Introduction

Wasmtime is a standalone runtime for WebAssembly, WASI, and the Component Model by the Bytecode Alliance.

WebAssembly (abbreviated Wasm) is a binary instruction format that is designed to be a portable compilation target for programming languages. Wasm binaries typically have a `.wasm` file extension. In this documentation, we'll also use the textual representation of the binary files, which have a `.wat` file extension.

WASI (the WebAssembly System Interface) defines interfaces that provide a secure and portable way to access several operating-system-like features such as filesystems, networking, clocks, and random numbers.

The Component Model is a Wasm architecture that provides a binary format for portable, cross-language composition. More specifically, it supports the use of interfaces via which components can communicate with each other. WASI is defined in terms of component model interfaces.

Wasmtime runs WebAssembly code outside of the Web, and can be used both as a command-line utility or as a library embedded in a larger application. It strives to be

- **Fast**: Wasmtime is built on the optimizing Cranelift code generator.
- **Secure**: Wasmtime's development is strongly focused on correctness and security.
- **Configurable**: Wasmtime uses sensible defaults, but can also be configured to provide more fine-grained control over things like CPU and memory consumption.
- **Standards Compliant**: Wasmtime passes the official WebAssembly test suite and the Wasmtime developers are intimately engaged with the WebAssembly standards process.

This documentation is intended to serve a number of purposes and within you'll find:

- How to use Wasmtime from a number of languages
- How to install and use the `wasmtime` CLI
- Information about stability and security in Wasmtime.
- Documentation about contributing to Wasmtime.

... and more! The source for this guide lives on GitHub and contributions are welcome!

Using the `wasmtime` CLI

In addition to the embedding API which allows you to use Wasmtime as a library, the Wasmtime project also provides a `wasmtime` CLI tool to conveniently execute WebAssembly modules from the command line.

This section will provide a guide to the `wasmtime` CLI and major functionality that it contains. In short, however, you can execute a WebAssembly file (actually doing work as part of the `start` function) like so:

```
$ wasmtime foo.wasm
```

Or similarly if you want to invoke a "start" function, such as with WASI modules, you can execute

```
$ wasmtime --invoke _start foo.wasm
```

For more information be sure to check out how to install the CLI, the list of options you can pass, and how to enable logging.

Installing `wasmtime`

Here we'll show you how to install the `wasmtime` command line tool. Note that this is distinct from embedding the Wasmtime project into another, for that you'll want to consult the embedding documentation.

The easiest way to install the `wasmtime` CLI tool is through our installation script. Linux and macOS users can execute the following:

```
$ curl https://wasmtime.dev/install.sh -sSf | bash
```

This will download a precompiled version of `wasmtime`, place it in `$HOME/.wasmtime`, and update your shell configuration to place the right directory in `PATH`.

Windows users will want to visit our releases page and can download the MSI installer (`wasmtime-dev-x86_64-windows.msi` for example) and use that to install.

You can confirm your installation works by executing:

```
$ wasmtime -V
wasmtime 0.12.0
```

And now you're off to the races! Be sure to check out the various CLI options as well.

Download Precompiled Binaries

If you'd prefer to not use an installation script, or you're perhaps orchestrating something in CI, you can also download one of our precompiled binaries of `wasmtime`. We have two channels of releases right now for precompiled binaries:

1. Each tagged release will have a full set of release artifacts on the GitHub releases page.
2. The `dev` release is also continuously updated with the latest build of the `main` branch. If you want the latest-and-greatest and don't mind a bit of instability, this is the release for you.

When downloading binaries you'll likely want one of the following archives (for the `dev` release)

- Linux users - [`wasmtime-dev-x86_64-linux.tar.xz`]
- macOS users - [`wasmtime-dev-x86_64-macos.tar.xz`]
- Windows users - [`wasmtime-dev-x86_64-windows.zip`]

Each of these archives has a `wasmtime` binary placed inside which can be executed normally as the CLI would.

Compiling from Source

If you'd prefer to compile the `wasmtime` CLI from source, you'll want to consult the contributing documentation for building. Be sure to use a `--release` build if you're curious to do benchmarking!

CLI Options for `wasmtime`

The `wasmtime` CLI is organized into a few subcommands. If no subcommand is provided it'll assume `run`, which is to execute a wasm file. The subcommands supported by `wasmtime` are:

`help`

This is a general subcommand used to print help information to the terminal. You can execute any number of the following:

```
$ wasmtime help
$ wasmtime --help
$ wasmtime -h
$ wasmtime help run
$ wasmtime run -h
```

When in doubt, try running the `help` command to learn more about functionality!

`run`

This is the `wasmtime` CLI's main subcommand, and it's also the default if no other subcommand is provided. The `run` command will execute a WebAssembly module. This means that the module will be compiled to native code, instantiated, and then optionally have an export executed.

The `wasmtime` CLI will automatically hook up any WASI-related imported functionality, but at this time if your module imports anything else it will fail instantiation.

The `run` command takes one positional argument which is the name of the module to run:

```
$ wasmtime run foo.wasm
$ wasmtime foo.wasm
```

Note that the `wasmtime` CLI can take both a binary WebAssembly file (`*.wasm`) as well as the text format for WebAssembly (`*.wat`):

```
$ wasmtime foo.wat
```

The `run` command accepts an optional `invoke` argument which is the name of an exported function of the module to run.

```
$ wasmtime run foo.wasm --invoke initialize
```

serve

The `serve` subcommand runs a WebAssembly component in the `wasi:http/proxy` world via the WASI HTTP API, which is available since Wasmtime 18.0.0. The goal of this world is to support sending and receiving HTTP requests.

The `serve` command takes one positional argument which is the name of the component to run:

```
$ wasmtime serve foo.wasm
```

Furthermore, an address can be specified via:

```
$ wasmtime serve --addr=0.0.0.0:8081 foo.wasm
```

At the time of writing, the `wasi:http/proxy` world is still experimental and requires setup of some `wit` dependencies. For more information, see the hello-wasi-http example.

wast

The `wast` command executes a `*.wast` file which is the test format for the official WebAssembly spec test suite. This subcommand will execute the script file which has a number of directives supported to instantiate modules, link tests, etc.

Executing this looks like:

```
$ wasmtime wast foo.wast
```

config

This subcommand is used to control and edit local Wasmtime configuration settings. The primary purpose of this currently is to configure how Wasmtime's code caching works. You can create a new configuration file for you to edit with:

```
$ wasmtime config new
```

And that'll print out the path to the file you can edit.

`compile`

This subcommand is used to Ahead-Of-Time (AOT) compile a WebAssembly module to produce a "compiled wasm" (.cwasm) file.

The `wasmtime run` subcommand can then be used to run a AOT-compiled WebAssembly module:

```
$ wasmtime compile foo.wasm
$ wasmtime foo.cwasm
```

AOT-compiled modules can be run from hosts that are compatible with the target environment of the AOT-completed module.

`settings`

This subcommand is used to print the available Cranelift settings for a given target.

When run without options, it will print the settings for the host target and also display what Cranelift settings are inferred for the host:

```
$ wasmtime settings
```

Logging in the `wasmtime` CLI

Wasmtime's libraries use Rust's `log` crate to log diagnostic information, and the `wasmtime` CLI executable uses `tracing-subscriber` for displaying this information on the console.

Basic logging is controlled by the `WASMTIME_LOG` environment variable. For example, To enable logging of WASI system calls, similar to the `strace` command on Linux, set `WASMTIME_LOG=wasmtime_wasi=trace`. For more information on specifying filters, see tracing-subscriber's EnvFilter docs.

```
$ WASMTIME_LOG=wasmtime_wasi=trace wasmtime hello.wasm
[...]
TRACE wiggle abi{module="wasi_snapshot_preview1" function="fd_write"}
wasmtime_wasi::preview1::wasi_snapshot_preview1                    > fd=Fd(1)
iovs=*guest 0x14/1
Hello, world!
TRACE wiggle abi{module="wasi_snapshot_preview1" function="fd_write"}:
wasmtime_wasi::preview1::wasi_snapshot_preview1: result=Ok(14)
TRACE wiggle abi{module="wasi_snapshot_preview1" function="proc_exit"}:
wasmtime_wasi::preview1::wasi_snapshot_preview1: rval=1
TRACE wiggle abi{module="wasi_snapshot_preview1" function="proc_exit"}:
wasmtime_wasi::preview1::wasi_snapshot_preview1: result=Exited with i32 exit
status 1
```

Wasmtime can also redirect the log messages into log files, with the `-D log-to-files` option. It creates one file per thread within Wasmtime, with the files named `wasmtime.dbg.*`.

Additional environment variables that work with `WASMTIME_LOG` (**not** `-D log-to-files`):

- `WASMTIME_LOG_NO_CONTEXT`: if set to `1`, removes the time, level and target from output.

Cache Configuration of `wasmtime`

The configuration file uses the toml format. You can create a configuration file at the default location with:

```
$ wasmtime config new
```

It will print the location regardless of the success. Please refer to the `--help` message for using a custom location.

All settings, except `enabled`, are **optional**. If the setting is not specified, the **default** value is used. ***Thus, if you don't know what values to use, don't specify them.*** The default values might be tuned in the future.

Wasmtime assumes all the options are in the `cache` section.

Example config:

```toml
[cache]
enabled = true
directory = "/nfs-share/wasmtime-cache/"
cleanup-interval = "30m"
files-total-size-soft-limit = "1Gi"
```

Please refer to the cache system section to learn how it works.

If you think some default value should be tuned, some new settings should be introduced or some behavior should be changed, you are welcome to discuss it and contribute to the Wasmtime repository.

Setting `enabled`

- **type**: boolean
- **format**: `true | false`
- **default**: `true`

Specifies whether the cache system is used or not.

This field is *mandatory*. The default value is used when configuration file is not specified and none exists at the default location.

Setting `directory`

- **type**: string (path)
- **default**: look up `cache_dir` in directories crate

Specifies where the cache directory is. Must be an absolute path.

Setting `worker-event-queue-size`

- **type**: string (SI prefix)
- **format**: `"{integer}(K | M | G | T | P)?"`
- **default**: `"16"`

Size of cache worker event queue. If the queue is full, incoming cache usage events will be dropped.

Setting `baseline-compression-level`

- **type**: integer
- **default**: `3`, the default zstd compression level

Compression level used when a new cache file is being written by the cache system. Wasmtime uses zstd compression.

Setting `optimized-compression-level`

- **type**: integer
- **default**: `20`

Compression level used when the cache worker decides to recompress a cache file. Wasmtime uses zstd compression.

Setting `optimized-compression-usage-counter-threshold`

- **type**: string (SI prefix)
- **format**: `"{integer}(K | M | G | T | P)?"`

- **default**: `"256"`

One of the conditions for the cache worker to recompress a cache file is to have usage count of the file exceeding this threshold.

Setting `cleanup-interval`

- **type**: string (duration)
- **format**: `"{integer}(s | m | h | d)"`
- **default**: `"1h"`

When the cache worker is notified about a cache file being updated by the cache system and this interval has already passed since last cleaning up, the worker will attempt a new cleanup.

Please also refer to `allowed-clock-drift-for-files-from-future`.

Setting `optimizing-compression-task-timeout`

- **type**: string (duration)
- **format**: `"{integer}(s | m | h | d)"`
- **default**: `"30m"`

When the cache worker decides to recompress a cache file, it makes sure that no other worker has started the task for this file within the last `optimizing-compression-task-timeout` interval. If some worker has started working on it, other workers are skipping this task.

Please also refer to the `allowed-clock-drift-for-files-from-future` section.

Setting `allowed-clock-drift-for-files-from-future`

- **type**: string (duration)
- **format**: `"{integer}(s | m | h | d)"`
- **default**: `"1d"`

Locks

When the cache worker attempts acquiring a lock for some task, it checks if some other worker has already acquired such a lock. To be fault tolerant and eventually execute every task, the locks expire after some interval. However, because of clock drifts and different timezones, it would happen that some lock was created in the future. This setting defines a tolerance limit for these locks. If the time has been changed in the system (i.e. two years backwards), the cache system should still work properly. Thus, these locks will be treated as expired (assuming the tolerance is not too big).

Cache files

Similarly to the locks, the cache files or their metadata might have modification time in distant future. The cache system tries to keep these files as long as possible. If the limits are not reached, the cache files will not be deleted. Otherwise, they will be treated as the oldest files, so they might survive. If the user actually uses the cache file, the modification time will be updated.

Setting `file-count-soft-limit`

- **type**: string (SI prefix)
- **format**: `"{integer}(K | M | G | T | P)?"`
- **default**: `"65536"`

Soft limit for the file count in the cache directory.

This doesn't include files with metadata. To learn more, please refer to the cache system section.

Setting `files-total-size-soft-limit`

- **type**: string (disk space)
- **format**: `"{integer}(K | Ki | M | Mi | G | Gi | T | Ti | P | Pi)?"`
- **default**: `"512Mi"`

Soft limit for the total size* of files in the cache directory.

This doesn't include files with metadata. To learn more, please refer to the cache system section.

*this is the file size, not the space physically occupied on the disk.

Setting `file-count-limit-percent-if-deleting`

- **type**: string (percent)
- **format**: `"{integer}%"`
- **default**: `"70%"`

If `file-count-soft-limit` is exceeded and the cache worker performs the cleanup task, then the worker will delete some cache files, so after the task, the file count should not exceed `file-count-soft-limit` * `file-count-limit-percent-if-deleting`.

This doesn't include files with metadata. To learn more, please refer to the cache system section.

Setting `files-total-size-limit-percent-if-deleting`

- **type**: string (percent)
- **format**: `"{integer}%"`
- **default**: `"70%"`

If `files-total-size-soft-limit` is exceeded and cache worker performs the cleanup task, then the worker will delete some cache files, so after the task, the files total size should not exceed `files-total-size-soft-limit` * `files-total-size-limit-percent-if-deleting`.

This doesn't include files with metadata. To learn more, please refer to the cache system section.

How does the cache work?

