Browsix

A UNIX-like process-model and kernel for the browser

Bobby Powers
bobbypowers@gmail.com

Craig Greenberg
q7h0u6h7@gmail.com

ABSTRACT

This paper introduces Browsix, a UNIX-like processing model, monolithic kernel, shell, and userspace designed to run in modern web browsers.

1. INTRODUCTION

As noted by Vilk et al., web browsers have become a *de facto* universal operating system and attractive platform for application developers[3]. Browsers expose a rich range of services to programs, such as child-process and network IO, and are continuing to expand the range of functionality they provide to web applications through additonal Javascript APIs, including primitive cooperative multitasking[1] and low-level hardware interaction like Bluetooth[4]. Despite all of this, the document object model (DOM) and event-based programming model provided by the browser are both a foreign and resource contstrained system compared to what the majority of developers are used to. Traditional JavaScript execution competes with in-browser work like layout and painting for CPU time in the single event loop of the browser. This is partially addressed by Web Workers, but the asynchronous nature of communication with Web Workers combined with the restrictions for what can run in them make them tedius to work with.

Javascript and its event-based programming model have become popular outside of web browsers - node.js (sometimes referred to as just node) is the canonical example. node.js enables the creation of high performance web servers by pairing fast single-threaded execution of JavaScript code by the V8 Javascript engine with asynchronous APIs for resources like the file system, network, and process management. Souci and Lemaire[2] provide a good overview of node's architecture.

There have been multiple attempts to bring node's APIs to the Browser. As part of Vilk et al.'s work on Doppio, an in-browser JVM, several large node APIs were ported to the browser, most notably the fs (filesystem) module. The fs module is available as a Doppio sub-project named BrowserFS. BrowserFS provides multiple file system backend implementations, such as an in-memory, XMLHttpRequest, dropbox and an overlay filesystem. This project is roughly structured like the Linux VFS - all of these disparate backends are accessible through a unified API that implements the interface specified by the node `fs` module. In this project we use BrowserFS to provide a shared filesystem to all of our processes.

In addition to Doppio and BrowserFS, there are tools designed to let developers program against node APIs and use node's synchronous `require('$module')` module system. These tools provide a compilation-like step where JavaScript source programs are parsed, transitive dependencies resolved, and all of the required code are bundled into a single JavaScript file. As part of this, tools provide implementations of node modules, such as the buffer and http modules. Examples of this approach are Browserify, WebPack and rollup.js. These projects typically restrict the APIs they provide to those that can be cleanly implemented on top of browser APIs. For example, Browserify provides a http module, but only implements the client side of the API.

This project has several contributions:

- An implementation of a traditional monolithic UNIX-like kernel, userspace, and syscall abstractions in the browser, utilizing the Web Workers API. We refer to this as the process model, and it includes asynchronous delivery of signals like `SIGCHLD`.
- A port of the node.js programming environment to the browser on top of our process model, including the ability to spawn child processes. We call this `browser-node`.
- A collection of traditional UNIX utilities, implemented as node.js applications in TypeScript. These utilities run unmodified in both our browser environment and under node.js on Mac OS X and Linux.
- A bash-like shell that enables the composition of utilities into pipelines (sometimes called filters).
- A web-based UI (terminal) for interacting with this shell that works in all modern browsers.

Combined, these contributions form the Browsix programming environment.

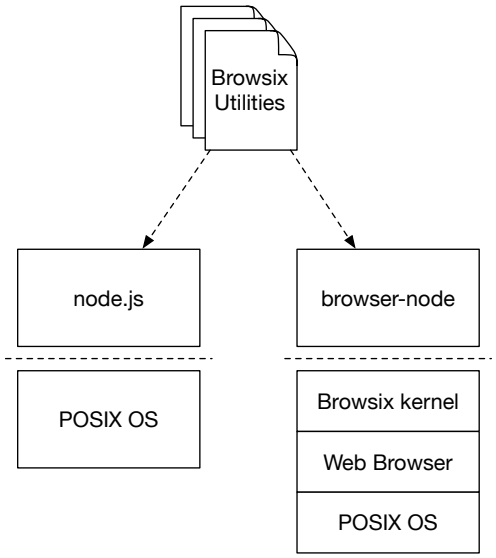

Figure 1: Utilities implemented for Browsix were designed to run in 2 execution environments - node.js on a traditional OS, as well as in a browser through `browser-node` and the Browsix kernel.

