New I/O Design for Chapel

Michael Ferguson (LTS)
May 11, 2011

Design Goals

- Good performance
- Consistent interface
- Flexible
- Parallel Friendly
- Easy C integration
- Handle Errors
- Easy to use

Keys to I/O Performance

- Buffering
- Zero copy
- Read-ahead
- Use the right system calls

Consistent Interface

- In-memory 'files'
- Too many options for I/O system calls:
  - read/write
  - pread/pwrite
  - send/recv
  - splice/vmsplice
  - Kernel asynchronous i/o
  - POSIX asynchronous i/o
  - select/poll/epoll

Flexible (1)

- C's FILE* deficient here. There is no way to manipulate the buffer. Can't push into buffer or pop from buffer in zero-copy way. No standard in-memory file.
- System calls are not simple, do not allow readahead, do not buffer, system dependent
- Need to allow other library writers to work with buffer in zero-copy way (e.g. for parallel I/O)

Flexible (2)

- Chapel's existing I/O system wraps FILE* I/O with the same benefits and hazards
  - No zero-copy buffer management
- But Chapel's existing I/O has no scheme for binary I/O
  - Spec perhaps would mark file as 'binary', but that fails to work with mixed binary/text.
- Chapel's existing scheme is also inflexible. Would like formatting “knobs”

Parallel Friendly

- C's FILE* is NOT parallel friendly because file position is stored in FILE*
- File position also stored in system file descriptor when used with read/write
- Leads to either using #threads open files or race conditions on file position
- Alternative system calls provide answer, but these are not supported by FILE*
  - mmap,pread/pwrite,async io

Easy C Integration

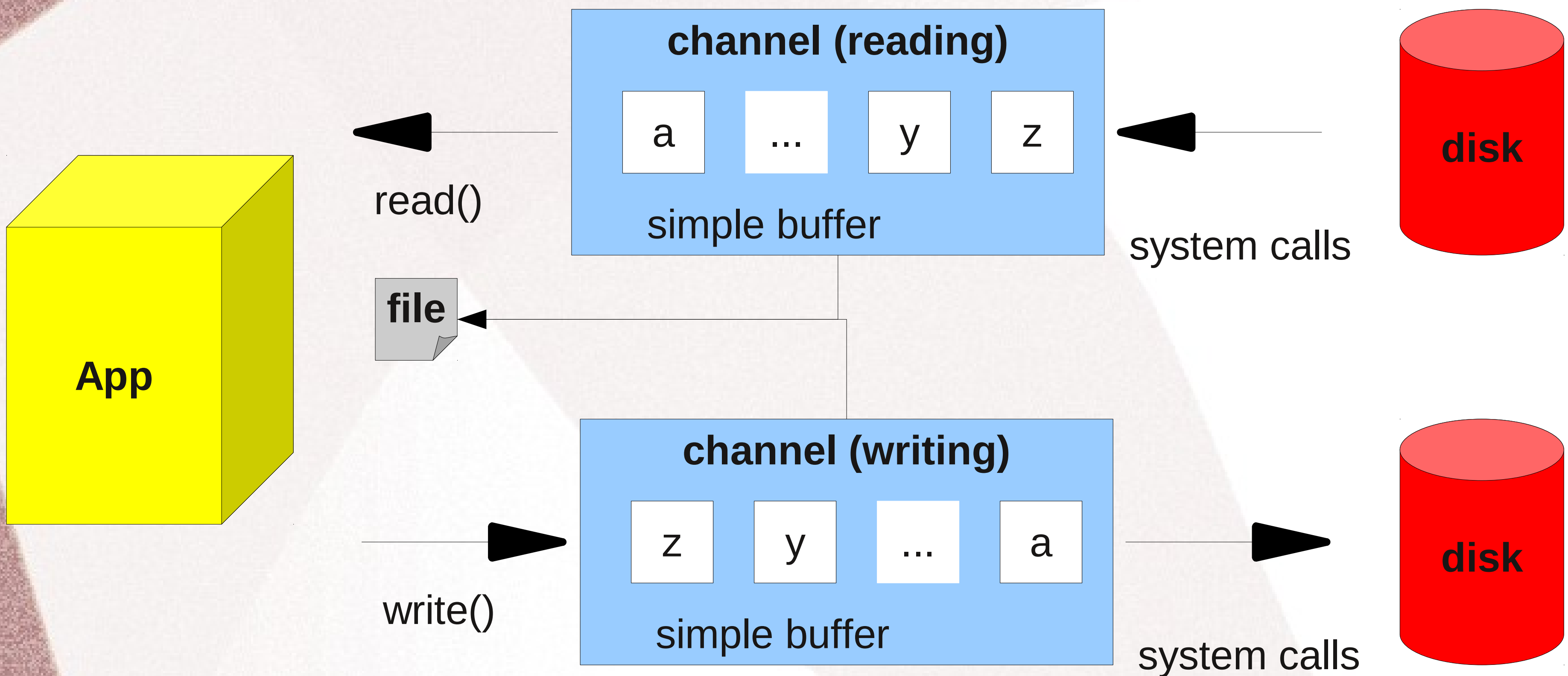
- I/O calls are all C functions
- Everything that can be written in C is written in C
- Chapel module calls C function, includes intelligent dispatching in generic read/write

