WASI Preview2

WASI Subgroup

June, 2022

Outline

- Introducing WASI Preview2
- Development Plan
- WASI Preview2 Commands
  - How do we define Commands?
  - Command inputs and outputs
    - Command-line arguments
    - Environment variables
    - Preopens
    - Exit codes
  - Stream I/O
  - Stderr

The evolving landscape since “wasi_snapshot_preview1”

- Witx aimed to anticipate where interface types and module linking were going.
- Interface types and module linking evolved a lot, and merged into the component model.
- Some features that we previously discussed for WASI are being incorporated into the component model, including:
  - Weak imports
  - Commands and Libraries (formerly “Reactors”)
  - I/O streams
- The component model has become the focus for many topics important to WASI:
  - Cross-language interoperability
  - Robust composition
  - Declarative interfaces
  - Defining an IDL for describing APIs

Introducing WASI Preview2

- A proposed major new iteration of WASI
  - A successor to “wasi_snapshot_preview1”
  - A new island of stability that we can implement in upstream toolchains
- Wit-based tooling
  - Expressive types, including lists, variants, records, errors, handles, streams, and more!
  - Aligned with the component model (but doesn’t require the component model)
  - Easy-to-use bindings
    - Less need for languages to call through libc
    - Less pressure for “virtual filesystem” APIs
  - Design around IDLs rather than ABIs
- New APIs, such as wasi-sockets
- Work toward a milestone, plan for the future

WASI Preview2: Other improvements

- Removal of the “rights” system
- A more convenient and robust preopen system
  - No more `--dir`!
- More capability-based APIs
- Revamp how readdir works
- Lots of other improvements

WASI Preview2: Development Plan

- Keep the feature set scoped to keep the milestone achievable
  - No support for async yet
  - No support for linking multiple components together yet
  - No support for DLLs yet
  - But, plan for the future which will have these
- Use Github project trackers to track work items
  - https://github.com/orgs/WebAssembly/projects/1
  - https://github.com/orgs/bytecodealliance/projects/6

WASI Preview2: Commands

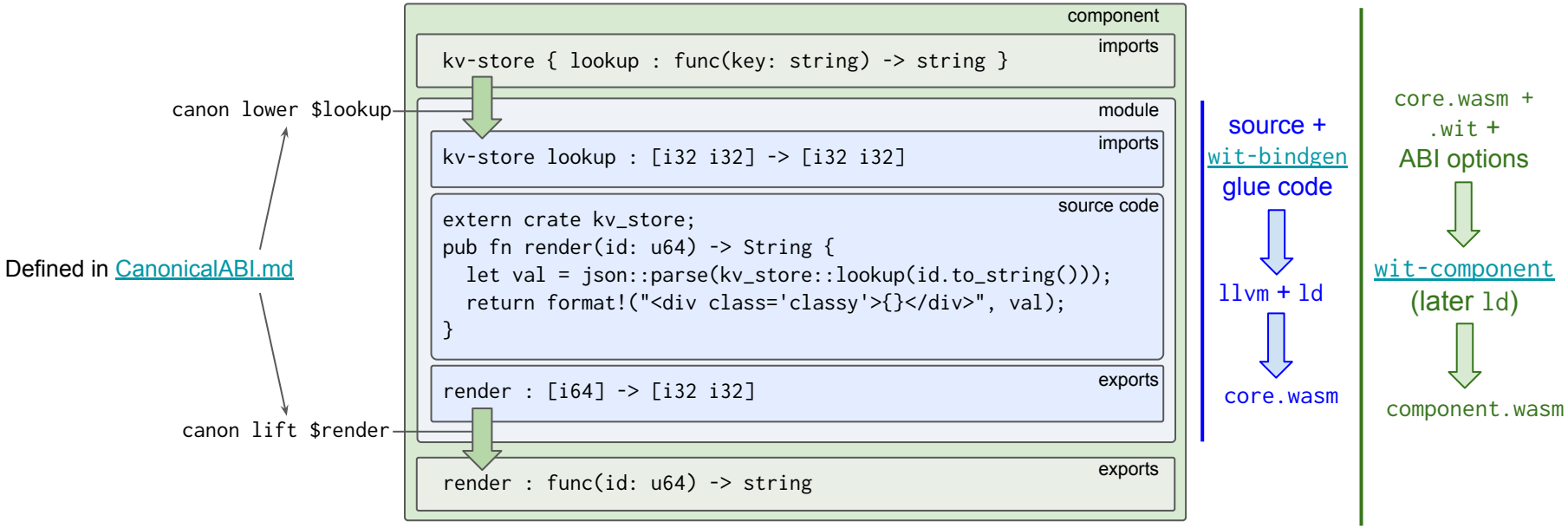
- With the component model, we can define a command:
  - Code runs from the wasm start function
  - Instance has a lifetime scoped to the start function call
    - Arguments are *value imports*
    - Result is a *value export*
  - No other exports
- This encodes the basic properties of a command
  - All one can do with it is run it and read out the result.
  - Command instances can’t be re-entered while running.
  - Command instances don’t live after the program has exited.

