IOPipe Library

Steven Schveighoffer

I/O Is Slow

- Moving data from device to memory is slow.
  - Mechanical devices are slow (HDD)
  - External bottlenecks (routers)
  - slow busses.
- Copies are made whenever we move from one area to another.
  - Device to kernel
  - Kernel to userspace
  - Library to application (x N)

Buffering

- Necessary for performance
- Used *only* as an optimization is a wasted opportunity!
- Buffer access is liberating.

Every Copy Counts

- Every copy, the CPU can be wasting cycles.
- Performance is trumped by API/correctness
- But this is an API problem, not inherent!

Use Case: `byLine`

- Read data until a line ending is found.
- Provide the line data in an array/range for processing, saving the rest of the stream for later.
- Reuse buffer for next line.

`FILE *` API solution

- Buffers data for performance reasons
- One `char` at a time.
- No direct buffer access

```
string nextLine(FILE *f) {
    static string line;
    line.length = 0;
    line.assumeSafeAppend;
    int c;
    while((c = fgetc(f)) != -1) {
        line ~= c;
        if(c == '\n')
            break;
    }
    return line;
}
```

Phobos uses `FILE *`

- `std.stdio.File` type uses tricks to get around limitations.
- Code for `readlnImpl` in Phobos is 357 LOC.
- OS specific implementations for best trick (e.g. `getdelim`)
- Unicode delimiters not supported
- Other limitations

iopipe - a Different Approach

- iopipe provides direct buffer access — less copies.
- All D code, so all D API, and all inlineable
- iopipe **`byLine`** code is 170 LOC (one implementation).
- No reliance on opaque shortcuts (and limitations).
- Full Unicode support.
- **iopipe** `byLine` **is 2x faster than fast Phobos** `byLine`**.**
- **iopipe** `byLine` **is 1.5x faster than straight** `getline`

iopipe API

- `Range window()`: Get the current accessible data window as random-access range.
- `size_t extend(size_t)`: Extend current window of accessible data given extra elements, or 0 for "whatever you think is best".
- `void release(size_t)`: Release given elements back to buffer for reuse.
- Note: `char[]` is considered Random Access for iopipe

```
size_t extend(size_t elements = 0) {
    ...
byline_outer_1:
     do {
        auto w = chain.window;
        immutable t = delimElems[0];
        static if(isArray!(WindowType!(Chain)))
        {
            auto p = w.ptr + checked;
            static if(CodeUnitType.sizeof == 1)
            {
                import core.stdc.string: memchr;
                auto delimp = memchr(p, t, w.length - checked);
                if(delimp != null)
                {
                    checked = delimp + 1 - w.ptr;
                    break byline_outer_1;
                }
            }
            else
                ...
```

```
static if(isArray!(WindowType!(Chain)))
{
    auto p = w.ptr + checked;
    static if(CodeUnitType.sizeof == 1)
        ... // memchr
    else
    {
        auto e = w.ptr + w.length;
        while(p < e)
        {
            if(*p++ == t)
            {
                checked = p - w.ptr;
                break byline_outer_1;
            }
        }
    }
    checked = w.length;
}
```

```
byline_outer_1:
     do {
        auto w = chain.window;
        immutable t = delimElems[0];
        static if(isArray!(WindowType!(Chain)))
            ... // memchr, ptr access
        else
        {
            while(checked < w.length)
            {
                if(w[checked] == t)
                {
                    // found it.
                    ++checked;
                    break byline_outer_1;
                }
                ++checked;
            }
        }
    } while(chain.extend(elements) != 0);
```

iopipe `byLine` is an iopipe

- `byLine` is a delimiting iopipe — `extend()` always provides next line.
- Wrapped into range by `iopipe.bufpipe.asInputRange`
- `byLineRange` for optional line endings.

All Arrays Are iopipes

- Complete implementation:

```
size_t extend(T)(T[] arr) {
    return 0;
}

T[] window(T)(T[] arr) {
    return arr;
}

void release(T)(ref T[] arr, size_t elems) {
    arr = arr[elems .. $];
}
```

iopipes Use Chaining

```
foreach (line;
         openDev("file.txt")   // open file
         .bufd                 // buffer
         .decodeText!UTF8      // convert to UTF8
         .byLineRange) { ... }

foreach (line;
         openDev("file.gz")    // open file
         .unzip                // decompress (to ubyte[])
         .decodeText!UTF8      // convert to UTF8
         .byLineRange) { ... }

foreach (line;
         "abc\n123"
         .byLineRange) { ... }
```

Output Is Harder

- Output needs to be pushed, not pulled.
- iopipe uses strictly pull mechanism.
- How to access “source” element in chain?
- Solution: Valves

> Can't you use a template lambda alias argument to pushTo instead, so then you can instantiate it inside pushTo?
>
> something like
> ```
> nullStream!char
>     .bufferedInput
>     .pushTo!(_ => _
>         .arrayCastPipe!ubyte
>         .outputFile("output.txt")
>     );
> ```
> maybe?