This is an implementation detail and might change in the future. Information provided here is meant to help understanding the big picture and configuring the cache.

There are two main components - the *cache system* and the *cache worker*.

Cache system

Handles GET and UPDATE cache requests.

- **GET request** - simply loads the cache from disk if it is there.
- **UPDATE request** - compresses received data with zstd and `baseline-compression-level`, then writes the data to the disk.

In case of successful handling of a request, it notifies the *cache worker* about this event using the queue. The queue has a limited size of `worker-event-queue-size`. If it is full, it will drop new events until the *cache worker* pops some event from the queue.

Cache worker

The cache worker runs in a single thread with lower priority and pops events from the queue in a loop handling them one by one.

On GET request

1. Read the statistics file for the cache file, increase the usage counter and write it back to the disk.

2. Attempt recompressing the cache file if all of the following conditions are met:

    - usage counter exceeds `optimized-compression-usage-counter-threshold`,
    - the file is compressed with compression level lower than `optimized-compression-level`,
    - no other worker has started working on this particular task within the last `optimizing-compression-task-timeout` interval.

    When recompressing, `optimized-compression-level` is used as a compression level.

On UPDATE request

1. Write a fresh statistics file for the cache file.
2. Clean up the cache if no worker has attempted to do this within the last `cleanup-interval`. During this task:
    - all unrecognized files and expired task locks in cache directory will be deleted
    - if `file-count-soft-limit` or `files-total-size-soft-limit` is exceeded, then recognized files will be deleted according to `file-count-limit-percent-if-deleting` and `files-total-size-limit-percent-if-deleting`. Wasmtime uses Least Recently Used (LRU) cache replacement policy and requires that the filesystem maintains proper mtime (modification time) of the files. Files with future mtimes are treated specially - more details in `allowed-clock-drift-for-files-from-future`.

Metadata files

- every cached WebAssembly module has its own statistics file

- every lock is a file

Using the Wasmtime API

Wasmtime can be used as a library to embed WebAssembly execution support within applications. Wasmtime is written in Rust, but bindings are available through a C API for a number of other languages too:

- Rust
- C
- Python
- .NET
- Go
- Bash
- Ruby
- Elixir

Using WebAssembly from Rust

This document shows an example of how to embed Wasmtime using the Rust API to execute a simple wasm program. Be sure to also check out the full API documentation for a full listing of what the `wasmtime` crate has to offer.

Creating the WebAssembly to execute

We'll just assume that you've already got a wasm file on hand for the rest of this tutorial. To make things simple we'll also just assume you've got a `hello.wat` file which looks like this:

```
(module
  (func (export "answer") (result i32)
     i32.const 42
  )
)
```

Here we're just exporting one function which returns an integer that we'll read from Rust.

Hello, World!

First up let's create a rust project

```
$ cargo new --bin wasmtime_hello
$ cd wasmtime_hello
```

Next you'll want to add `hello.wat` to the root of your project.

We will be using the `wasmtime` crate to run the wasm file. Please execute the command `cargo add wasmtime` to use the latest version of the crate. The `dependencies` block in the `Cargo.toml` file will appear as follows:

```
[dependencies]
wasmtime = "19.0.0"
```

Next up let's write the code that we need to execute this wasm file. The simplest version of this looks like so:

```rust
use std::error::Error;
use wasmtime::*;

fn main() -> Result<(), Box<dyn Error>> {
    // An engine stores and configures global compilation settings like
    // optimization level, enabled wasm features, etc.
    let engine = Engine::default();

    // We start off by creating a `Module` which represents a compiled form
    // of our input wasm module. In this case it'll be JIT-compiled after
    // we parse the text format.
    let module = Module::from_file(&engine, "hello.wat")?;

    // A `Store` is what will own instances, functions, globals, etc. All wasm
    // items are stored within a `Store`, and it's what we'll always be using
to
    // interact with the wasm world. Custom data can be stored in stores but
for
    // now we just use `()`.
    let mut store = Store::new(&engine, ());

    // With a compiled `Module` we can then instantiate it, creating
    // an `Instance` which we can actually poke at functions on.
    let instance = Instance::new(&mut store, &module, &[])?;

    // The `Instance` gives us access to various exported functions and items,
    // which we access here to pull out our `answer` exported function and
    // run it.
    let answer = instance.get_func(&mut store, "answer")
        .expect("`answer` was not an exported function");

    // There's a few ways we can call the `answer` `Func` value. The easiest
    // is to statically assert its signature with `typed` (in this case
    // asserting it takes no arguments and returns one i32) and then call it.
    let answer = answer.typed::<(), i32>(&store)?;

    // And finally we can call our function! Note that the error propagation
    // with `?` is done to handle the case where the wasm function traps.
    let result = answer.call(&mut store, ())?;
    println!("Answer: {:?}", result);
    Ok(())
}
```

We can build and execute our example with `cargo run`. Note that by depending on `wasmtime` you're depending on a JIT compiler, so it may take a moment to build all of its dependencies:

```
$ cargo run
  Compiling ...
  ...
   Finished dev [unoptimized + debuginfo] target(s) in 42.32s
    Running `wasmtime_hello/target/debug/wasmtime_hello`
Answer: 42
```

and there we go! We've now executed our first WebAssembly in `wasmtime` and gotten the result back.

Importing Host Functionality

What we've just seen is a pretty small example of how to call a wasm function and take a look at the result. Most interesting wasm modules, however, are going to import some functions to do something a bit more interesting. For that you'll need to provide imported functions from Rust for wasm to call!

Let's take a look at a wasm module which imports a logging function as well as some simple arithmetic from the environment.

```
(module
  (import "" "log" (func $log (param i32)))
  (import "" "double" (func $double (param i32) (result i32)))
  (func (export "run")
    i32.const 0
    call $log
    i32.const 1
    call $log
    i32.const 2
    call $double
    call $log
  )
)
```

This wasm module will call our `"log"` import a few times and then also call the `"double"` import. We can compile and instantiate this module with code that looks like this:

```rust
use std::error::Error;
use wasmtime::*;

struct Log {
    integers_logged: Vec<u32>,
}

fn main() -> Result<(), Box<dyn Error>> {
    let engine = Engine::default();
    let module = Module::from_file(&engine, "hello.wat")?;

    // For host-provided functions it's recommended to use a `Linker` which
does
    // name-based resolution of functions.
    let mut linker = Linker::new(&engine);

    // First we create our simple "double" function which will only multiply
its
    // input by two and return it.
    linker.func_wrap("", "double", |param: i32| param * 2)?;

    // Next we define a `log` function. Note that we're using a
    // Wasmtime-provided `Caller` argument to access the state on the `Store`,
    // which allows us to record the logged information.
    linker.func_wrap("", "log", |mut caller: Caller<'_, Log>, param: u32| {
        println!("log: {}", param);
        caller.data_mut().integers_logged.push(param);
    })?;

    // As above, instantiation always happens within a `Store`. This means to
    // actually instantiate with our `Linker` we'll need to create a store.
Note
    // that we're also initializing the store with our custom data here too.
    //
    // Afterwards we use the `linker` to create the instance.
    let data = Log { integers_logged: Vec::new() };
    let mut store = Store::new(&engine, data);
    let instance = linker.instantiate(&mut store, &module)?;

    // Like before, we can get the run function and execute it.
    let run = instance.get_typed_func::<(), ()>(&mut store, "run")?;
    run.call(&mut store, ())?;

    // We can also inspect what integers were logged:
    println!("logged integers: {:?}", store.data().integers_logged);

    Ok(())
}
```

Note that there's a number of ways to define a `Func`, be sure to consult its documentation for other ways to create a host-defined function.

Hello, world!

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to instantiate a simple wasm module and interact with it. For more information about the types used here be sure to review the core concepts of the `wasmtime` API as well as the general API documentation.

`hello.wat`

```
(module
  (func $hello (import "" "hello"))
  (func (export "run") (call $hello))
)
```

hello.rs

```rust
//! Small example of how to instantiate a wasm module that imports one
function,
//! showing how you can fill in host functionality for a wasm module.

// You can execute this example with `cargo run --example hello`

use wasmtime::*;

struct MyState {
    name: String,
    count: usize,
}

fn main() -> Result<()> {
    // First the wasm module needs to be compiled. This is done with a global
    // "compilation environment" within an `Engine`. Note that engines can be
    // further configured through `Config` if desired instead of using the
    // default like this is here.
    println!("Compiling module...");
    let engine = Engine::default();
    let module = Module::from_file(&engine, "examples/hello.wat")?;

    // After a module is compiled we create a `Store` which will contain
    // instantiated modules and other items like host functions. A Store
    // contains an arbitrary piece of host information, and we use `MyState`
    // here.
    println!("Initializing...");
    let mut store = Store::new(
        &engine,
        MyState {
            name: "hello, world!".to_string(),
            count: 0,
        },
    );

    // Our wasm module we'll be instantiating requires one imported function.
    // the function takes no parameters and returns no results. We create a
host
    // implementation of that function here, and the `caller` parameter here is
    // used to get access to our original `MyState` value.
    println!("Creating callback...");
    let hello_func = Func::wrap(&mut store, |mut caller: Caller<'_, MyState>| {
        println!("Calling back...");
        println!("> {}", caller.data().name);
        caller.data_mut().count += 1;
    });

    // Once we've got that all set up we can then move to the instantiation
    // phase, pairing together a compiled module as well as a set of imports.
    // Note that this is where the wasm `start` function, if any, would run.
    println!("Instantiating module...");
    let imports = [hello_func.into()];
    let instance = Instance::new(&mut store, &module, &imports)?;
```

```
    // Next we poke around a bit to extract the `run` function from the module.
    println!("Extracting export...");
    let run = instance.get_typed_func::<(), ()>(&mut store, "run")?;

    // And last but not least we can call it!
    println!("Calling export...");
    run.call(&mut store, ())?;

    println!("Done.");
    Ok(())
}
```

Calculating the GCD

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how run a wasm program which calculates the GCD of two numbers.

`gcd.wat`

```
(module
  (func $gcd (param i32 i32) (result i32)
    (local i32)
    block  ;; label = @1
      block  ;; label = @2
        local.get 0
        br_if 0 (;@2;)
        local.get 1
        local.set 2
        br 1 (;@1;)
      end
      loop  ;; label = @2
        local.get 1
        local.get 0
        local.tee 2
        i32.rem_u
        local.set 0
        local.get 2
        local.set 1
        local.get 0
        br_if 0 (;@2;)
      end
    end
    local.get 2
  )
  (export "gcd" (func $gcd))
)
```

gcd.rs

```rust
//! Example of instantiating of the WebAssembly module and invoking its
exported
//! function.

// You can execute this example with `cargo run --example gcd`

use wasmtime::*;

fn main() -> Result<()> {
    // Load our WebAssembly (parsed WAT in our case), and then load it into a
    // `Module` which is attached to a `Store` cache. After we've got that we
    // can instantiate it.
    let mut store = Store::<()>::default();
    let module = Module::from_file(store.engine(), "examples/gcd.wat")?;
    let instance = Instance::new(&mut store, &module, &[])?;

    // Invoke `gcd` export
    let gcd = instance.get_typed_func::<(i32, i32), i32>(&mut store, "gcd")?;

    println!("gcd(6, 27) = {}", gcd.call(&mut store, (6, 27))?);
    Ok(())
}
```

Using linear memory

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to interact with wasm memory in a module. Be sure to read the documentation for `Memory` as well.

`memory.wat`

```
(module
  (memory (export "memory") 2 3)

  (func (export "size") (result i32) (memory.size))
  (func (export "load") (param i32) (result i32)
    (i32.load8_s (local.get 0))
  )
  (func (export "store") (param i32 i32)
    (i32.store8 (local.get 0) (local.get 1))
  )

  (data (i32.const 0x1000) "\01\02\03\04")
)
```

memory.rs

```
//! An example of how to interact with wasm memory.
//!
//! Here a small wasm module is used to show how memory is initialized, how to
//! read and write memory through the `Memory` object, and how wasm functions
//! can trap when dealing with out-of-bounds addresses.

// You can execute this example with `cargo run --example memory`

use wasmtime::*;

fn main() -> Result<()> {
    // Create our `store_fn` context and then compile a module and create an
    // instance from the compiled module all in one go.
    let mut store: Store<()> = Store::default();
    let module = Module::from_file(store.engine(), "examples/memory.wat")?;
    let instance = Instance::new(&mut store, &module, &[])?;

    // load_fn up our exports from the instance
    let memory = instance
        .get_memory(&mut store, "memory")
        .ok_or(anyhow::format_err!("failed to find `memory` export"))?;
    let size = instance.get_typed_func::<(), i32>(&mut store, "size")?;
    let load_fn = instance.get_typed_func::<i32, i32>(&mut store, "load")?;
    let store_fn = instance.get_typed_func::<(i32, i32), ()>(&mut store,
"store")?;

    println!("Checking memory...");
    assert_eq!(memory.size(&store), 2);
    assert_eq!(memory.data_size(&store), 0x20000);
    assert_eq!(memory.data_mut(&mut store)[0], 0);
    assert_eq!(memory.data_mut(&mut store)[0x1000], 1);
    assert_eq!(memory.data_mut(&mut store)[0x1003], 4);

    assert_eq!(size.call(&mut store, ())?, 2);
    assert_eq!(load_fn.call(&mut store, 0)?, 0);
    assert_eq!(load_fn.call(&mut store, 0x1000)?, 1);
    assert_eq!(load_fn.call(&mut store, 0x1003)?, 4);
    assert_eq!(load_fn.call(&mut store, 0x1ffff)?, 0);
    assert!(load_fn.call(&mut store, 0x20000).is_err()); // out of bounds trap

    println!("Mutating memory...");
    memory.data_mut(&mut store)[0x1003] = 5;

    store_fn.call(&mut store, (0x1002, 6))?;
    assert!(store_fn.call(&mut store, (0x20000, 0)).is_err()); // out of bounds
trap

    assert_eq!(memory.data(&store)[0x1002], 6);
    assert_eq!(memory.data(&store)[0x1003], 5);
    assert_eq!(load_fn.call(&mut store, 0x1002)?, 6);
    assert_eq!(load_fn.call(&mut store, 0x1003)?, 5);

    // Grow memory.
    println!("Growing memory...");
```

```
    memory.grow(&mut store, 1)?;
    assert_eq!(memory.size(&store), 3);
    assert_eq!(memory.data_size(&store), 0x30000);

    assert_eq!(load_fn.call(&mut store, 0x20000)?, 0);
    store_fn.call(&mut store, (0x20000, 0))?;
    assert!(load_fn.call(&mut store, 0x30000).is_err());
    assert!(store_fn.call(&mut store, (0x30000, 0)).is_err());

    assert!(memory.grow(&mut store, 1).is_err());
    assert!(memory.grow(&mut store, 0).is_ok());

    println!("Creating stand-alone memory...");
    let memorytype = MemoryType::new(5, Some(5));
    let memory2 = Memory::new(&mut store, memorytype)?;
    assert_eq!(memory2.size(&store), 5);
    assert!(memory2.grow(&mut store, 1).is_err());
    assert!(memory2.grow(&mut store, 0).is_ok());

    Ok(())
}
```

WASI

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows how to use the `wasi-common` crate to define WASI functions within a `Linker` which can then be used to instantiate a WebAssembly module.