Define the core-Wasm ABI using the Canonical ABI

- Components are a simple and powerful way to model commands and other useful concepts.
- But what about engines that don’t support the component model directly?
  - At this time, this is all engines.
- The Canonical ABI defines core-wasm ABIs for component APIs.
- For the Preview2 milestone, engines can implement the Canonical ABIs directly.
  - We can still get all the benefits of designing APIs in terms of Wit
  - And engines can still have relatively simple ABIs to implement
  - And it gives us clear paths to upgrading things in the future

Background: synchronous canonical ABI

Slide credit: Luke Wagner's Component Model Async Support



Profiles (in the component model)

A *profile* describes a component shape:

- What can the imports be?
- What can the exports be?

cli/Command.world (“.world” is my favorite proposed file extension for profiles):

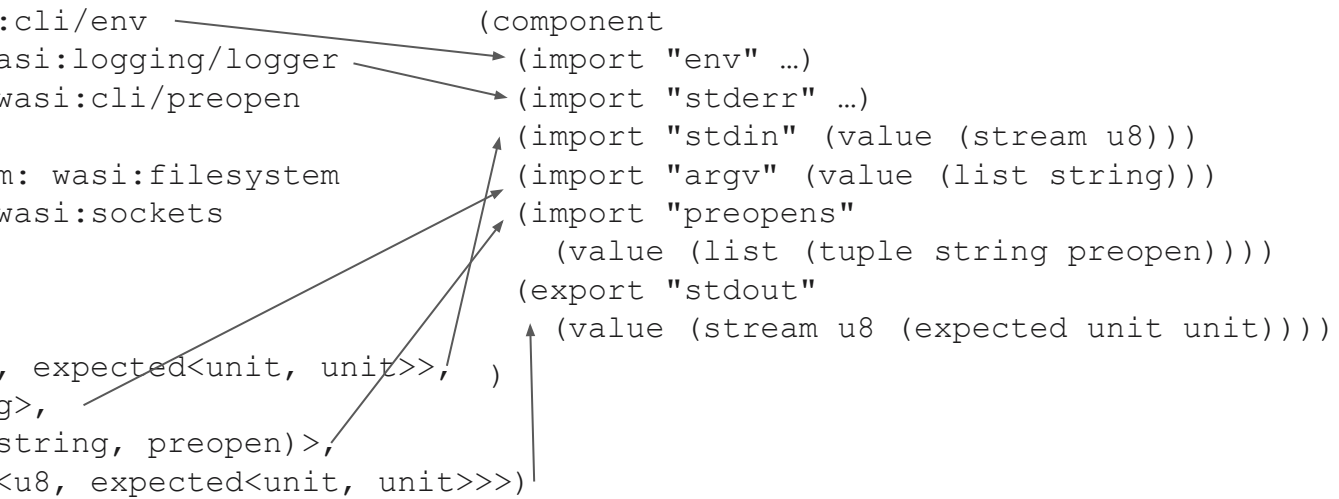
```
depends on env: wasi:cli/env
depends on stderr: wasi:logging/logger
depends on preopen: wasi:cli/preopen

depends on filesystem: wasi:filesystem
depends on sockets: wasi:sockets
…

command(
    stdin: stream<u8, expected<unit, unit>>,
    argv: list<string>,
    preopens: list<(string, preopen)>,
) -> (stdout: stream<u8, expected<unit, unit>>>)
```

Component pseudo-code:

```
(component
  (import "env" …)
  (import "stderr" …)
  (import "stdin" (value (stream u8)))
  (import "argv" (value (list string)))
  (import "preopens"
    (value (list (tuple string preopen))))
  (export "stdout"
    (value (stream u8 (expected unit unit))))
)
```



WASI Preview2: Command-line arguments

- Plan for the future: Typed Main
  - Instead of passing strings to main, what if we could pass typed values, like streams or handles?
  - Strings:
    - require ambient permissions to resolve
    - require an implicit agreement about how to interpret them
    - require an implicit agreement on which namespace to resolve them in
    - carry spurious information
  - Streams and handles:
    - carry permissions with them
    - have types that describe what can be done with them
- Adoption
  - Level 0: User writes int main(int argc, char **argv), or equivalent
    - Handles passed in with preopens
  - Level 1: Like level 0, but user writes a Wit and a wrapper is generated
  - Level 2: Toolchain support for programs described entirely by Wit interfaces
- Work toward a milestone: Preview2 will focus on Level 0. Args are `list<string>`

Environment variables

cli/env.wit:

```
*: string
```

This is the value-import syntax.

