tune and rebuild the kernel to something that suits your needs better, eg. much more frequent but then also smaller flushes to the permanent storage.

A quick check in my man pages finds this:

```
SYNC(2)                           Linux Programmer’s
Manual                          SYNC(2)
```

```
NAME
       sync - commit buffer cache to disk

SYNOPSIS
       #include <unistd.h>

       void sync(void);

   Feature Test Macro Requirements for glibc (see
feature_test_macros(7)):

       sync(): _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION
       sync() first commits inodes to buffers, and
then buffers to disk.

ERRORS
       This function is always successful.
```

Share Improve this answer Follow

answered Sep 16, 2008 at 23:03

jfs
240 ● 1 ● 9

0

Doing your own flush()ing sounds right to me - you want to be in control, not leave it to the vagaries of the generic buffer layer.

This may be obvious, but make sure you're not calling write() too often - make sure every write() has enough data to be written to make the syscall overhead worth it. Also, in the other direction, don't call it too seldom, or it'll block for long enough to cause a problem.

On a more difficult-to-reimplement track, have you tried switching to asynchronous i/o? Using aio you could fire off a write and hand it one set of buffers while you're sucking video data into the other set, and when the write finishes you switch sets of buffers.

Share Improve this answer Follow

answered Sep 17, 2008 at 16:59

pjz
43k ● 6 ● 53 ● 60