2. APPROACH

In order to enable utilities to be tested and debugged independently of our kernel and process model implementation, all utilities were implemented as standard node.js applications depending only on the node.js standard library, with two small exceptions. The shell needs the ability to create UNIX pipes using the `pipe(2)` syscall. node.js uses pipes internally to communicate between threads, as well as with a certain class of child processes, but this syscall is not exposed anywhere in the node.js API. node.js does expose named pipes in its net module, but it was simpler to develop a small package to expose an API allowing use of `pipe(2)` from JavaScript and TypeScript. A similar choice was made to develop a package to expose `getpriority(2)` and `setpriority(2)`, for use in `nice(1)`.

To run these utilities in the browser, we developed `browser-node` and the Browsix kernel. `browser-node` is a port of node.js to our in-browser process model. `browser-node` utilizes over a dozen of node.js's pure-JavaScript modules directly. The only modifications to most of the node stdlib were to remove the use of let and const (as Safari does not support them), and to rewrite their require statements to be local. The one exception was the `internal/child_process` module. This was hand-modified to remove dependencies on node's networking libraries.

Some of the modules in the node stdlib, like `'internal/util'` are self-contained - it doesn't depend on any other modules, or the transitive set of dependencies only includes other pure-JavaScript modules in the stdlib. However, many important modules make foreign function interface (FFI) calls into C++ functions and methods that are part of the node runtime. This set of C++ functions and methods are collectively reffered to as the bindings. `browser-node` provides pure-JavaScript (written in TypeScript, then compiled to JavaScript) implementations of these functions and methods. There are 2 major types of bindings that had to be provided - those that exist for performance reasons, and those that glue JavaScript to the underlying operating system and libraries. There are some bindings, like those used by the `buffer` module, that are written in C++ for performance reasons, especially to avoid generating garbage. Other bindings, like that for the `fs` module, perform syscalls to the Browsix kernel, and invoke callbacks (back into the node JavaScript libraries) after receiving responses.

A consequence of this approach is that only async APIs are implemented. No synchronous APIs that require invoking a system call, such as the methods ending in *Sync* in `fs`, are available.

3. SYSTEM DESIGN

The system was designed as 5 major components: a kernel, `browser-node`, a shell, UNIX utilities, and a Web Terminal for the shell. This approach has led to a system that is straightforward to extend, and is easy to compare to traditional UNIX programming environments in terms of behavior.

3.1 Kernel

The kernel is modeled after a standard classic UNIX kernel, with some inspiration taken from Linux. Its main data members are a single shared filesystem tree provided by BrowserFS, along with a list of currently running and recently exited processes. While BrowserFS provides a single filesystem root, there maybe several instantiated filesystems, combined either through the use of an overlay filesystem or through filesystem mountpoints.

The kernel communicates with process running in Web Workers (referred to as userspace processes) over a MessagePort provided by the browser. This has several consequences. First, messages sent to and from Web Workers are copied, resulting in a shared-nothing architecture. This combined with the way workers are isolated and scheduled means that the syscall roundtrip path is orders of magnitude more expensive than on modern operating systems like Linux. There is a mechanism to transfer ownership of certain types of objects (notably ArrayBuffers), but this is an optimization that is not used in this project. Second, because communication with the worker is based on message passing and JavaScript has no control over message buffer sizes or the schedulability of the worker, there is no way to enforce that processes do not continue to execute while waiting for the results of a syscall from the kernel without cooperation. This is a major point - worker processes are essentially cooperatively scheduled with respect to the kernel.

There is a small 'syscall' layer (module) that lives in the userspace process. It is useful to think of that as conceptually part of the kernel - it is the ABI that the kernel provides to processes; processes do not need to know anything about message passing, they simply invoke system calls, providing a callback to receive the result as the final argument. The syscall layer also provides a way to register for delivery of signals, such as `SIGCHLD`.

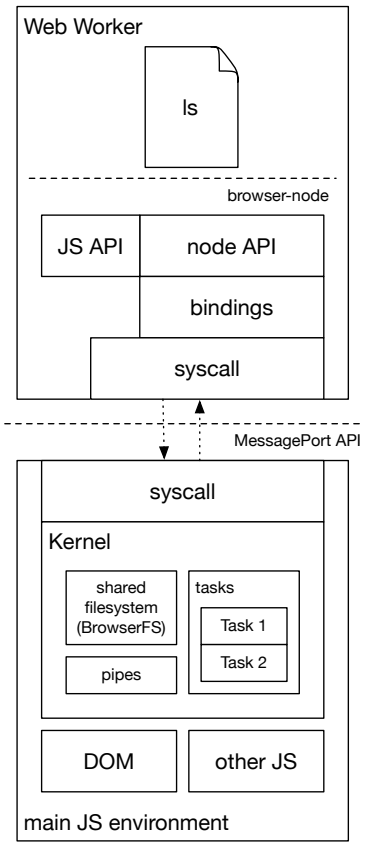

Figure 2: The Browsix stack.