WebAssembly module source code

For this WASI example, this Hello World program is compiled to a WebAssembly module using the WASI Preview 1 API.

`wasi.rs`

```
fn main() {
    println!("Hello, world!");
}
```

Building this program generates `target/wasm32-wasip1/debug/wasi.wasm`, used below.

Invoke the WASM module

This example shows adding and configuring the WASI imports to invoke the above WASM module.

`main.rs`

```rust
//! Example of instantiating a wasm module which uses WASI imports.

/*
You can execute this example with:
    cmake examples/
    cargo run --example wasi
*/

use wasi_common::sync::WasiCtxBuilder;
use wasmtime::*;

fn main() -> Result<()> {
    // Define the WASI functions globally on the `Config`.
    let engine = Engine::default();
    let mut linker = Linker::new(&engine);
    wasi_common::sync::add_to_linker(&mut linker, |s| s)?;

    // Create a WASI context and put it in a Store; all instances in the store
    // share this context. `WasiCtxBuilder` provides a number of ways to
    // configure what the target program will have access to.
    let wasi = WasiCtxBuilder::new()
        .inherit_stdio()
        .inherit_args()?
        .build();
    let mut store = Store::new(&engine, wasi);

    // Instantiate our module with the imports we've created, and run it.
    let module = Module::from_file(&engine, "target/wasm32-
wasip1/debug/wasi.wasm")?;
    linker.module(&mut store, "", &module)?;
    linker
        .get_default(&mut store, "")?
        .typed::<(), ()>(&store)?
        .call(&mut store, ())?;

    Ok(())
}
```

WASI state with other custom host state

The `add_to_linker` takes a second argument which is a closure to access `&mut WasiCtx` from within the `T` stored in the `Store<T>` itself. In the above example this is trivial because the `T` in `Store<T>` is `WasiCtx` itself, but you can also store other state in `Store` like so:

```rust
use anyhow::Result;
use std::borrow::{Borrow, BorrowMut};
use wasmtime::*;
use wasi_common::{WasiCtx, sync::WasiCtxBuilder};

struct MyState {
    message: String,
    wasi: WasiCtx,
}

fn main() -> Result<()> {
    let engine = Engine::default();
    let mut linker = Linker::new(&engine);
    wasi_common::sync::add_to_linker(&mut linker, |state: &mut MyState| &mut
state.wasi)?;

    let wasi = WasiCtxBuilder::new()
        .inherit_stdio()
        .inherit_args()?
        .build();
    let mut store = Store::new(&engine, MyState {
        message: format!("hello!"),
        wasi,
    });

    // ...

    Ok(())
}
```

WASI Preview 2

An experimental implementation of the WASI Preview 2 API is also available, along with an adapter layer for WASI Preview 1 WebAssembly modules. In future this `preview2` API will become the default. There are some features which are currently only accessible through the `preview2` API such as async support and overriding the clock and random implementations.

Async example

This async example code shows how to use the wasmtime-wasi::preview2 module to execute the same WASI Preview 1 WebAssembly module from the example above. This example requires the `wasmtime` crate `async` feature to be enabled.

This does not require any change to the WebAssembly module, it's just the WASI API host functions which are implemented to be async. See wasmtime async support.

```rust
//! Example of instantiating a wasm module which uses WASI preview1 imports
//! implemented through the async preview2 WASI implementation.

/*
You can execute this example with:
    cmake examples/
    cargo run --example wasi-async
*/

use anyhow::Result;
use wasmtime::{Config, Engine, Linker, Module, Store};
use wasmtime_wasi::preview1::{self, WasiP1Ctx};
use wasmtime_wasi::WasiCtxBuilder;

#[tokio::main]
async fn main() -> Result<()> {
    // Construct the wasm engine with async support enabled.
    let mut config = Config::new();
    config.async_support(true);
    let engine = Engine::new(&config)?;

    // Add the WASI preview1 API to the linker (will be implemented in terms of
    // the preview2 API)
    let mut linker: Linker<WasiP1Ctx> = Linker::new(&engine);
    preview1::add_to_linker_async(&mut linker, |t| t)?;

    // Add capabilities (e.g. filesystem access) to the WASI preview2 context
    // here. Here only stdio is inherited, but see docs of `WasiCtxBuilder` for
    // more.
    let wasi_ctx = WasiCtxBuilder::new().inherit_stdio().build_p1();

    let mut store = Store::new(&engine, wasi_ctx);

    // Instantiate our 'Hello World' wasm module.
    // Note: This is a module built against the preview1 WASI API.
    let module = Module::from_file(&engine, "target/wasm32-
wasip1/debug/wasi.wasm")?;
    let func = linker
        .module_async(&mut store, "", &module)
        .await?
        .get_default(&mut store, "")?
        .typed::<(), ()>(&store)?;

    // Invoke the WASI program default function.
    func.call_async(&mut store, ()).await?;

    Ok(())
}
```

You can also browse this source code online and clone the wasmtime repository to run the example locally.

Linking modules

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to compile and instantiate modules which link together. Be sure to read the API documentation for `Linker` as well.

`linking1.wat`

```
(module
  (import "linking2" "double" (func $double (param i32) (result i32)))
  (import "linking2" "log" (func $log (param i32 i32)))
  (import "linking2" "memory" (memory 1))
  (import "linking2" "memory_offset" (global $offset i32))

  (func (export "run")
    ;; Call into the other module to double our number, and we could print it
    ;; here but for now we just drop it
    i32.const 2
    call $double
    drop

    ;; Our `data` segment initialized our imported memory, so let's print the
    ;; string there now.
    global.get $offset
    i32.const 14
    call $log
  )

  (data (global.get $offset) "Hello, world!\n")
)
```

linking2.wat

```
(module
  (type $fd_write_ty (func (param i32 i32 i32 i32) (result i32)))
  (import "wasi_snapshot_preview1" "fd_write" (func $fd_write (type
$fd_write_ty)))

  (func (export "double") (param i32) (result i32)
    local.get 0
    i32.const 2
    i32.mul
  )

  (func (export "log") (param i32 i32)
    ;; store the pointer in the first iovec field
    i32.const 4
    local.get 0
    i32.store

    ;; store the length in the first iovec field
    i32.const 4
    local.get 1
    i32.store offset=4

    ;; call the `fd_write` import
    i32.const 1     ;; stdout fd
    i32.const 4     ;; iovs start
    i32.const 1     ;; number of iovs
    i32.const 0     ;; where to write nwritten bytes
    call $fd_write
    drop
  )

  (memory (export "memory") 2)
  (global (export "memory_offset") i32 (i32.const 65536))
)
```

linking.rs

```rust
//! Example of instantiating two modules which link to each other.

// You can execute this example with `cargo run --example linking`

use wasi_common::sync::WasiCtxBuilder;
use wasmtime::*;

fn main() -> Result<()> {
    let engine = Engine::default();

    // First set up our linker which is going to be linking modules together.
We
    // want our linker to have wasi available, so we set that up here as well.
    let mut linker = Linker::new(&engine);
    wasi_common::sync::add_to_linker(&mut linker, |s| s)?;

    // Load and compile our two modules
    let linking1 = Module::from_file(&engine, "examples/linking1.wat")?;
    let linking2 = Module::from_file(&engine, "examples/linking2.wat")?;

    // Configure WASI and insert it into a `Store`
    let wasi = WasiCtxBuilder::new()
        .inherit_stdio()
        .inherit_args()?
        .build();
    let mut store = Store::new(&engine, wasi);

    // Instantiate our first module which only uses WASI, then register that
    // instance with the linker since the next linking will use it.
    let linking2 = linker.instantiate(&mut store, &linking2)?;
    linker.instance(&mut store, "linking2", linking2)?;

    // And with that we can perform the final link and the execute the module.
    let linking1 = linker.instantiate(&mut store, &linking1)?;
    let run = linking1.get_typed_func::<(), ()>(&mut store, "run")?;
    run.call(&mut store, ())?;
    Ok(())
}
```

Debugging

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to set up a module for dynamic runtime debugging via a native debugger like GDB or LLDB.

`main.rs`

```
//! Example of enabling debuginfo for wasm code which allows interactive
//! debugging of the wasm code. When using recent versions of LLDB
//! you can debug this executable and set breakpoints in wasm code and look at
//! the rust source code as input.

// To execute this example you'll need to run two commands:
//
//      cargo build -p example-fib-debug-wasm --target wasm32-unknown-unknown
//      cargo run --example fib-debug

use wasmtime::*;

fn main() -> Result<()> {
    // Load our previously compiled wasm file (built previously with Cargo) and
    // also ensure that we generate debuginfo so this executable can be
    // debugged in GDB.
    let engine = Engine::new(
        Config::new()
            .debug_info(true)
            .cranelift_opt_level(OptLevel::None),
    )?;
    let mut store = Store::new(&engine, ());
    let module = Module::from_file(&engine, "target/wasm32-unknown-
unknown/debug/fib.wasm")?;
    let instance = Instance::new(&mut store, &module, &[])?;

    // Invoke `fib` export
    let fib = instance.get_typed_func::<i32, i32>(&mut store, "fib")?;
    println!("fib(6) = {}", fib.call(&mut store, 6)?);
    Ok(())
}
```

Core Dumps

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This examples shows how to configure capturing core dumps when a Wasm guest traps that can then be passed to external tools (like `wasmgdb`) for post-mortem analysis.

main.rs

```
//! An example of how to configure capturing core dumps when the guest Wasm
//! traps that can then be passed to external tools for post-mortem analysis.

// You can execute this example with `cargo run --example coredump`.

use wasmtime::*;

fn main() -> Result<()> {
    println!("Configure core dumps to be captured on trap.");
    let mut config = Config::new();
    config.coredump_on_trap(true);
    let engine = Engine::new(&config)?;
    let mut store = Store::new(&engine, ());

    println!("Define a Wasm module that will mutate local state and then
trap.");
    let module = Module::new(
        store.engine(),
        r#"
            (module $trapper
                (memory 10)
                (global $g (mut i32) (i32.const 0))

                (func (export "run")
                    call $a
                )

                (func $a
                    i32.const 0x1234
                    i64.const 42
                    i64.store
                    call $b
                )

                (func $b
                    i32.const 36
                    global.set $g
                    call $c
                )

                (func $c
                    unreachable
                )
            )
        "#,
    )?;

    println!("Instantiate the module.");
    let instance = Instance::new(&mut store, &module, &[])?;

    println!("Invoke its 'run' function.");
    let run = instance
        .get_func(&mut store, "run")
        .expect("should have 'run' export");
```

```
    let args = &[];
    let results = &mut [];
    let ok = run.call(&mut store, args, results);

    println!("Calling that function trapped.");
    assert!(ok.is_err());
    let err = ok.unwrap_err();
    assert!(err.is::<Trap>());

    println!("Extract the captured core dump.");
    let dump = err
        .downcast_ref::<WasmCoreDump>()
        .expect("should have an attached core dump, since we configured core
dumps on");

    println!(
        "Number of memories in the core dump: {}",
        dump.memories().len()
    );
    for (i, mem) in dump.memories().iter().enumerate() {
        if let Some(addr) = mem.data(&store).iter().position(|byte| *byte != 0)
{
            let val = mem.data(&store)[addr];
            println!("  First nonzero byte for memory {i}: {val} @ {addr:#x}");
        } else {
            println!("  Memory {i} is all zeroes.");
        }
    }

    println!(
        "Number of globals in the core dump: {}",
        dump.globals().len()
    );
    for (i, global) in dump.globals().iter().enumerate() {
        let val = global.get(&mut store);
        println!("  Global {i} = {val:?}");
    }

    println!("Serialize the core dump and write it to ./example.coredump");
    let serialized = dump.serialize(&mut store, "trapper.wasm");
    std::fs::write("./example.coredump", serialized)?;

    Ok(())
}
```

Using multi-value

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to interact with a wasm module that uses multi-value exports and imports.

