GWDG
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

GWDG AG-C

High-Bandwidth Linux File IO with O_DIRECT

Dr. Freja Nordsiek

GöHPC

Table of contents

Why

Why do IO?

- Give program data
- Get program results
- Move data to another device

Why

Why do IO?

- Give program data
- Get program results
- Move data to another device

IO Devices:

- terminal
- files
- direct disk access
- sockets
- various other pseudo-files

Simple Examples Without Error Checking

Python

```python
with open('foo.txt', 'rb') as fin:
    data = fin.read()
with open('bar.txt', 'wb') as fout:
    fout.write(data)
```

C with libc

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE * fin = fopen("foo.txt", "r");
    FILE * fout = fopen("bar.txt", "w");

    for (int c = fgetc(fin); c != EOF; c = fgetc(fin))
        fputc(c, fout);

    fclose(fin);
    fclose(fout);
}
```

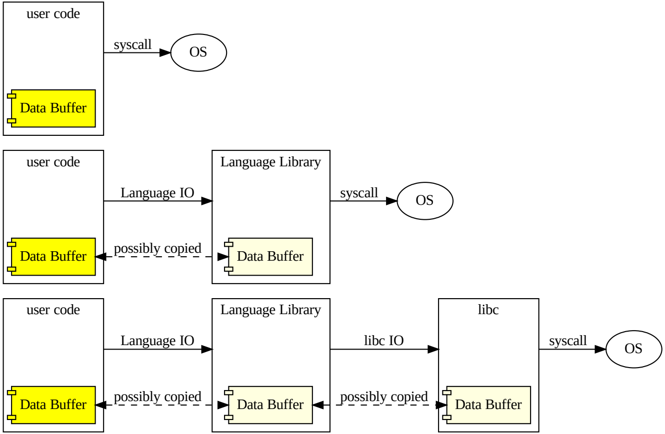
Bash

```bash
data=`cat foo.txt`
echo $data > bar.txt
```

C with POSIX

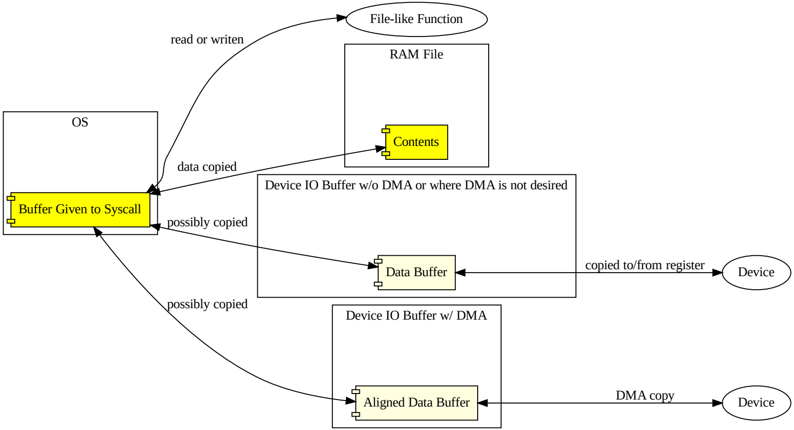
```c
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int fin = open("foo.txt", O_RDONLY);
    int fout = open("bar.txt", O_WRONLY | O_CREAT | O_TRUNC);

    char c;
    while (read(fin, &c, 1))
        write(fout, &c, 1);