3.1.1 Fork and Spawn

The traditional method of launching a child process on UNIX is to invoke `fork(2)`, perform some bookkeeping work in the child (like closing unused FDs), and then calling `execve(2)` to transfer control to a new executable. This approach is untenable in the browser because of the use of `fork` – the kernel doesn't have control over the memory layout or stack of WebWorkers.

Instead, we implement a new system call we call `spawn`. Spawn is very similar to the node.js `child_process.spawn` function. It allows a process to specify an executable to run, the arguments to pass to that executable, along with control over resources inherited by the new Task. Resouces include file descriptors, the current working directory, and the environment.

3.1.2 Task Scheduler

As Web Workers cannot be explicitly preempted by the kernel, the kernel requires cooperation with processes to ensure fairness. In this case the process is `browser-node`, not the dozen-and-a-half utility programs. Priority is made additionally difficult by the single-threaded + event-based nature of the browser environment. The Browsix kernel implements a fairly primitive scheduling algorithm, where each of the 40 standard priority levels have their own queue. Each queue is in turn checked for tasks, and the first runnable task (where system call results are queued up) is chosen. This can in theory result in starvation of lower-priority tasks, but in practice has not been an issue.

3.2 `browser-node`

As discussed in the Approach section, browser-node provides an execution environment for unmodified node.js programs in the browser. The JavaScript-facing API of node is itself written in JavaScript.

3.3 Shell

Taking inspiration from *nix systems, the shell was implemented as a utility (see next section). Among the advantages of doing so is that the shell can be launched like any other utility (including being lauched by other shell instances). The shell is composed of six main components:

- *Tokenize* the input statement. This turned out to be quite simple. Unlike more sophisticated shells that include the '>' and '<' redirect operators, the only operator in our shell is pipe. This allowed us to simply split the input statement into a series of commands separated by pipe characters.
- *Parse* the token sequence into a sequence of commands. Included in this step was simple path-expansion (limited to expanding paths to utilities) and syntax checking (verifying that each pipe was surrounded by commands).
- *Setup pipes*, a total of n-1 pipes joining n commands. Each pipe consists of two file descriptors– one to a read-only buffer and the other to a write-only buffer. Piping one command to another simply required replacing stdin with the read file descriptor of the pipe preceding the command and replacing stdout with the write file descriptor of the pipe following the command. The first command uses the process's stdin and the last command uses the process's stdout.
- *Spawn and execute children*, which required the use of the spawn system call as described above in the subsection on Fork and Spawn.
- *Collect exit codes and exit*, returning a 0 exit code only when all child processes returned a 0 exit code.

3.4 Utilities

As discussed above, a number of standard UNIX utilities were implemented from scratch in TypeScript using standard node.js APIs. Of note was the use of node's Stream objects rather than directly performing `read(2)`s and `write(2)`s on file descriptors. This led us to implementations that feel very at home in the node ecosystem, rather than looking like transliterations of C programs. Additional information is given in the Approach section above.

3.5 Web Terminal

The web terminal is implemented using Web Components and the Polymer library. This allows us to define a single new tag for use in the HTML of any application, where a shell can be inserted in 2 lines, by first doing an HTML import of the `"browsix-terminal.html"` file, and then by placing one or multiple `"<browsix-terminal>"` tags on the

page. This terminal web component owns a kernel instance, and invokes methods on the kernel to evaluate commands and pipelines.

4. RESULTS

The resulting system provides a similar-feeling environment to that of classic UNIX, BSD, and Plan9 systems, where commands have minimal options and produce plain-text output.

4.1 Scheduler

In order to ensure that higher-priority tasks are given preference over other tasks, it was necessary to insert a small delay between when a system call request comes in and when its result is sent back to the client (currently 2 milliseconds). Without this, priority isn't preserved in the face of internal kernel callbacks scheduled with `setInterval`, and even with this delay the scheduler test sometimes fails in Chrome (but not Firefox). This could potentially be due to the more complicated scheduling algorithm chrome implements in an attempt to improve responsiveness.