`multi.wat`

```
(module
  (func $f (import "" "f") (param i32 i64) (result i64 i32))

  (func $g (export "g") (param i32 i64) (result i64 i32)
    (call $f (local.get 0) (local.get 1))
  )

  (func $round_trip_many
    (export "round_trip_many")
    (param i64 i64 i64 i64 i64 i64 i64 i64 i64 i64)
    (result i64 i64 i64 i64 i64 i64 i64 i64 i64 i64)

    local.get 0
    local.get 1
    local.get 2
    local.get 3
    local.get 4
    local.get 5
    local.get 6
    local.get 7
    local.get 8
    local.get 9)
)
```

multi.rs

```
//! This is an example of working with multi-value modules and dealing with
//! multi-value functions.
//!
//! Note that the `Func::wrap*` interfaces cannot be used to return multiple
//! values just yet, so we need to use the more dynamic `Func::new` and
//! `Func::call` methods.

// You can execute this example with `cargo run --example multi`

use anyhow::Result;

fn main() -> Result<()> {
    use wasmtime::*;

    println!("Initializing...");
    let engine = Engine::default();
    let mut store = Store::new(&engine, ());

    // Compile.
    println!("Compiling module...");
    let module = Module::from_file(&engine, "examples/multi.wat")?;

    // Create a host function which takes multiple parameters and returns
    // multiple results.
    println!("Creating callback...");
    let callback_func = Func::wrap(&mut store, |a: i32, b: i64| -> (i64, i32) {
        (b + 1, a + 1)
    });

    // Instantiate.
    println!("Instantiating module...");
    let instance = Instance::new(&mut store, &module, &
[callback_func.into()])?;

    // Extract exports.
    println!("Extracting export...");
    let g = instance.get_typed_func::<(i32, i64), (i64, i32)>(&mut store,
"g")?;

    // Call `$g`.
    println!("Calling export \"g\"...");
    let (a, b) = g.call(&mut store, (1, 3))?;

    println!("Printing result...");
    println!("> {a} {b}");

    assert_eq!(a, 4);
    assert_eq!(b, 2);

    // Call `$round_trip_many`.
    println!("Calling export \"round_trip_many\"...");
    let round_trip_many = instance
        .get_typed_func::<
        (i64, i64, i64, i64, i64, i64, i64, i64, i64, i64),
```

```
        (i64, i64, i64, i64, i64, i64, i64, i64, i64, i64),
        >
        (&mut store, "round_trip_many")?;
    let results = round_trip_many.call(&mut store, (0, 1, 2, 3, 4, 5, 6, 7, 8,
9))?;

    println!("Printing result...");
    println!("> {results:?}");
    assert_eq!(results, (0, 1, 2, 3, 4, 5, 6, 7, 8, 9));

    Ok(())
}
```

Embedding in C

This section is intended to showcase the C embedding API for Wasmtime. Full reference documentation for the C API can be found online.

Hello, world!

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to instantiate a simple wasm module and interact with it.

`hello.wat`

```
(module
  (func $hello (import "" "hello"))
  (func (export "run") (call $hello))
)
```

hello.c

```
/*
Example of instantiating of the WebAssembly module and invoking its exported
function.

You can compile and run this example on Linux with:

   cargo build --release -p wasmtime-c-api
   cc examples/hello.c \
       -I crates/c-api/include \
       target/release/libwasmtime.a \
       -lpthread -ldl -lm \
       -o hello
   ./hello

Note that on Windows and macOS the command will be similar, but you'll need
to tweak the `-lpthread` and such annotations as well as the name of the
`libwasmtime.a` file on Windows.

You can also build using cmake:

mkdir build && cd build && cmake .. && cmake --build . --target wasmtime-hello
*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <wasm.h>
#include <wasmtime.h>

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap);

static wasm_trap_t *hello_callback(void *env, wasmtime_caller_t *caller,
                                   const wasmtime_val_t *args, size_t nargs,
                                   wasmtime_val_t *results, size_t nresults) {
  printf("Calling back...\n");
  printf("> Hello World!\n");
  return NULL;
}

int main() {
  int ret = 0;
  // Set up our compilation context. Note that we could also work with a
  // `wasm_config_t` here to configure what feature are enabled and various
  // compilation settings.
  printf("Initializing...\n");
  wasm_engine_t *engine = wasm_engine_new();
  assert(engine != NULL);

  // With an engine we can create a *store* which is a long-lived group of wasm
  // modules. Note that we allocate some custom data here to live in the store,
  // but here we skip that and specify NULL.
  wasmtime_store_t *store = wasmtime_store_new(engine, NULL, NULL);
  assert(store != NULL);
```

```c
  wasmtime_context_t *context = wasmtime_store_context(store);

  // Read our input file, which in this case is a wasm text file.
  FILE *file = fopen("examples/hello.wat", "r");
  assert(file != NULL);
  fseek(file, 0L, SEEK_END);
  size_t file_size = ftell(file);
  fseek(file, 0L, SEEK_SET);
  wasm_byte_vec_t wat;
  wasm_byte_vec_new_uninitialized(&wat, file_size);
  if (fread(wat.data, file_size, 1, file) != 1) {
    printf("> Error loading module!\n");
    return 1;
  }
  fclose(file);

  // Parse the wat into the binary wasm format
  wasm_byte_vec_t wasm;
  wasmtime_error_t *error = wasmtime_wat2wasm(wat.data, wat.size, &wasm);
  if (error != NULL)
    exit_with_error("failed to parse wat", error, NULL);
  wasm_byte_vec_delete(&wat);

  // Now that we've got our binary webassembly we can compile our module.
  printf("Compiling module...\n");
  wasmtime_module_t *module = NULL;
  error = wasmtime_module_new(engine, (uint8_t *)wasm.data, wasm.size,
&module);
  wasm_byte_vec_delete(&wasm);
  if (error != NULL)
    exit_with_error("failed to compile module", error, NULL);

  // Next up we need to create the function that the wasm module imports. Here
  // we'll be hooking up a thunk function to the `hello_callback` native
  // function above. Note that we can assign custom data, but we just use NULL
  // for now).
  printf("Creating callback...\n");
  wasm_functype_t *hello_ty = wasm_functype_new_0_0();
  wasmtime_func_t hello;
  wasmtime_func_new(context, hello_ty, hello_callback, NULL, NULL, &hello);

  // With our callback function we can now instantiate the compiled module,
  // giving us an instance we can then execute exports from. Note that
  // instantiation can trap due to execution of the `start` function, so we
need
  // to handle that here too.
  printf("Instantiating module...\n");
  wasm_trap_t *trap = NULL;
  wasmtime_instance_t instance;
  wasmtime_extern_t import;
  import.kind = WASMTIME_EXTERN_FUNC;
  import.of.func = hello;
  error = wasmtime_instance_new(context, module, &import, 1, &instance, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to instantiate", error, trap);

  // Lookup our `run` export function
  printf("Extracting export...\n");
```

```
  wasmtime_extern_t run;
  bool ok = wasmtime_instance_export_get(context, &instance, "run", 3, &run);
  assert(ok);
  assert(run.kind == WASMTIME_EXTERN_FUNC);

  // And call it!
  printf("Calling export...\n");
  error = wasmtime_func_call(context, &run.of.func, NULL, 0, NULL, 0, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to call function", error, trap);

  // Clean up after ourselves at this point
  printf("All finished!\n");
  ret = 0;

  wasmtime_module_delete(module);
  wasmtime_store_delete(store);
  wasm_engine_delete(engine);
  return ret;
}

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap) {
  fprintf(stderr, "error: %s\n", message);
  wasm_byte_vec_t error_message;
  if (error != NULL) {
    wasmtime_error_message(error, &error_message);
    wasmtime_error_delete(error);
  } else {
    wasm_trap_message(trap, &error_message);
    wasm_trap_delete(trap);
  }
  fprintf(stderr, "%.*s\n", (int)error_message.size, error_message.data);
  wasm_byte_vec_delete(&error_message);
  exit(1);
}
```

Calculating the GCD

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how run a wasm program which calculates the GCD of two numbers.

`gcd.wat`

```
(module
  (func $gcd (param i32 i32) (result i32)
    (local i32)
    block  ;; label = @1
      block  ;; label = @2
        local.get 0
        br_if 0 (;@2;)
        local.get 1
        local.set 2
        br 1 (;@1;)
      end
      loop  ;; label = @2
        local.get 1
        local.get 0
        local.tee 2
        i32.rem_u
        local.set 0
        local.get 2
        local.set 1
        local.get 0
        br_if 0 (;@2;)
      end
    end
    local.get 2
  )
  (export "gcd" (func $gcd))
)
```

gcd.c

```
/*
Example of instantiating of the WebAssembly module and invoking its exported
function.

You can compile and run this example on Linux with:

   cargo build --release -p wasmtime-c-api
   cc examples/gcd.c \
       -I crates/c-api/include \
       target/release/libwasmtime.a \
       -lpthread -ldl -lm \
       -o gcd
   ./gcd

Note that on Windows and macOS the command will be similar, but you'll need
to tweak the `-lpthread` and such annotations.

You can also build using cmake:

mkdir build && cd build && cmake .. && cmake --build . --target wasmtime-gcd
*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <wasm.h>
#include <wasmtime.h>

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap);

int main() {
  int ret = 0;
  // Set up our context
  wasm_engine_t *engine = wasm_engine_new();
  assert(engine != NULL);
  wasmtime_store_t *store = wasmtime_store_new(engine, NULL, NULL);
  assert(store != NULL);
  wasmtime_context_t *context = wasmtime_store_context(store);

  // Load our input file to parse it next
  FILE *file = fopen("examples/gcd.wat", "r");
  if (!file) {
    printf("> Error loading file!\n");
    return 1;
  }
  fseek(file, 0L, SEEK_END);
  size_t file_size = ftell(file);
  fseek(file, 0L, SEEK_SET);
  wasm_byte_vec_t wat;
  wasm_byte_vec_new_uninitialized(&wat, file_size);
  if (fread(wat.data, file_size, 1, file) != 1) {
    printf("> Error loading module!\n");
    return 1;
```

```
  }
  fclose(file);

  // Parse the wat into the binary wasm format
  wasm_byte_vec_t wasm;
  wasmtime_error_t *error = wasmtime_wat2wasm(wat.data, wat.size, &wasm);
  if (error != NULL)
    exit_with_error("failed to parse wat", error, NULL);
  wasm_byte_vec_delete(&wat);

  // Compile and instantiate our module
  wasmtime_module_t *module = NULL;
  error = wasmtime_module_new(engine, (uint8_t *)wasm.data, wasm.size,
&module);
  if (module == NULL)
    exit_with_error("failed to compile module", error, NULL);
  wasm_byte_vec_delete(&wasm);

  wasm_trap_t *trap = NULL;
  wasmtime_instance_t instance;
  error = wasmtime_instance_new(context, module, NULL, 0, &instance, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to instantiate", error, trap);

  // Lookup our `gcd` export function
  wasmtime_extern_t gcd;
  bool ok = wasmtime_instance_export_get(context, &instance, "gcd", 3, &gcd);
  assert(ok);
  assert(gcd.kind == WASMTIME_EXTERN_FUNC);

  // And call it!
  int a = 6;
  int b = 27;
  wasmtime_val_t params[2];
  params[0].kind = WASMTIME_I32;
  params[0].of.i32 = a;
  params[1].kind = WASMTIME_I32;
  params[1].of.i32 = b;
  wasmtime_val_t results[1];
  error =
      wasmtime_func_call(context, &gcd.of.func, params, 2, results, 1, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to call gcd", error, trap);
  assert(results[0].kind == WASMTIME_I32);

  printf("gcd(%d, %d) = %d\n", a, b, results[0].of.i32);

  // Clean up after ourselves at this point
  ret = 0;

  wasmtime_module_delete(module);
  wasmtime_store_delete(store);
  wasm_engine_delete(engine);
  return ret;
}

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap) {
```

```
  fprintf(stderr, "error: %s\n", message);
  wasm_byte_vec_t error_message;
  if (error != NULL) {
    wasmtime_error_message(error, &error_message);
  } else {
    wasm_trap_message(trap, &error_message);
  }
  fprintf(stderr, "%.*s\n", (int)error_message.size, error_message.data);
  wasm_byte_vec_delete(&error_message);
  exit(1);
}
```

Using linear memory

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to interact with wasm memory in a module. Be sure to read the documentation for `Memory` as well.

`memory.wat`

```
(module
  (memory (export "memory") 2 3)

  (func (export "size") (result i32) (memory.size))
  (func (export "load") (param i32) (result i32)
    (i32.load8_s (local.get 0))
  )
  (func (export "store") (param i32 i32)
    (i32.store8 (local.get 0) (local.get 1))
  )

  (data (i32.const 0x1000) "\01\02\03\04")
)
```