    close(fin);
    close(fout);
}
```

How IO Is Done (non-memory-mapped)– User Side



Note: functions can return earlier on the chain depending on buffering, size of data, previous operations, etc.

How IO Is Done (non-memory-mapped) – OS Side



Note: depending on buffering/caching and previous operations, not every syscall results in a read/write.

Where Problems Come From

Alignment for Devices with DMA on OS Side

- Data must be read/written in increments of N bytes
- Read/written data start/end addresses must be multiples of N bytes

Where Problems Come From

Alignment for Devices with DMA on OS Side

- Data must be read/written in increments of N bytes
- Read/written data start/end addresses must be multiples of N bytes

Multiple IO Layers User Side

- 1 – 3+ layers
- worst: wrapper $\rightarrow$ library $\rightarrow$ language lib $\rightarrow$ libstdc++ $\rightarrow$ libc $\rightarrow$ syscall
- Time overhead of each layer
  - Validate arguments
  - Check errors
  - ...
- Each layer might copy data buffer one or more times

Limits

- Device latency
- Device bandwidth limits

Limits

- Device latency
- Device bandwidth limits
- Logic and setup latency
  - Each IO layer requires time to complete
  - Logic for file-like functions
  - Filesystem logic for disks
  - Setting up transfers

Limits

- Device latency
- Device bandwidth limits
- Logic and setup latency
  - Each IO layer requires time to complete
  - Logic for file-like functions
  - Filesystem logic for disks
  - Setting up transfers
- Memory bandwidth limits
  - Data is read and/or written every copy
  - Desktop/mobile CPUs have low bandwidth
  - Server CPUs have more more bandwidth but many more cores
  - Competing with other memory bandwidth intensive operations
  - Crossing NUMA boundaries can reduce bandwidth limit

Performance Limit – Device

- Latency and bandwidth can be looked up or calculated
- Must consider full chain (e.g. network connections)
- Must consider which steps are a/synchronous

Performance Limit – IO Logic Latency

Must be measured (often a distribution) with data copying time subtracted.

$$\text{limit} = \frac{\langle n \rangle}{\langle t_{latency} \rangle}$$

Where $\langle n \rangle$ is average number of bytes read/written each call and $\langle t_{latency} \rangle$ is the average latency of each call.

Increasing $\langle n \rangle$ increases this limit.

Small reads and writes most likely to hit this limit.

Less layers can reduce $\langle t_{latency} \rangle$ depending on buffering (can make worse with some configurations).

Performance Limit – Memory Bandwidth (ignoring NUMA)

Total memory bandwidth for N_{chan} with bandwidth $B_{mem,chan}$ is

$$B_{mem,tot} = N_{chan} B_{mem,chan}$$

IO to/from the device requires 1 read/write plus 1 read and 1 write for each buffer copy N_{copy}. The memory bandwidth limit is then

$$\text{limit} = \frac{B_{mem,tot}}{1 + 2N_{copy}}$$

Reducing N_{copy} improves limit.

Reducing number of layers is easiest way to reduce N_{copy}.

General Strategy for High Bandwidth

- Bigger reads/writes minimize impact of $\langle t_{latency} \rangle$

General Strategy for High Bandwidth

- Bigger reads/writes minimize impact of $\langle t_{latency} \rangle$
- Going to lower level IO to reduce layers
  - Can reduce $\langle t_{latency} \rangle$
  - Reduces N_{copy}

General Strategy for High Bandwidth

- Bigger reads/writes minimize impact of $\langle t_{latency} \rangle$
- Going to lower level IO to reduce layers
  - Can reduce $\langle t_{latency} \rangle$
  - Reduces N_{copy}
- Aligning reads/writes to N bytes for DMA
  - Only possible for lowest level IO (direct syscalls)
  - `O_DIRECT` on Linux
    - $N_{copy} = 0$
    - DMA does all work freeing core for other tasks while IO completes

libc and Linux equivalents

libc calls	Linux calls
#include <stdio.h>	#include <fcntl.h> #include <unistd.h>
FILE *fopen(char *filename, char *mode)	int open(const char *pathname, int flags)
int fclose(FILE *f)	int close(int fd)
int fflush(FILE *f)	int fsync(int fd)
int fseek(FILE *f, long offset, int origin)	off_t lseek(int fd, off_t offset, int whence) off64_t lseek64(int fd, off64_t offset, int whence)
long ftell(FILE *f)	off_t lseek(int fd, 0, SEEK_CUR) off64_t lseek64(int fd, 0, SEEK_CUR)
size_t fread(void *buf, size_t size, size_t count, FILE *f)	ssize_t read(int fd, void *buf, size_t count)
size_t fwrite(void *buf, size_t size, size_t count, FILE *f)	ssize_t write(int fd, const void *buf, size_t count)

Linux file handles are `int`.

On Linux, the following are file handles

- sockets
- stdin, stdout, stderr
- pipes

Open File

```
int fd = open("foo.txt", FLAGS);
```

FLAGS are OR-ed together

Open File

```
int fd = open("foo.txt", FLAGS);
```

FLAGS are OR-ed together

Access FLAGS

read	`O_RDONLY`
write	`O_WRONLY`
read and write	`O_RDWR`

Creation FLAGS

create if not exist	`O_CREATE`
must create	`O_EXCL`
truncate	`O_TRUNC`
append	`O_APPEND`

Open File