The scheduler was able to be tested by use of a pair of programs written in TypeScript using node.js APIs. The first program (named cpu-intensive-program) simulates a CPU intensive program that does a mix of useful work and system calls. It first adjusts its priority with a call to `setpriority(2)`, and then performs a chain of SHA1 digest computations. The priority and number of loop iterations are given as command line parameters. The second program, called priority-test, spawns 2 children, each of which is an instance of the the first program that differ only in their priority. It then waits for both to complete and if the lower priority (more nice) task completes first priority-test reports an error and exits with a non-zero code, otherwise it exits normally.

Because we implement the node `child_process` API, implementing this test of priority scheduling that runs both on the desktop and in the browser was trivial. A caveat was that in order to have the test work reliably on the desktop, it was necessary to simulate a single-processor environment. Details for how to do this can be found as comments in the source of the priority-test program.

5. DISCUSSION

A downside of the approach taken is that it doesn't fit the typical use case the Chrome and Firefox developer tools were designed around. The BrowserFS file system pulls down all of our utilities using XMLHttpRequests upon boot, and stores the context as binary Buffers. When the Browsix kernel receives a `spawn` system call, it requests the file from the filesystem, and then must convert the text JavaScript contents back into a UTF-8 string for execution in a Web Worker. This process results in a Blob URL that is unique each time a new Web Worker is started.

With regular scripts included with `<script>` tags and in workers using `importScripts`, you can browse the script source in the browser's debugger and insert a breakpoint that will be triggered the next time the given line is executed, even across page reloads. A result of using Blob URLs is that the browser is unable to associate the text being executed in a Web Worker with the original web request, and breakpoints do not persist across Web Worker start/stop cycles. Additionally, our Web Workers run very quickly, typically in well under 100 milliseconds, making it very hard to manually hit the 'pause' button in the debugger. Chrome used to have a mechanism to pause a Web Worker upon startup, but that has been removed in current versions of Chrome.

To deal with this, kernel instances have a debug attribute. If this evaluates to true, the kernel delays sending the init message to a worker. The init message contains a processes arguments and environment. With this, developers have enough time to manually tell the browser to pause execution in the Worker (which will happen when it receives the init message).

5.1 Further work

The system call layer that `browser-node` uses to communicate with the kernel is not dependent on other parts of `browser-node`. It should be possible to use this layer to port other runtimes into this process model, such as the Doppio JVM, emscripten apps built with the emterpreter option, and GopherJS.

5.1.1 Terminal

The terminal currently works by taking a line of input, executing it as a shell, and displaying the results when the command finishes. This is a fine first step, but interactive UNIX programs (like the venerable `ed`) expect that their stdin is hooked up to a terminal and that writes to stdout appear on the terminal before program termination. We don't expect that implementing this functionality on top of the existing project will require significant restructuring.

5.1.2 Pseudo-Filesystems

Linux and Plan9 before it expose a lot of functionality through the filesystem, including devices, information on the process tree, and hardware information. This is appealing because programs that want to interact with devices or, for example, the process tree don't need to learn additional system calls, they just need to read from specific files and directories. An interesting area for future work is providing access to browser APIs to process in Web Workers through pseudo-filesystems.

5.1.3 Scheduler

Additional integration with processes that run in workers, like `browser-node` and Doppio would result in better scheduling decisions. Doppio yields the main event loop for reasons of interactivity using a set of heuristics, it would be preferrable for browser-node to do something similar. It would also be useful to have a signal-like approach where the kernel could inform a task that it should yield. Ideally, however, the browser should expose the ability to change the OS-level priority of Web Workers.

6. CONCLUSION

This paper introduced and described the Browsix programming environment, including the kernel, `browser-node`, a

number of UNIX utilities implemented in TypeScript, a shell, and web-based Terminal UI.

7. REFERENCES

[1] R. McIlroy. Chrome 47 beta: Idle time work, splash screens, and desktop notification management. `https://blog.chromium.org/2015/10/chrome-47-beta-idle-time-work-splash.html`. Accessed: 2015-10-22.

[2] B. Souci and M. Lemaire. An inside look at the architecture of nodejs. Technical report, McGill University, 2014.

[3] J. Vilk and E. D. Berger. Doppio: breaking the browser language barrier. In *ACM SIGPLAN Notices*, volume 49, pages 508–518. ACM, 2014.

[4] J. Yasskin. Web bluetooth. `https://webbluetoothcg.github.io/web-bluetooth/`. Accessed: 2015-10-22.