The `*` here is a special profile identifier that lets components import environment variables with any key names.

- Programs will declare at build time which environment variables they will read.
- Wasi-libc will present a C-compatible `getenv` API.

Preopens

- Preopens in “wasi_preview_snapshot1”
  - `--dir` is inconvenient
  - Host paths are still exposed to content
- Preopens in WASI Preview2
  - “Allow the program to open the filesystem paths I pass to it”
    - “`grep red foo/bar.txt`”
  - Automatically infer preopens
    - Strings containing “/” are paths
    - Prefix “%” introduces special modifiers, eg. “%=” to force something to be an uninterpreted argument
  - Blind preopen strings with UUIDs
    - UUIDs can be filtered back on stdout
  - Program is passed a list<(string, preopen)>
    - The strings are UUIDs
    - The `preopens` are handles, where specific preopens can be defined as subtypes of the `preopen` resource.

Preopens example

- User types in:
  - `grep -r flowers /home/sunfishcode/here.txt /some/dir`
- Application sees:
  - `wasm -r flowers /args/cdce-9eea-496e1891882e.txt /args/6ba6-a27f-653235b72a82`
- Application prints:
  - `/args/cdce-9eea-496e1891882e.txt: And rain will make the flowers grow`
  - `/args/6ba6-a27f-653235b72a82/there.txt: A few flowers at his feet and above him the stars.`
- User sees:
  - `/home/sunfishcode/here.txt: And rain will make the flowers grow`
  - `/some/dir/there.txt: A few flowers at his feet and above him the stars.`

Exit status

- The command return type is `(stdout: stream<u8, expected<unit, unit>>>)`
  - Stdout is a stream which the program creates and returns
  - u8 means this is a stream of bytes
  - The second parameter to stream is the “end” value.
    - `expected` takes two type parameters: a type for a success value, and a type for a failure value.
    - For a command, these are both `unit`.
      - This translates into a boolean success/failure.
      - For more detail, we may want to investigate defining a custom error type.

Stream I/O

```
extern "C" {
    /// Consume some data from a stream.
    fn read_bytes(subtask: u32, ptr: *mut u8, nelem: usize, end: *mut u8) -> (ReadStatus, usize);

    /// Write some data to a stream.
    fn write_bytes(task: u32, ptr: *const u8, nelem: usize) -> (WriteStatus, usize);

    /// Write some data to a stream and end it.
    fn return(task: u32, ptr: *const u8, nelem: usize, end: *const u8) -> (WriteStatus, usize);

    /// Wait for progress on a task or subtask.
    fn wait() -> (EventKind, u32, usize);
}

enum ReadStatus {
    // ptr must stay valid until `ReadComplete`.
    Wait,
    // T* written to ptr, n = |T*|.
    Ready,
    // T*U written to ptr, n = |T*|.
    AtEnd,
}
```

```
enum EventKind {
    ReturnComplete,
    ReadComplete,
    WriteComplete,
    Cancelled,
}
```

```
enum WriteStatus {
    // ptr must stay valid until `WriteComplete`.
    Wait,
    // `n` (<= `nelem`) values written.
    Ready,
}
```

Stream I/O

- In the full async vision, streams will be actually async
- In the Preview2 milestone:
  - `read_bytes` just does bookkeeping
  - `wait` does synchronous I/O
- Either way, synchronous users, such as libc APIs, will use `wait` to implement synchronous behavior
  - Libc's read will do `read_bytes` and then immediately `wait`.

Stderr

- Stdout is a stream
  - This allows commands to participate in component-model stream pipelines
- But, streams do things which aren't relevant to stderr
  - Async, with backpressure, scheduling
  - Error reporting
- Let's make stderr a simple synchronous logging API
- Libc can still present it as a fd for compatibility
- This gives non-CLI environments more flexibility
  - Eg. `console.log`

```
log: func(
        level: level
        context: string
        message: string
)
```

```
enum level {
        trace,
        debug,
        info,
        warning,
        error,
}
```

Next steps

- Discuss the proposal with the Subgroup
  - Should we take a vote?
- Convert these slides into design documents
  - Where should they live?
- Add more items to the WASI Preview 2 project tracker
- Start figuring out who wants to do that, and let’s do it!