memory.c

```
/*
Example of instantiating of the WebAssembly module and invoking its exported
function.

You can compile and run this example on Linux with:

   cargo build --release -p wasmtime-c-api
   cc examples/memory.c \
       -I crates/c-api/include \
       target/release/libwasmtime.a \
       -lpthread -ldl -lm \
       -o memory
   ./memory

Note that on Windows and macOS the command will be similar, but you'll need
to tweak the `-lpthread` and such annotations.

You can also build using cmake:

mkdir build && cd build && cmake .. && cmake --build . --target wasmtime-memory

Also note that this example was taken from
https://github.com/WebAssembly/wasm-c-api/blob/master/example/memory.c
originally
*/

#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wasm.h>
#include <wasmtime.h>

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap);

void check(bool success) {
  if (!success) {
    printf("> Error, expected success\n");
    exit(1);
  }
}

void check_call(wasmtime_context_t *store, wasmtime_func_t *func,
                const wasmtime_val_t *args, size_t nargs, int32_t expected) {
  wasmtime_val_t results[1];
  wasm_trap_t *trap = NULL;
  wasmtime_error_t *error =
      wasmtime_func_call(store, func, args, nargs, results, 1, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to call function", error, trap);
  if (results[0].of.i32 != expected) {
    printf("> Error on result\n");
    exit(1);
```

```
  }
}

void check_call0(wasmtime_context_t *store, wasmtime_func_t *func,
                 int32_t expected) {
  check_call(store, func, NULL, 0, expected);
}

void check_call1(wasmtime_context_t *store, wasmtime_func_t *func, int32_t arg,
                 int32_t expected) {
  wasmtime_val_t args[1];
  args[0].kind = WASMTIME_I32;
  args[0].of.i32 = arg;
  check_call(store, func, args, 1, expected);
}

void check_call2(wasmtime_context_t *store, wasmtime_func_t *func, int32_t
arg1,
                 int32_t arg2, int32_t expected) {
  wasmtime_val_t args[2];
  args[0].kind = WASMTIME_I32;
  args[0].of.i32 = arg1;
  args[1].kind = WASMTIME_I32;
  args[1].of.i32 = arg2;
  check_call(store, func, args, 2, expected);
}

void check_ok(wasmtime_context_t *store, wasmtime_func_t *func,
              const wasmtime_val_t *args, size_t nargs) {
  wasm_trap_t *trap = NULL;
  wasmtime_error_t *error =
      wasmtime_func_call(store, func, args, nargs, NULL, 0, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to call function", error, trap);
}

void check_ok2(wasmtime_context_t *store, wasmtime_func_t *func, int32_t arg1,
               int32_t arg2) {
  wasmtime_val_t args[2];
  args[0].kind = WASMTIME_I32;
  args[0].of.i32 = arg1;
  args[1].kind = WASMTIME_I32;
  args[1].of.i32 = arg2;
  check_ok(store, func, args, 2);
}

void check_trap(wasmtime_context_t *store, wasmtime_func_t *func,
                const wasmtime_val_t *args, size_t nargs, size_t num_results) {
  assert(num_results <= 1);
  wasmtime_val_t results[1];
  wasm_trap_t *trap = NULL;
  wasmtime_error_t *error =
      wasmtime_func_call(store, func, args, nargs, results, num_results,
&trap);
  if (error != NULL)
    exit_with_error("failed to call function", error, NULL);
  if (trap == NULL) {
    printf("> Error on result, expected trap\n");
```

```
    exit(1);
  }
  wasm_trap_delete(trap);
}

void check_trap1(wasmtime_context_t *store, wasmtime_func_t *func,
                 int32_t arg) {
  wasmtime_val_t args[1];
  args[0].kind = WASMTIME_I32;
  args[0].of.i32 = arg;
  check_trap(store, func, args, 1, 1);
}

void check_trap2(wasmtime_context_t *store, wasmtime_func_t *func, int32_t
arg1,
                 int32_t arg2) {
  wasmtime_val_t args[2];
  args[0].kind = WASMTIME_I32;
  args[0].of.i32 = arg1;
  args[1].kind = WASMTIME_I32;
  args[1].of.i32 = arg2;
  check_trap(store, func, args, 2, 0);
}

int main(int argc, const char *argv[]) {
  // Initialize.
  printf("Initializing...\n");
  wasm_engine_t *engine = wasm_engine_new();
  wasmtime_store_t *store = wasmtime_store_new(engine, NULL, NULL);
  wasmtime_context_t *context = wasmtime_store_context(store);

  // Load our input file to parse it next
  FILE *file = fopen("examples/memory.wat", "r");
  if (!file) {
    printf("> Error loading file!\n");
    return 1;
  }
  fseek(file, 0L, SEEK_END);
  size_t file_size = ftell(file);
  fseek(file, 0L, SEEK_SET);
  wasm_byte_vec_t wat;
  wasm_byte_vec_new_uninitialized(&wat, file_size);
  if (fread(wat.data, file_size, 1, file) != 1) {
    printf("> Error loading module!\n");
    return 1;
  }
  fclose(file);

  // Parse the wat into the binary wasm format
  wasm_byte_vec_t binary;
  wasmtime_error_t *error = wasmtime_wat2wasm(wat.data, wat.size, &binary);
  if (error != NULL)
    exit_with_error("failed to parse wat", error, NULL);
  wasm_byte_vec_delete(&wat);

  // Compile.
  printf("Compiling module...\n");
  wasmtime_module_t *module = NULL;
```

```
  error =
      wasmtime_module_new(engine, (uint8_t *)binary.data, binary.size,
&module);
  if (error)
    exit_with_error("failed to compile module", error, NULL);
  wasm_byte_vec_delete(&binary);

  // Instantiate.
  printf("Instantiating module...\n");
  wasmtime_instance_t instance;
  wasm_trap_t *trap = NULL;
  error = wasmtime_instance_new(context, module, NULL, 0, &instance, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to instantiate", error, trap);
  wasmtime_module_delete(module);

  // Extract export.
  printf("Extracting exports...\n");
  wasmtime_memory_t memory;
  wasmtime_func_t size_func, load_func, store_func;
  wasmtime_extern_t item;
  bool ok;
  ok = wasmtime_instance_export_get(context, &instance, "memory",
                                    strlen("memory"), &item);
  assert(ok && item.kind == WASMTIME_EXTERN_MEMORY);
  memory = item.of.memory;
  ok = wasmtime_instance_export_get(context, &instance, "size", strlen("size"),
                                    &item);
  assert(ok && item.kind == WASMTIME_EXTERN_FUNC);
  size_func = item.of.func;
  ok = wasmtime_instance_export_get(context, &instance, "load", strlen("load"),
                                    &item);
  assert(ok && item.kind == WASMTIME_EXTERN_FUNC);
  load_func = item.of.func;
  ok = wasmtime_instance_export_get(context, &instance, "store",
                                    strlen("store"), &item);
  assert(ok && item.kind == WASMTIME_EXTERN_FUNC);
  store_func = item.of.func;

  // Check initial memory.
  printf("Checking memory...\n");
  check(wasmtime_memory_size(context, &memory) == 2);
  check(wasmtime_memory_data_size(context, &memory) == 0x20000);
  check(wasmtime_memory_data(context, &memory)[0] == 0);
  check(wasmtime_memory_data(context, &memory)[0x1000] == 1);
  check(wasmtime_memory_data(context, &memory)[0x1003] == 4);

  check_call0(context, &size_func, 2);
  check_call1(context, &load_func, 0, 0);
  check_call1(context, &load_func, 0x1000, 1);
  check_call1(context, &load_func, 0x1003, 4);
  check_call1(context, &load_func, 0x1ffff, 0);
  check_trap1(context, &load_func, 0x20000);

  // Mutate memory.
  printf("Mutating memory...\n");
  wasmtime_memory_data(context, &memory)[0x1003] = 5;
  check_ok2(context, &store_func, 0x1002, 6);
```

```
  check_trap2(context, &store_func, 0x20000, 0);

  check(wasmtime_memory_data(context, &memory)[0x1002] == 6);
  check(wasmtime_memory_data(context, &memory)[0x1003] == 5);
  check_call1(context, &load_func, 0x1002, 6);
  check_call1(context, &load_func, 0x1003, 5);

  // Grow memory.
  printf("Growing memory...\n");
  uint64_t old_size;
  error = wasmtime_memory_grow(context, &memory, 1, &old_size);
  if (error != NULL)
    exit_with_error("failed to grow memory", error, trap);
  check(wasmtime_memory_size(context, &memory) == 3);
  check(wasmtime_memory_data_size(context, &memory) == 0x30000);

  check_call1(context, &load_func, 0x20000, 0);
  check_ok2(context, &store_func, 0x20000, 0);
  check_trap1(context, &load_func, 0x30000);
  check_trap2(context, &store_func, 0x30000, 0);

  error = wasmtime_memory_grow(context, &memory, 1, &old_size);
  assert(error != NULL);
  wasmtime_error_delete(error);
  error = wasmtime_memory_grow(context, &memory, 0, &old_size);
  if (error != NULL)
    exit_with_error("failed to grow memory", error, trap);

  // Create stand-alone memory.
  printf("Creating stand-alone memory...\n");
  wasm_limits_t limits = {5, 5};
  wasm_memorytype_t *memorytype = wasm_memorytype_new(&limits);
  wasmtime_memory_t memory2;
  error = wasmtime_memory_new(context, memorytype, &memory2);
  if (error != NULL)
    exit_with_error("failed to create memory", error, trap);
  wasm_memorytype_delete(memorytype);
  check(wasmtime_memory_size(context, &memory2) == 5);

  // Shut down.
  printf("Shutting down...\n");
  wasmtime_store_delete(store);
  wasm_engine_delete(engine);

  // All done.
  printf("Done.\n");
  return 0;
}

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap) {
  fprintf(stderr, "error: %s\n", message);
  wasm_byte_vec_t error_message;
  if (error != NULL) {
    wasmtime_error_message(error, &error_message);
    wasmtime_error_delete(error);
  } else {
    wasm_trap_message(trap, &error_message);
```

```
    wasm_trap_delete(trap);
  }
  fprintf(stderr, "%.*s\n", (int)error_message.size, error_message.data);
  wasm_byte_vec_delete(&error_message);
  exit(1);
}
```

WASI

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to instantiate a wasm module using WASI imports.

Wasm Source code

```
fn main() {
    println!("Hello, world!");
}
```

wasi.c

```
/*
Example of instantiating a WebAssembly which uses WASI imports.

You can compile and run this example on Linux with:

   cmake examples/
   cargo build --release -p wasmtime-c-api
   cc examples/wasi/main.c \
       -I crates/c-api/include \
       target/release/libwasmtime.a \
       -lpthread -ldl -lm \
       -o wasi
   ./wasi

Note that on Windows and macOS the command will be similar, but you'll need
to tweak the `-lpthread` and such annotations.

You can also build using cmake:

mkdir build && cd build && cmake .. && cmake --build . --target wasmtime-wasi
*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <wasi.h>
#include <wasm.h>
#include <wasmtime.h>

#define MIN(a, b) ((a) < (b) ? (a) : (b))

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap);

int main() {
  // Set up our context
  wasm_engine_t *engine = wasm_engine_new();
  assert(engine != NULL);
  wasmtime_store_t *store = wasmtime_store_new(engine, NULL, NULL);
  assert(store != NULL);
  wasmtime_context_t *context = wasmtime_store_context(store);

  // Create a linker with WASI functions defined
  wasmtime_linker_t *linker = wasmtime_linker_new(engine);
  wasmtime_error_t *error = wasmtime_linker_define_wasi(linker);
  if (error != NULL)
    exit_with_error("failed to link wasi", error, NULL);

  wasm_byte_vec_t wasm;
  // Load our input file to parse it next
  FILE *file = fopen("target/wasm32-wasip1/debug/wasi.wasm", "rb");
  if (!file) {
    printf("> Error loading file!\n");
    exit(1);
```

```
  }
  fseek(file, 0L, SEEK_END);
  size_t file_size = ftell(file);
  wasm_byte_vec_new_uninitialized(&wasm, file_size);
  fseek(file, 0L, SEEK_SET);
  if (fread(wasm.data, file_size, 1, file) != 1) {
    printf("> Error loading module!\n");
    exit(1);
  }
  fclose(file);

  // Compile our modules
  wasmtime_module_t *module = NULL;
  error = wasmtime_module_new(engine, (uint8_t *)wasm.data, wasm.size,
&module);
  if (!module)
    exit_with_error("failed to compile module", error, NULL);
  wasm_byte_vec_delete(&wasm);

  // Instantiate wasi
  wasi_config_t *wasi_config = wasi_config_new();
  assert(wasi_config);
  wasi_config_inherit_argv(wasi_config);
  wasi_config_inherit_env(wasi_config);
  wasi_config_inherit_stdin(wasi_config);
  wasi_config_inherit_stdout(wasi_config);
  wasi_config_inherit_stderr(wasi_config);
  wasm_trap_t *trap = NULL;
  error = wasmtime_context_set_wasi(context, wasi_config);
  if (error != NULL)
    exit_with_error("failed to instantiate WASI", error, NULL);

  // Instantiate the module
  error = wasmtime_linker_module(linker, context, "", 0, module);
  if (error != NULL)
    exit_with_error("failed to instantiate module", error, NULL);

  // Run it.
  wasmtime_func_t func;
  error = wasmtime_linker_get_default(linker, context, "", 0, &func);
  if (error != NULL)
    exit_with_error("failed to locate default export for module", error, NULL);

  error = wasmtime_func_call(context, &func, NULL, 0, NULL, 0, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("error calling default export", error, trap);

  // Clean up after ourselves at this point
  wasmtime_module_delete(module);
  wasmtime_store_delete(store);
  wasm_engine_delete(engine);
  return 0;
}

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap) {
  fprintf(stderr, "error: %s\n", message);
  wasm_byte_vec_t error_message;
```

```
  if (error != NULL) {
    wasmtime_error_message(error, &error_message);
    wasmtime_error_delete(error);
  } else {
    wasm_trap_message(trap, &error_message);
    wasm_trap_delete(trap);
  }
  fprintf(stderr, "%.*s\n", (int)error_message.size, error_message.data);
  wasm_byte_vec_delete(&error_message);
  exit(1);
}
```

Linking modules

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to compile and instantiate modules which link together.