Handling Errors
Easy to Use

- Handling Errors
  - Current Chapel I/O just stops with halt() if e.g. permissions not satisfied
  - Want programmer to be able to respond to these errors.
- Easy to Use
  - Don't want to have to delete file or other structures, so reference counting is used extensively

New I/O Design

In a Picture



channel (reading)
a
...
y
z
simple buffer
read()
system calls
disk
App
file
channel (writing)
z
y
...
a
simple buffer
write()
system calls
disk

No seek (1)

- No seek call, no file position. Instead, make a channel at a particular offset in a file. Why?
  - In the parallel context, seek calls lead to either (1) too many file descriptors or (2) race conditions on file position.
  - By specifying start/end of a channel within a file, can guarantee no data races on file contents (when channel regions do not overlap)
  - More consistent I/O design; file I/O is more like network I/O (connect).

No seek (2)

- Once I had to write an on-disk in-place radix sort. I had to make my own buffers since the C I/O system is tied to files, and I needed 256 buffers for 1 file. This is easily solved with the channel design.
- I believe that the file vs. channel distinction offers clearer semantics of where a buffer is used. It's easy to create more buffers by using more channels.
- Channels are useful independent of files

Files and Channels

- A file represents e.g. file on disk
- Can't read or write to a file
- Instead, create a channel for some region of a file and read/write to that
- Channel represents a single pass of reading or writing; like a pipe
- Channels include flexible buffers
- Channel data structures protected by a lock

Working Example

```
var f = opentmp();
var writer = f.writer(start=0, end=256);
writer.write(1,2,3,4,5,6,7,8,9,10);
```

I/O with style

- Files and channels have a 'style'. 'style' contains description of how to format the data:
  - In binary? Big endian?
  - Text? Field padding? Precision?
  - Strings escaped with quotes?
  - etc.
- New channels use file's style
- write() or read() functions use channel's style

Working Example

```
var f = opentmp();
var s = default_style;
s.binary = 1;
s.byteorder = big;
var w = f.writer(start=0,end=256,style=s);
w.write(1,2,3,4,5,6,7,8,9,10);
```

Hinting for Performance

- File and channel creating functions take in a 'hints' field, which can specify exactly how the I/O is to be done (ie. pread vs mmap) or just ask library to choose based on how it will be used:
  - Random
  - Sequential
  - Latency
  - Bandwidth
  - Cached
  - Noreuse

Also, can hint a channel to not buffer at all (instead just use read/write/mmap/etc) No buffer = very low overhead of channel creation

Avoiding Overhead

- Reading/writing an array of integers, in native format, to a buffer should amount to a memcpy()
- But we have all of this style stuff adding flexibility, and branches
- Solution: channel has a param kind field:
  - dynamic (default, consult style)
  - native (binary, native endianness)
  - big (binary, big endian)
  - little (binary, little endian)

Readahead/Transactions

- Channels support arbitrary amounts of readahead (data just ends up in the channel buffer)
- channel.mark() will save the current position on a stack within the channel
- Then read/write as much as you like
- call channel.backup() to abort the read/write (ie, put the channel position back to the mark, pop the mark)
- Or, call channel.commit() to keep the changes, and pop the mark.

Bytes and Buffers

- The channel buffer is implemented in a flexible way; more functionality could be exposed
- A buffer is a C++ ish deque (we use a C version), storing sub-regions of bytes objects
- bytes object is ptr and length
- Fast push/pop on either end of a buffer
- Logarithmic search to find bytes at offset
- buffer, bytes reference counted

Handling Errors

- I/O calls take in an error object or nil (default argument is nil)
- If error object is nil and error occurs, we halt()
- If error object not nil, we save error information to the error object
- Also channel.flush and file.fsync are important for reporting errors (e.g. mmap can silently lose data if you use more disk space than exists, but file.fsync would report an error).

Future Work

- Finish integration for Nov. release
- “file” that is actually in-memory buffer
- printf/scanf support (really just translates “%i%f” type string into array of style records, then calls read/write)
- regular expression integration
- building strings on top of bytes
- single-type channels (ie. only works with one type of data)

Chapel GOTCHAs

- These are things that frustrated me during this development:
  - as always, precise error handling. “Internal failure #######” is not helpful...
  - I can't ever decide without changing it 5 times – do I want a record or a class? Some 'style guide' input on this decision would be useful.
  - It would be really nice to have a “How to wrap C types in Chapel for multi-locale operation” document

Implementation Oddities

- For byte order conversion, I use htobe64 and friends (man endian.h); or I manually define them. For this to work, I need to #define _BSD_SOURCE before any standard library #includes
- Tasks will block on blocking I/O (and a core will be idle). I've talked to Kyle (of qthreads) about this and we have a strategy for fixing it in qthreads, but I don't know of a tasking-independent strategy

How did I do?

- Performance
  - low overhead
  - lib chooses syscalls
- Consistency
  - file vs. channel
- Flexibility
  - buffer & bytes
  - mark/etc
- Parallel Friendly
  - file regions
  - no seek
- Easy C integration
- Handle Errors
- Easy to use