```
int fd = open("foo.txt", FLAGS);
```

FLAGS are OR-ed together

Access FLAGS

read	O_RDONLY
write	O_WRONLY
read and write	O_RDWR

Creation FLAGS

create if not exist	O_CREATE
must create	O_EXCL
truncate	O_TRUNC
append	O_APPEND

Synchronization FLAGS

fsync ever write	O_SYNC
non-blocking	O_NONBLOCK

Open File

```
int fd = open("foo.txt", FLAGS);
```

FLAGS are OR-ed together

Access FLAGS

read	`O_RDONLY`
write	`O_WRONLY`
read and write	`O_RDWR`

Creation FLAGS

create if not exist	`O_CREATE`
must create	`O_EXCL`
truncate	`O_TRUNC`
append	`O_APPEND`

Synchronization FLAGS

fsync ever write	`O_SYNC`
non-blocking	`O_NONBLOCK`

Aligned IO FLAGS

aligned reads/writes only	`O_DIRECT`

Getting And Changing FLAGS

Get FLAGS

```
int flags = fcntl(fd, F_GETFL);
```

Change some FLAGS

```
int err = fcntl(fd, F_SETFL, FLAGS);
```

Read And Write

Read

```
ssize_t bytes_read = read(fd, buffer, bytes_to_read);
```

Write

```
ssize_t bytes_written = write(fd, buffer, bytes_to_write);
```

Enable And Disable

Enable O_DIRECT

```
int flags = fcntl(fd, F_GETFL);
int err = fcntl(fd, F_SETFL, flags | O_DIRECT);
```

Disable O_DIRECT

```
int flags = fcntl(fd, F_GETFL);
int err = fcntl(fd, F_SETFL, flags & ~O_DIRECT);
```

Aligned Reads And Writes – Requirements

```
ssize_t bytes_read = read(fd, buffer, bytes_to_read_write);
ssize_t bytes_written = write(fd, buffer, bytes_to_read_write);
```

Requirements

- `buffer` starting address must be a multiple of N
- `bytes_to_read_write` must be a multiple of N

Alignment N

N is generally 512 bytes, but page aligning (4096 bytes usually) reads/writes can result in better performance in many cases (particularly for disks).

Aligned Reads And Writes – When

- If performance gains are worth the trouble
- Sometimes data records are a multiple of N
  - The number of pixels in high resolution cameras is often a multiple of 512 or even 4096
  - Large fixed size records can often be padded to 512 or 4096 bytes with negligible loss of space
- Sometimes something big must be copied (enable `O_DIRECT` for all but the unaligned head and tail)
- Reading a big file sequentially (buffer large aligned chunks at a time)

Making An Aligned Buffer – Direct C Allocation

If direct C calls can be made:

C11 or newer

```c
#include <stdlib.h>

// ...

    char * buf = aligned_alloc(alignment, size);
```

POSIX (includes Linux)

```c
#include <stdlib.h>

// ...

    char * buf;
    int err = posix_memalign(&buf, alignment, size);
```

Freed as normal with `free(buf);`.

Making An Aligned Buffer – From Unaligned Allocation (Method)

Sometimes, one only has access to unaligned allocation.

An aligned suballocation of n bytes can be made in the following steps:

1. Round n up to the nearest multiple of N to get $n_{unaligned}$
2. Allocate $n_{unaligned}$ bytes for the unaligned buffer
3. Round the starting address of the unaligned buffer up to the nearest multple of N to get $a_{aligned}$
4. The aligned buffer of size n starts at address $a_{aligned}$
5. When done with the buffer, free the unaligned buffer

Making An Aligned Buffer – From Unaligned Allocation (Example)

```c
#include <stdlib.h>

// ...

    size_t blocks = n / alignment;
    if (n % alignment != 0)
        blocks++;
    char * buf_unaligned = malloc(blocks * alignment);
    char * buf = buf_unaligned;
    uintptr_t misalignment = (uintptr_t)buf_unaligned % alignment
    if (misalignment != 0)
        buf += (alignment - misalignment);
```

Alternatives When O_DIRECT Is Not Possible

Memory Mapping for Files

- Uses virtual memory system
- File looks like an array (very simple access)
- OS handles aligned reads and writes dynamically in response to reads and writes in the background
- More overhead compared to `O_DIRECT`
- Extra work for large files on 32-bit systems (can only map chunks at a time)