`linking1.wat`

```
(module
  (import "linking2" "double" (func $double (param i32) (result i32)))
  (import "linking2" "log" (func $log (param i32 i32)))
  (import "linking2" "memory" (memory 1))
  (import "linking2" "memory_offset" (global $offset i32))

  (func (export "run")
    ;; Call into the other module to double our number, and we could print it
    ;; here but for now we just drop it
    i32.const 2
    call $double
    drop

    ;; Our `data` segment initialized our imported memory, so let's print the
    ;; string there now.
    global.get $offset
    i32.const 14
    call $log
  )

  (data (global.get $offset) "Hello, world!\n")
)
```

linking2.wat

```
(module
  (type $fd_write_ty (func (param i32 i32 i32 i32) (result i32)))
  (import "wasi_snapshot_preview1" "fd_write" (func $fd_write (type
$fd_write_ty)))

  (func (export "double") (param i32) (result i32)
    local.get 0
    i32.const 2
    i32.mul
  )

  (func (export "log") (param i32 i32)
    ;; store the pointer in the first iovec field
    i32.const 4
    local.get 0
    i32.store

    ;; store the length in the first iovec field
    i32.const 4
    local.get 1
    i32.store offset=4

    ;; call the `fd_write` import
    i32.const 1     ;; stdout fd
    i32.const 4     ;; iovs start
    i32.const 1     ;; number of iovs
    i32.const 0     ;; where to write nwritten bytes
    call $fd_write
    drop
  )

  (memory (export "memory") 2)
  (global (export "memory_offset") i32 (i32.const 65536))
)
```

linking.c

```
/*
Example of compiling, instantiating, and linking two WebAssembly modules
together.

You can compile and run this example on Linux with:

   cargo build --release -p wasmtime-c-api
   cc examples/linking.c \
       -I crates/c-api/include \
       target/release/libwasmtime.a \
       -lpthread -ldl -lm \
       -o linking
   ./linking

Note that on Windows and macOS the command will be similar, but you'll need
to tweak the `-lpthread` and such annotations.

You can also build using cmake:

mkdir build && cd build && cmake .. && cmake --build . --target wasmtime-
linking
*/

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <wasi.h>
#include <wasm.h>
#include <wasmtime.h>

#define MIN(a, b) ((a) < (b) ? (a) : (b))

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap);
static void read_wat_file(wasm_engine_t *engine, wasm_byte_vec_t *bytes,
                          const char *file);

int main() {
  // Set up our context
  wasm_engine_t *engine = wasm_engine_new();
  assert(engine != NULL);
  wasmtime_store_t *store = wasmtime_store_new(engine, NULL, NULL);
  assert(store != NULL);
  wasmtime_context_t *context = wasmtime_store_context(store);

  wasm_byte_vec_t linking1_wasm, linking2_wasm;
  read_wat_file(engine, &linking1_wasm, "examples/linking1.wat");
  read_wat_file(engine, &linking2_wasm, "examples/linking2.wat");

  // Compile our two modules
  wasmtime_error_t *error;
  wasmtime_module_t *linking1_module = NULL;
  wasmtime_module_t *linking2_module = NULL;
  error = wasmtime_module_new(engine, (uint8_t *)linking1_wasm.data,
```

```
                                linking1_wasm.size, &linking1_module);
if (error != NULL)
  exit_with_error("failed to compile linking1", error, NULL);
error = wasmtime_module_new(engine, (uint8_t *)linking2_wasm.data,
                                linking2_wasm.size, &linking2_module);
if (error != NULL)
  exit_with_error("failed to compile linking2", error, NULL);
wasm_byte_vec_delete(&linking1_wasm);
wasm_byte_vec_delete(&linking2_wasm);

// Configure WASI and store it within our `wasmtime_store_t`
wasi_config_t *wasi_config = wasi_config_new();
assert(wasi_config);
wasi_config_inherit_argv(wasi_config);
wasi_config_inherit_env(wasi_config);
wasi_config_inherit_stdin(wasi_config);
wasi_config_inherit_stdout(wasi_config);
wasi_config_inherit_stderr(wasi_config);
wasm_trap_t *trap = NULL;
error = wasmtime_context_set_wasi(context, wasi_config);
if (error != NULL)
  exit_with_error("failed to instantiate wasi", NULL, trap);

// Create our linker which will be linking our modules together, and then add
// our WASI instance to it.
wasmtime_linker_t *linker = wasmtime_linker_new(engine);
error = wasmtime_linker_define_wasi(linker);
if (error != NULL)
  exit_with_error("failed to link wasi", error, NULL);

// Instantiate `linking2` with our linker.
wasmtime_instance_t linking2;
error = wasmtime_linker_instantiate(linker, context, linking2_module,
                                    &linking2, &trap);
if (error != NULL || trap != NULL)
  exit_with_error("failed to instantiate linking2", error, trap);

// Register our new `linking2` instance with the linker
error = wasmtime_linker_define_instance(linker, context, "linking2",
                                        strlen("linking2"), &linking2);
if (error != NULL)
  exit_with_error("failed to link linking2", error, NULL);

// Instantiate `linking1` with the linker now that `linking2` is defined
wasmtime_instance_t linking1;
error = wasmtime_linker_instantiate(linker, context, linking1_module,
                                    &linking1, &trap);
if (error != NULL || trap != NULL)
  exit_with_error("failed to instantiate linking1", error, trap);

// Lookup our `run` export function
wasmtime_extern_t run;
bool ok = wasmtime_instance_export_get(context, &linking1, "run", 3, &run);
assert(ok);
assert(run.kind == WASMTIME_EXTERN_FUNC);
error = wasmtime_func_call(context, &run.of.func, NULL, 0, NULL, 0, &trap);
if (error != NULL || trap != NULL)
  exit_with_error("failed to call run", error, trap);
```

```c
  // Clean up after ourselves at this point
  wasmtime_linker_delete(linker);
  wasmtime_module_delete(linking1_module);
  wasmtime_module_delete(linking2_module);
  wasmtime_store_delete(store);
  wasm_engine_delete(engine);
  return 0;
}

static void read_wat_file(wasm_engine_t *engine, wasm_byte_vec_t *bytes,
                          const char *filename) {
  wasm_byte_vec_t wat;
  // Load our input file to parse it next
  FILE *file = fopen(filename, "r");
  if (!file) {
    printf("> Error loading file!\n");
    exit(1);
  }
  fseek(file, 0L, SEEK_END);
  size_t file_size = ftell(file);
  wasm_byte_vec_new_uninitialized(&wat, file_size);
  fseek(file, 0L, SEEK_SET);
  if (fread(wat.data, file_size, 1, file) != 1) {
    printf("> Error loading module!\n");
    exit(1);
  }
  fclose(file);

  // Parse the wat into the binary wasm format
  wasmtime_error_t *error = wasmtime_wat2wasm(wat.data, wat.size, bytes);
  if (error != NULL)
    exit_with_error("failed to parse wat", error, NULL);
  wasm_byte_vec_delete(&wat);
}

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap) {
  fprintf(stderr, "error: %s\n", message);
  wasm_byte_vec_t error_message;
  if (error != NULL) {
    wasmtime_error_message(error, &error_message);
    wasmtime_error_delete(error);
  } else {
    wasm_trap_message(trap, &error_message);
    wasm_trap_delete(trap);
  }
  fprintf(stderr, "%.*s\n", (int)error_message.size, error_message.data);
  wasm_byte_vec_delete(&error_message);
  exit(1);
}
```

Debugging

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to set up a module for dynamic runtime debugging via a native debugger like GDB or LLDB.

main.c

```c
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wasm.h>
#include <wasmtime.h>

#ifdef WASMTIME_TEST_ONLY
// These are the declarations provided from GDB documentation, used to validate
// that we actually added some DWARF info:
//
https://sourceware.org/gdb/current/onlinedocs/gdb.html/Declarations.html#Declar
ations
//
// NOTE: These are not required in your code, rather they are used for wasmtime
// testing only.
typedef enum {
  JIT_NOACTION = 0,
  JIT_REGISTER_FN,
  JIT_UNREGISTER_FN
} jit_actions_t;

struct jit_code_entry {
  struct jit_code_entry *next_entry;
  struct jit_code_entry *prev_entry;
  const char *symfile_addr;
  uint64_t symfile_size;
};

struct jit_descriptor {
  uint32_t version;
  /* This type should be jit_actions_t, but we use uint32_t
     to be explicit about the bitwidth.  */
  uint32_t action_flag;
  struct jit_code_entry *relevant_entry;
  struct jit_code_entry *first_entry;
};

/*
 * Import the descriptor, defined elsewhere in wasmtime
 */
extern struct jit_descriptor __jit_debug_descriptor;
#endif

#define own

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap);

int main(int argc, const char *argv[]) {
  // Configuring engine to support generating of DWARF info.
  // lldb can be used to attach to the program and observe
  // original fib-wasm.c source code and variables.
  wasm_config_t *config = wasm_config_new();
```

```
  wasmtime_config_debug_info_set(config, true);
  wasmtime_config_cranelift_opt_level_set(config, WASMTIME_OPT_LEVEL_NONE);

  // Initialize.
  printf("Initializing...\n");
  wasm_engine_t *engine = wasm_engine_new_with_config(config);
  wasmtime_store_t *store = wasmtime_store_new(engine, NULL, NULL);
  wasmtime_context_t *context = wasmtime_store_context(store);

#ifdef WASMTIME_TEST_ONLY
  // NOTE: This validation is for wasmtime testing and should not be included
in
  // your code.
  if (__jit_debug_descriptor.first_entry != NULL) {
    fprintf(stderr, "FAIL: JIT descriptor is already initialized\n");
    return 1;
  }
#endif

  // Load binary.
  printf("Loading binary...\n");
  FILE *file = fopen("target/wasm32-unknown-unknown/debug/fib.wasm", "rb");
  if (!file) {
    printf("> Error opening module!\n");
    return 1;
  }
  fseek(file, 0L, SEEK_END);
  size_t file_size = ftell(file);
  fseek(file, 0L, SEEK_SET);
  wasm_byte_vec_t binary;
  wasm_byte_vec_new_uninitialized(&binary, file_size);
  if (fread(binary.data, file_size, 1, file) != 1) {
    printf("> Error reading module!\n");
    return 1;
  }
  fclose(file);

  // Compile.
  printf("Compiling module...\n");
  wasmtime_module_t *module = NULL;
  wasmtime_error_t *error =
      wasmtime_module_new(engine, (uint8_t *)binary.data, binary.size,
&module);
  if (!module)
    exit_with_error("failed to compile module", error, NULL);
  wasm_byte_vec_delete(&binary);

  // Instantiate.
  printf("Instantiating module...\n");
  wasmtime_instance_t instance;
  wasm_trap_t *trap = NULL;
  error = wasmtime_instance_new(context, module, NULL, 0, &instance, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to instantiate", error, trap);
  wasmtime_module_delete(module);

#ifdef WASMTIME_TEST_ONLY
  // NOTE: This validation is for wasmtime testing and should not be included
```

```
in
  // your code.
  if (__jit_debug_descriptor.first_entry == NULL) {
    fprintf(stderr, "FAIL: JIT descriptor is NOT initialized\n");
    return 1;
  }
#endif

  // Extract export.
  wasmtime_extern_t fib;
  bool ok = wasmtime_instance_export_get(context, &instance, "fib", 3, &fib);
  assert(ok);

  // Call.
  printf("Calling fib...\n");
  wasmtime_val_t params[1];
  params[0].kind = WASMTIME_I32;
  params[0].of.i32 = 6;
  wasmtime_val_t results[1];
  error =
      wasmtime_func_call(context, &fib.of.func, params, 1, results, 1, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to call function", error, trap);

  assert(results[0].kind == WASMTIME_I32);
  printf("> fib(6) = %d\n", results[0].of.i32);

  // Shut down.
  printf("Shutting down...\n");
  wasmtime_store_delete(store);
  wasm_engine_delete(engine);

  // All done.
  printf("Done.\n");
  return 0;
}

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap) {
  fprintf(stderr, "error: %s\n", message);
  wasm_byte_vec_t error_message;
  if (error != NULL) {
    wasmtime_error_message(error, &error_message);
  } else {
    wasm_trap_message(trap, &error_message);
  }
  fprintf(stderr, "%.*s\n", (int)error_message.size, error_message.data);
  wasm_byte_vec_delete(&error_message);
  exit(1);
}
```

Using multi-value

You can also browse this source code online and clone the wasmtime repository to run the example locally.

This example shows off how to interact with a wasm module that uses multi-value exports and imports.

`multi.wat`

```
(module
  (func $f (import "" "f") (param i32 i64) (result i64 i32))

  (func $g (export "g") (param i32 i64) (result i64 i32)
    (call $f (local.get 0) (local.get 1))
  )

  (func $round_trip_many
    (export "round_trip_many")
    (param i64 i64 i64 i64 i64 i64 i64 i64 i64 i64)
    (result i64 i64 i64 i64 i64 i64 i64 i64 i64 i64)

    local.get 0
    local.get 1
    local.get 2
    local.get 3
    local.get 4
    local.get 5
    local.get 6
    local.get 7
    local.get 8
    local.get 9)
)
```