I could do that. But I don't like it 😏

- Thank you, John Colvin!

```
import iopipe.valve;
auto oChain = bufd!(wchar) // make a buffered source
    .push!(a => a
           .encodeText!(UTFType.UTF16LE)
           .outputPipe(openDev("resultFile.txt"))
           )
    .textOutput; // turn into a standard output range

foreach(w; input.ensureDecodeable.asInputRange)
    put(oChain, w);
```

Valves

- A valve is a control point along the chain, allowing direct access.
- All iopipe wrappers *must* provide access to next upstream valve.
- A “push” iopipe is really a “pull” iopipe, with a wrapped holding valve

```
auto oChain = bufd!(wchar) // make a buffered source
    .holdingValve      // start the loop
    .encodeText!(UTFType.UTF16LE)
    .outputPipe(openDev("resultFile.txt"))
    .holdingLoop       // close the loop
    .autoFlushed       // flush on destruction
    .textOutput;       // turn into a standard output range
```

Use case: JSON

```
JSONToken jsonTok(Chain)(ref Chain c, ref size_t pos)
if (isIopipe!Chain &&
    isSomeChar!(ElementEncodingType!(WindowType!Chain)))
{
    import std.ascii: isWhite;
    // find a non-whitespace character
    while(true)
    {
        while(pos < c.window.length && isWhite(c.window[pos]))
            ++pos;
        if(pos < c.window.length)
            break;
        if(c.extend(0) == 0)
            return JSONToken.EOF;
    }
```

```
    with(JSONToken) switch(c.window[pos])
    {
    case '{': return ObjectStart;
    case '}': return ObjectEnd;
    case '"': return String;
    case ':': return Colon;
    case ',': return Comma;
    case '[': return ArrayStart;
    case ']': return ArrayEnd;
    case 't': return True;
    case 'f': return False;
    case 'n': return Null;
    case '-': case '0': .. case '9':
        return Number;
    default: return Error;
    }
}
```

```
JSONItem jsonItem(bool replaceEscapes = true, Chain)
   (ref Chain c, ref size_t pos)
{
    JSONItem result; // token and buffer slice
    result.token = jsonTok(c, pos);
    result.offset = pos;

    void validateToken(string expected)
    {
        ... // does stream data match expected data
    }

    with(JSONToken) final switch(result.token)
    {
```

```
case ObjectStart: case ObjectEnd: case Colon:
case Comma:       case ArrayStart: case ArrayEnd:
    result.length = 1;
    ++pos; // skip over the single character item
    break;
case EOF:
case Error:
    break; // no changes to result needed.
case True:
    validateToken("true");
    break;
case False:
    validateToken("false");
    break;
case Null:
    validateToken("null");
    break;
```

```
    case String:
        {
            auto numChars = parseString!replaceEscapes(c,
                                    pos, result.hint);
            ... // setup the result
        }
        break;
    case Number:
        {
            auto numChars = parseNumber(c, pos, result.hint);
            ... // setup the result
        }
        break;
    }
    return result;
}

auto jsonTokenizer(bool replaceEscapes = true, Chain)(Chain c)
{ … }
```

```
template parseJSON(bool inPlace = false, bool duplicate = true, Chain)
{
    alias SType = WindowType!Chain;
    alias JT = JSONValue!SType;
    struct DOMParser
    { … }
    auto parseJSON(Chain c)
    {
        auto dp = DOMParser(JSONTokenizer!(Chain, inPlace)(c));
        switch(dp.parser.next.token) with (JSONToken)
        {
        case ObjectStart:
            return dp.buildObject();
        case ArrayStart:
            return dp.buildArray();
        default:
            throw new Exception("Expected object or array");
        }
    }
}
```

Use Case: JSON Formatter

- Given some JSON input, output that in a given format.
- Validate the syntax, but don’t need to parse into DOM.
- LIVE DEMO!!!!

The Future

- XML
- MySQL native driver
- vibe.d event-driven i/o support
- Phobos(?)

End

https://github.com/schveiguy/iopipe