multi.c

```
/*
Example of instantiating of the WebAssembly module and invoking its exported
function.

You can compile and run this example on Linux with:

   cargo build --release -p wasmtime-c-api
   cc examples/multi.c \
       -I crates/c-api/include \
       target/release/libwasmtime.a \
       -lpthread -ldl -lm \
       -o multi
   ./multi

Note that on Windows and macOS the command will be similar, but you'll need
to tweak the `-lpthread` and such annotations.

You can also build using cmake:

mkdir build && cd build && cmake .. && cmake --build . --target wasmtime-multi

Also note that this example was taken from
https://github.com/WebAssembly/wasm-c-api/blob/master/example/multi.c
originally
*/

#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wasm.h>
#include <wasmtime.h>

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap);

// A function to be called from Wasm code.
wasm_trap_t *callback(void *env, wasmtime_caller_t *caller,
                      const wasmtime_val_t *args, size_t nargs,
                      wasmtime_val_t *results, size_t nresults) {
  printf("Calling back...\n");
  printf("> %" PRIu32 " %" PRIu64 "\n", args[0].of.i32, args[1].of.i64);
  printf("\n");

  results[0] = args[1];
  results[1] = args[0];
  return NULL;
}

// A function closure.
wasm_trap_t *closure_callback(void *env, wasmtime_caller_t *caller,
                              const wasmtime_val_t *args, size_t nargs,
                              wasmtime_val_t *results, size_t nresults) {
  int i = *(int *)env;
```

```c
  printf("Calling back closure...\n");
  printf("> %d\n", i);

  results[0].kind = WASMTIME_I32;
  results[0].of.i32 = (int32_t)i;
  return NULL;
}

int main(int argc, const char *argv[]) {
  // Initialize.
  printf("Initializing...\n");
  wasm_engine_t *engine = wasm_engine_new();
  wasmtime_store_t *store = wasmtime_store_new(engine, NULL, NULL);
  wasmtime_context_t *context = wasmtime_store_context(store);

  // Load our input file to parse it next
  FILE *file = fopen("examples/multi.wat", "r");
  if (!file) {
    printf("> Error loading file!\n");
    return 1;
  }
  fseek(file, 0L, SEEK_END);
  size_t file_size = ftell(file);
  fseek(file, 0L, SEEK_SET);
  wasm_byte_vec_t wat;
  wasm_byte_vec_new_uninitialized(&wat, file_size);
  if (fread(wat.data, file_size, 1, file) != 1) {
    printf("> Error loading module!\n");
    return 1;
  }
  fclose(file);

  // Parse the wat into the binary wasm format
  wasm_byte_vec_t binary;
  wasmtime_error_t *error = wasmtime_wat2wasm(wat.data, wat.size, &binary);
  if (error != NULL)
    exit_with_error("failed to parse wat", error, NULL);
  wasm_byte_vec_delete(&wat);

  // Compile.
  printf("Compiling module...\n");
  wasmtime_module_t *module = NULL;
  error =
      wasmtime_module_new(engine, (uint8_t *)binary.data, binary.size,
&module);
  if (error)
    exit_with_error("failed to compile module", error, NULL);
  wasm_byte_vec_delete(&binary);

  // Create external print functions.
  printf("Creating callback...\n");
  wasm_functype_t *callback_type =
      wasm_functype_new_2_2(wasm_valtype_new_i32(), wasm_valtype_new_i64(),
                            wasm_valtype_new_i64(), wasm_valtype_new_i32());
  wasmtime_func_t callback_func;
  wasmtime_func_new(context, callback_type, callback, NULL, NULL,
                    &callback_func);
  wasm_functype_delete(callback_type);
```

```
  // Instantiate.
  printf("Instantiating module...\n");
  wasmtime_extern_t imports[1];
  imports[0].kind = WASMTIME_EXTERN_FUNC;
  imports[0].of.func = callback_func;
  wasmtime_instance_t instance;
  wasm_trap_t *trap = NULL;
  error = wasmtime_instance_new(context, module, imports, 1, &instance, &trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to instantiate", error, trap);
  wasmtime_module_delete(module);

  // Extract export.
  printf("Extracting export...\n");
  wasmtime_extern_t run;
  bool ok = wasmtime_instance_export_get(context, &instance, "g", 1, &run);
  assert(ok);
  assert(run.kind == WASMTIME_EXTERN_FUNC);

  // Call.
  printf("Calling export...\n");
  wasmtime_val_t args[2];
  args[0].kind = WASMTIME_I32;
  args[0].of.i32 = 1;
  args[1].kind = WASMTIME_I64;
  args[1].of.i64 = 2;
  wasmtime_val_t results[2];
  error = wasmtime_func_call(context, &run.of.func, args, 2, results, 2,
&trap);
  if (error != NULL || trap != NULL)
    exit_with_error("failed to call run", error, trap);

  // Print result.
  printf("Printing result...\n");
  printf("> %" PRIu64 " %" PRIu32 "\n", results[0].of.i64, results[1].of.i32);

  assert(results[0].kind == WASMTIME_I64);
  assert(results[0].of.i64 == 2);
  assert(results[1].kind == WASMTIME_I32);
  assert(results[1].of.i32 == 1);

  // Shut down.
  printf("Shutting down...\n");
  wasmtime_store_delete(store);
  wasm_engine_delete(engine);

  // All done.
  printf("Done.\n");
  return 0;
}

static void exit_with_error(const char *message, wasmtime_error_t *error,
                            wasm_trap_t *trap) {
  fprintf(stderr, "error: %s\n", message);
  wasm_byte_vec_t error_message;
  if (error != NULL) {
    wasmtime_error_message(error, &error_message);
```

```
    wasmtime_error_delete(error);
  } else {
    wasm_trap_message(trap, &error_message);
    wasm_trap_delete(trap);
  }
  fprintf(stderr, "%.*s\n", (int)error_message.size, error_message.data);
  wasm_byte_vec_delete(&error_message);
  exit(1);
}
```

Using WebAssembly from Python

Wasmtime is available on PyPI and can be used programmatically or as a python module loader, which allows almost any WebAssembly module to be used as a python module. This guide will go over adding Wasmtime to your project, and some provided examples of what can be done with WebAssembly modules.

Make sure you've got Python 3.5 or newer installed locally, and we can get started!

Getting started and simple example

First, copy this example WebAssembly text module into your project with the filename `gcd.wat`. It exports a function for calculating the greatest common denominator of two numbers.

```
(module
  (func $gcd (param i32 i32) (result i32)
    (local i32)
    block  ;; label = @1
      block  ;; label = @2
        local.get 0
        br_if 0 (;@2;)
        local.get 1
        local.set 2
        br 1 (;@1;)
      end
      loop  ;; label = @2
        local.get 1
        local.get 0
        local.tee 2
        i32.rem_u
        local.set 0
        local.get 2
        local.set 1
        local.get 0
        br_if 0 (;@2;)
      end
    end
    local.get 2
  )
  (export "gcd" (func $gcd))
)
```

Next, install the Wasmtime package from PyPi. It can be installed as a dependency through Pip or related tools such as Pipenv.

```
pip install wasmtime
```

Or

```
pipenv install wasmtime
```

After you have Wasmtime installed and you've imported `wasmtime`, you can import WebAssembly modules in your project like any other python module.

```
import wasmtime.loader
import gcd

print("gcd(27, 6) =", gcd.gcd(27, 6))
```

This script should output

```
gcd(27, 6) = 3
```

If this is the output you see, congrats! You've successfully ran your first WebAssembly code in python!

You can also alternatively use the `wasmtime` package's API:

```
from wasmtime import Store, Module, Instance

store = Store()
module = Module.from_file(store.engine, 'gcd.wat')
instance = Instance(store, module, [])
gcd = instance.exports(store)['gcd']
print("gcd(27, 6) = %d" % gcd(store, 27, 6))
```

More examples and contributing

The `wasmtime` Python package currently lives in its own repository outside of `wasmtime` and has a number of other more advanced examples as well. Feel free to browse those, but if you find anything missing don't hesitate to open an issue and let us know if you have any questions!

Using WebAssembly from .NET

The Wasmtime NuGet package can be used to programmatically interact with WebAssembly modules.

This guide will go over adding Wasmtime to your project and demonstrate a simple example of using a WebAssembly module from C#.

Make sure you have a .NET Core SDK 3.0 SDK or later installed before we get started!

Getting started and simple example

Start by creating a new .NET Core console project:

```
$ mkdir gcd
$ cd gcd
$ dotnet new console
```

Next, add a reference to the Wasmtime NuGet package to your project:

```
$ dotnet add package --version 0.19.0-preview1 wasmtime
```

Copy this example WebAssembly text module into your project directory as `gcd.wat`.

```
(module
  (func $gcd (param i32 i32) (result i32)
    (local i32)
    block  ;; label = @1
      block  ;; label = @2
        local.get 0
        br_if 0 (;@2;)
        local.get 1
        local.set 2
        br 1 (;@1;)
      end
      loop  ;; label = @2
        local.get 1
        local.get 0
        local.tee 2
        i32.rem_u
        local.set 0
        local.get 2
        local.set 1
        local.get 0
        br_if 0 (;@2;)
      end
    end
    local.get 2
  )
  (export "gcd" (func $gcd))
)
```

This module exports a function for calculating the greatest common denominator of two numbers.

Replace the code in `Program.cs` with the following:

```
using System;
using Wasmtime;

namespace Tutorial
{
    class Program
    {
        static void Main(string[] args)
        {
            using var engine = new Engine();
            using var module = Module.FromTextFile(engine, "gcd.wat");

            using var host = new Host(engine);
            using dynamic instance = host.Instantiate(module);

            Console.WriteLine($"gcd(27, 6) = {instance.gcd(27, 6)}");
        }
    }
}
```

Run the .NET core program:

```
$ dotnet run
```

The program should output:

```
gcd(27, 6) = 3
```

If this is the output you see, congrats! You've successfully ran your first WebAssembly code in .NET!

More examples and contributing

The .NET embedding of Wasmtime repository contains the source code for the Wasmtime NuGet package.

The repository also has more examples as well.

Feel free to browse those, but if you find anything missing don't hesitate to open an issue and let us know if you have any questions!

Using WebAssembly from Go

Wasmtime is available as a Go Module. This guide will go over adding Wasmtime to your project, and some provided examples of what can be done with WebAssembly modules.

Make sure you're using Go 1.12 or later with modules support.

Getting started and simple example

First up you'll want to start a new module:

```
$ mkdir hello-wasm
$ cd hello-wasm
$ go mod init hello-wasm
$ go get github.com/bytecodealliance/wasmtime-go
```

Next, copy this example WebAssembly text module into your project. It exports a function for calculating the greatest common denominator of two numbers.

```
(module
  (func $gcd (param i32 i32) (result i32)
    (local i32)
    block  ;; label = @1
      block  ;; label = @2
        local.get 0
        br_if 0 (;@2;)
        local.get 1
        local.set 2
        br 1 (;@1;)
      end
      loop  ;; label = @2
        local.get 1
        local.get 0
        local.tee 2
        i32.rem_u
        local.set 0
        local.get 2
        local.set 1
        local.get 0
        br_if 0 (;@2;)
      end
    end
    local.get 2
  )
  (export "gcd" (func $gcd))
)
```

Next, we can write our code in `main.go` which reads this file and runs it:

```
package main

import (
    "fmt"
    "github.com/bytecodealliance/wasmtime-go"
)

func main() {
    engine := wasmtime.NewEngine()
    store := wasmtime.NewStore(engine)
    module, err := wasmtime.NewModuleFromFile(engine, "gcd.wat")
    check(err)
    instance, err := wasmtime.NewInstance(store, module, []wasmtime.AsExtern{})
    check(err)

    gcd := instance.GetExport(store, "gcd").Func()
    val, err := gcd.Call(store, 6, 27)
    check(err)
    fmt.Printf("gcd(6, 27) = %d\n", val.(int32))
}

func check(err error) {
    if err != nil {
        panic(err)
    }
}
```

And finally we can build and run it:

```
$ go run main.go
gcd(6, 27) = 3
```

If this is the output you see, congrats! You've successfully ran your first WebAssembly code in Go!

More examples and contributing

The `wasmtime` Go package lives in its own repository and has a number of other more advanced examples as well. Feel free to browse those, but if you find anything missing don't hesitate to open an issue and let us know if you have any questions!

Using WebAssembly from Bash

Getting started and simple example

First up you'll want to start a new module:

```
$ mkdir -p gcd-bash
$ cd gcd-bash
$ touch gcd.wat gcd.sh
```

Next, copy this example WebAssembly text module into your project. It exports a function for calculating the greatest common denominator of two numbers.

`gcd.wat`

```
(module
  (func $gcd (param i32 i32) (result i32)
    (local i32)
    block  ;; label = @1
      block  ;; label = @2
        local.get 0
        br_if 0 (;@2;)
        local.get 1
        local.set 2
        br 1 (;@1;)
      end
      loop  ;; label = @2
        local.get 1
        local.get 0
        local.tee 2
        i32.rem_u
        local.set 0
        local.get 2
        local.set 1
        local.get 0
        br_if 0 (;@2;)
      end
    end
    local.get 2
  )
  (export "gcd" (func $gcd))
)
```

Create a bash script that will invoke GCD three times.

gcd.sh

```
#!/bin/bash

function gcd() {
  # Cast to number; default = 0
  local x=$(($1))
  local y=$(($2))
  # Invoke GCD from module; suppress stderr
  local result=$(wasmtime --invoke gcd examples/gcd.wat $x $y 2>/dev/null)
  echo "$result"
}

# main
for num in "27 6" "6 27" "42 12"; do
  set -- $num
  echo "gcd($1, $2) = $(gcd "$1" "$2")"
done
```

Using WebAssembly from Ruby

Wasmtime is available on RubyGems and can be used programmatically to interact with Wasm modules. This guide will go over installing the Wasmtime gem and running a simple Wasm module from Ruby.

Make sure you've got Ruby 3.0 or newer installed locally, and we can get started!

Getting started and simple example

First, copy this example WebAssembly text module into your project. It exports a function for calculating the greatest common divisor of two numbers.

```
(module
  (func $gcd (param i32 i32) (result i32)
    (local i32)
    block  ;; label = @1
      block  ;; label = @2
        local.get 0
        br_if 0 (;@2;)
        local.get 1
        local.set 2
        br 1 (;@1;)
      end
      loop  ;; label = @2
        local.get 1
        local.get 0
        local.tee 2
        i32.rem_u
        local.set 0
        local.get 2
        local.set 1
        local.get 0
        br_if 0 (;@2;)
      end
    end
    local.get 2
  )
  (export "gcd" (func $gcd))
)
```

Next, install the Wasmtime Ruby gems by either adding it your project's `Gemfile`:

```
bundle add wasmtime
```

Or by using the `gem` command directly:

```
gem install wasmtime
```

The gem has a Rust-based native extension, but thanks to precompiled gems, you should not have to compile anything. It'll just work!

Now that you have the Wasmtime gem installed, let's create a Ruby script to execute the `gcd` module from before.

```ruby
require "wasmtime"

engine = Wasmtime::Engine.new
mod = Wasmtime::Module.from_file(engine, "gcd.wat")
store = Wasmtime::Store.new(engine)
instance = Wasmtime::Instance.new(store, mod)

puts "gcd(27, 6) = #{instance.invoke("gcd", 27, 6)}"
```

This script should output

```
gcd(27, 6) = 3
```

If this is the output you see, congrats! You've successfully ran your first WebAssembly code in Ruby!

More examples and contributing

To learn more, check out the more advanced examples and the API documentation. If you have any questions, do not hesitate to open an issue on the GitHub repository.

Using WebAssembly from Elixir

Wasmtime is available on Hex and can be used programmatically to interact with Wasm modules. This guide will go over installing the wasmex package and running a simple Wasm module from Elixir.

Getting started and simple example

First, copy this example WebAssembly text module into the current directory. It exports a function for calculating the greatest common denominator of two numbers.

```
(module
  (func $gcd (param i32 i32) (result i32)
    (local i32)
    block  ;; label = @1
      block  ;; label = @2
        local.get 0
        br_if 0 (;@2;)
        local.get 1
        local.set 2
        br 1 (;@1;)
      end
      loop  ;; label = @2
        local.get 1
        local.get 0
        local.tee 2
        i32.rem_u
        local.set 0
        local.get 2
        local.set 1
        local.get 0
        br_if 0 (;@2;)
      end
    end
    local.get 2
  )
  (export "gcd" (func $gcd))
)
```

The library has a Rust-based native extension, but thanks to `rustler_precompiled`, you should not have to compile anything. It'll just work!

This WAT file can be executed in `iex`:

```
Mix.install([:wasmex])
bytes = File.read!("gcd.wat")
{:ok, pid} = Wasmex.start_link(%{bytes: bytes}) # starts a GenServer running a
WASM instance
Wasmex.call_function(pid, "gcd", [27, 6])
```

The last command should output:

```
iex(5)> Wasmex.call_function(pid, "gcd", [27, 6])
{:ok, [3]}
```

If this is the output you see, congrats! You've successfully ran your first WebAssembly code in Elixir!

More examples and contributing

To learn more, check out an another example and the API documentation. If you have any questions, do not hesitate to open an issue on the GitHub repository.

Further Examples

The examples contained in this section explain how to use Wasmtime in several common scenarios.

Debugging WebAssembly

Wasmtime currently provides the following support for debugging misbehaving WebAssembly:

- We can live debug and step through the guest Wasm and the host at the same time with `gdb` or `lldb`.
- When a Wasm guest traps, we can generate Wasm core dumps, that can be consumed by other tools for post-mortem analysis.

Debugging with gdb and `lldb`

The following steps describe how to use `gdb` or `lldb` to debug both the Wasm guest and the host (i.e. the Wasmtime CLI or your Wasmtime-embedding program) at the same time:

1. Compile your WebAssembly with debug info enabled, usually `-g`; for example:

   ```
   clang foo.c -g -o foo.wasm
   ```

2. Run Wasmtime with the debug info enabled; this is `-D debug-info` from the CLI and `Config::debug_info(true)` in an embedding (e.g. see debugging in a Rust embedding). It's also recommended to use `-O opt-level=0` for better inspection of local variables if desired.

3. Use a supported debugger:

   ```
   lldb -- wasmtime run -D debug-info foo.wasm
   ```

   ```
   gdb --args wasmtime run -D debug-info -O opt-level=0 foo.wasm
   ```

If you run into trouble, the following discussions might help:

- On MacOS with LLDB you may need to run: `settings set plugin.jit-loader.gdb.enable on` (#1953)
- With LLDB, call `__vmctx.set()` to set the current context before calling any dereference operators (#1482):

  ```
  (lldb) p __vmctx->set()
  (lldb) p *foo
  ```

- The address of the start of instance memory can be found in `__vmctx->memory`
- On Windows you may experience degraded WASM compilation throughput due to the enablement of additional native heap checks when under the debugger by default. You can set the environment variable `_NO_DEBUG_HEAP` to `1` to disable them.

Debugging WebAssembly with Core Dumps

Wasmtime can be configured to generate the standard Wasm core dump format whenever guest Wasm programs trap. These core dumps can then be consumed by external tooling (such as `wasmgdb`) for post-mortem analysis.

This page focuses on generating and inspecting core dumps via the Wasmtime command-line interface. For details on how to generate core dumps via the `wasmtime` embedding API, see Core Dumps in a Rust Embedding.

First, we need to compile some code to Wasm that can trap. Consider the following Rust code:

```
// trap.rs

fn main() {
    foo(42);
}

fn foo(x: u32) {
    bar(x);
}

fn bar(x: u32) {
    baz(x);
}

fn baz(x: u32) {
    assert!(x != 42);
}
```

We can compile it to Wasm with the following command:

```
$ rustc --target wasm32-wasip1 -o ./trap.wasm ./trap.rs
```

Next, we can run it in Wasmtime and capture a core dump when it traps:

```
$ wasmtime -D coredump=./trap.coredump ./trap.wasm
thread 'main' panicked at /home/nick/scratch/trap.rs:14:5:
assertion failed: x != 42
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Error: failed to run main module `/home/nick/scratch/trap.wasm`

Caused by:
    0: core dumped at /home/nick/scratch/trap.coredump
    1: failed to invoke command default
    2: wasm coredump generated while executing store_name:
       modules:
         <module>
       instances:
         Instance(store=1, index=1)
       memories:
         Memory(store=1, index=1)
       globals:
         Global(store=1, index=0)
       backtrace:
       error while executing at wasm backtrace:
           0: 0x5961 - <unknown>!__rust_start_panic
           1: 0x562a - <unknown>!rust_panic
           2: 0x555d -
<unknown>!std::panicking::rust_panic_with_hook::h58e7d0b3d70e95b6
           3: 0x485d - <unknown>!std::panicking::begin_panic_handler::
{{closure}}::h1853004619879cfd
           4: 0x47bd -
<unknown>!std::sys_common::backtrace::__rust_end_short_backtrace::hed32bc555740
5634
           5: 0x4f02 - <unknown>!rust_begin_unwind
           6: 0xac01 - <unknown>!core::panicking::panic_fmt::h53ca5bf48b428895
           7: 0xb1c5 - <unknown>!core::panicking::panic::h62c2c2bb054da7e1
           8:  0x661 - <unknown>!trap::baz::h859f39b65389c077
           9:  0x616 - <unknown>!trap::bar::h7ad12f9c5b730d17
          10:  0x60a - <unknown>!trap::foo::ha69c95723611c1a0
          11:  0x5fe - <unknown>!trap::main::hdfcd9f2d150fc3dc
          12:  0x434 -
<unknown>!core::ops::function::FnOnce::call_once::h24336e950fb97d1e
          13:  0x40b -
<unknown>!std::sys_common::backtrace::__rust_begin_short_backtrace::h2b37384d2b
1a57ff
          14:  0x4ec - <unknown>!std::rt::lang_start::
{{closure}}::he86eb1b6ac6d7501
          15: 0x24f7 -
<unknown>!std::rt::lang_start_internal::h21f6a1d8f3633b54
          16:  0x497 - <unknown>!std::rt::lang_start::h7d256f21902ff32b
          17:  0x687 - <unknown>!__main_void
          18:  0x3e6 - <unknown>!_start
       note: using the `WASMTIME_BACKTRACE_DETAILS=1` environment variable may
show more debugging information
```

You now have a core dump at `./trap.coredump` that can be consumed by external tooling to do post-mortem analysis of the failure.

Profiling WebAssembly

One of WebAssembly's major goals is to be quite close to native code in terms of performance, so typically when executing Wasm you'll be quite interested in how well your Wasm module is performing! From time to time you might want to dive a bit deeper into the performance of your Wasm, and this is where profiling comes into the picture.

For best results, ideally you'd use hardware performance counters for your timing measurements. However, that requires special support from your CPU and operating system. Because Wasmtime is a JIT, that also requires hooks from Wasmtime to your platform's native profiling tools.

As a result, Wasmtime support for native profiling is limited to certain platforms. See the following sections of this book if you're using these platforms:

- On Linux, we support perf.
- For Intel's x86 CPUs on Linux or Windows, we support VTune.
- For Linux and macOS, we support samply.
- For everything else, see the cross-platform profiler.

The native profilers can measure time spent in WebAssembly guest code as well as time spent in the Wasmtime host and potentially even time spent in the kernel. This provides a comprehensive view of performance.

The cross-platform-profiler can only measure time spent in WebAssembly guest code, and its timing measurements are not as precise as the native profilers. However, it works on every platform that Wasmtime supports.

Using `perf` on Linux

One profiler supported by Wasmtime is the `perf` profiler for Linux. This is an extremely powerful profiler with lots of documentation on the web, but for the rest of this section we'll assume you're running on Linux and already have `perf` installed.

There are two profiling agents for `perf`:

- a very simple one that will map code regions to symbol names: `perfmap`.
- a more detailed one that can provide additional information and mappings between the source language statements and generated JIT code: `jitdump`.

Profiling with `perfmap`

Simple profiling support with `perf` generates a "perf map" file that the `perf` CLI will automatically look for, when running into unresolved symbols. This requires runtime support from Wasmtime itself, so you will need to manually change a few things to enable profiling support in your application. Enabling runtime support depends on how you're using Wasmtime:

- **Rust API** - you'll want to call the [`Config::profiler`] method with `ProfilingStrategy::PerfMap` to enable profiling of your wasm modules.
- **C API** - you'll want to call the `wasmtime_config_profiler_set` API with a `WASMTIME_PROFILING_STRATEGY_PERFMAP` value.
- **Command Line** - you'll want to pass the `--profile=perfmap` flag on the command line.

Once perfmap support is enabled, you'll use `perf record` like usual to record your application's performance.

For example if you're using the CLI, you'll execute:

```
$ perf record -k mono wasmtime --profile=perfmap foo.wasm
```

This will create a `perf.data` file as per usual, but it will *also* create a `/tmp/perf-XXXX.map` file. This extra `.map` file is the perf map file which is specified by `perf` and Wasmtime generates at runtime.

After that you can explore the `perf.data` profile as you usually would, for example with:

```
$ perf report --input perf.data
```

You should be able to see time spent in wasm functions, generate flamegraphs based on that, etc.. You should also see entries for wasm functions show up as one function and the name of each function matches the debug name section in the wasm file.

Note that support for perfmap is still relatively new in Wasmtime, so if you have any problems, please don't hesitate to file an issue!

Profiling with `jitdump`

Profiling support with `perf` uses the "jitdump" support in the `perf` CLI. This requires runtime support from Wasmtime itself, so you will need to manually change a few things to enable profiling support in your application. First you'll want to make sure that Wasmtime is compiled with the `jitdump` Cargo feature (which is enabled by default). Otherwise enabling runtime support depends on how you're using Wasmtime:

- **Rust API** - you'll want to call the [`Config::profiler`] method with `ProfilingStrategy::JitDump` to enable profiling of your wasm modules.
- **C API** - you'll want to call the `wasmtime_config_profiler_set` API with a `WASMTIME_PROFILING_STRATEGY_JITDUMP` value.
- **Command Line** - you'll want to pass the `--profile=jitdump` flag on the command line.

Once jitdump support is enabled, you'll use `perf record` like usual to record your application's performance. You'll need to also be sure to pass the `--clockid mono` or `-k mono` flag to `perf record`.

For example if you're using the CLI, you'll execute:

```
$ perf record -k mono wasmtime --profile=jitdump foo.wasm
```

This will create a `perf.data` file as per usual, but it will *also* create a `jit-XXXX.dump` file. This extra `*.dump` file is the jitdump file which is specified by `perf` and Wasmtime generates at runtime.

The next thing you need to do is to merge the `*.dump` file into the `perf.data` file, which you can do with the `perf inject` command:

```
$ perf inject --jit --input perf.data --output perf.jit.data
```

This will read `perf.data`, automatically pick up the `*.dump` file that's correct, and then create `perf.jit.data` which merges all the JIT information together. This should also create

a lot of `jitted-XXXX-N.so` files in the current directory which are ELF images for all the JIT functions that were created by Wasmtime.

After that you can explore the `perf.jit.data` profile as you usually would, for example with:

```
$ perf report --input perf.jit.data
```

You should be able to annotate wasm functions and see their raw assembly. You should also see entries for wasm functions show up as one function and the name of each function matches the debug name section in the wasm file.

Note that support for jitdump is still relatively new in Wasmtime, so if you have any problems, please don't hesitate to file an issue!

`perf` and DWARF information

If the jitdump profile doesn't give you enough information by default, you can also enable dwarf debug information to be generated for JIT code which should give the `perf` profiler more information about what's being profiled. This can include information like more descriptive function names, filenames, and line numbers.

Enabling dwarf debug information for JIT code depends on how you're using Wasmtime:

- **Rust API** - you'll want to call the `Config::debug_info` method.
- **C API** - you'll want to call the `wasmtime_config_debug_info_set` API.
- **Command Line** - you'll want to pass the `-g` flag on the command line.

You shouldn't need to do anything else to get this information into `perf`. The perf collection data should automatically pick up all this dwarf debug information.

`perf` example

Let's run through a quick example with `perf` to get the feel for things. First let's take a look at some wasm:

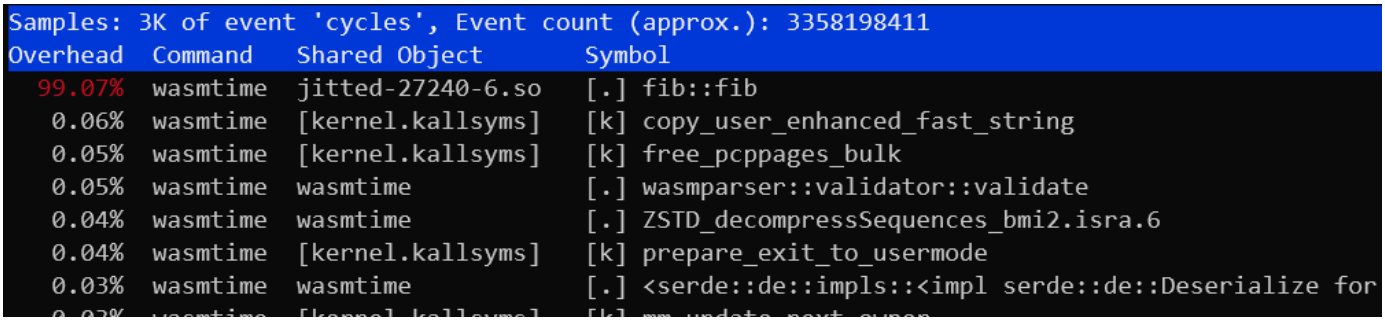
```rust
fn main() {
    let n = 42;
    println!("fib({}) = {}", n, fib(n));
}

fn fib(n: u32) -> u32 {
    if n <= 2 {
        1
    } else {
        fib(n - 1) + fib(n - 2)
    }
}
```

To collect perf information for this wasm module we'll execute:

```
$ rustc --target wasm32-wasip1 fib.rs -O
$ perf record -k mono wasmtime --profile=jitdump fib.wasm
fib(42) = 267914296
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.147 MB perf.data (3435 samples) ]
$ perf inject --jit --input perf.data --output perf.jit.data
```

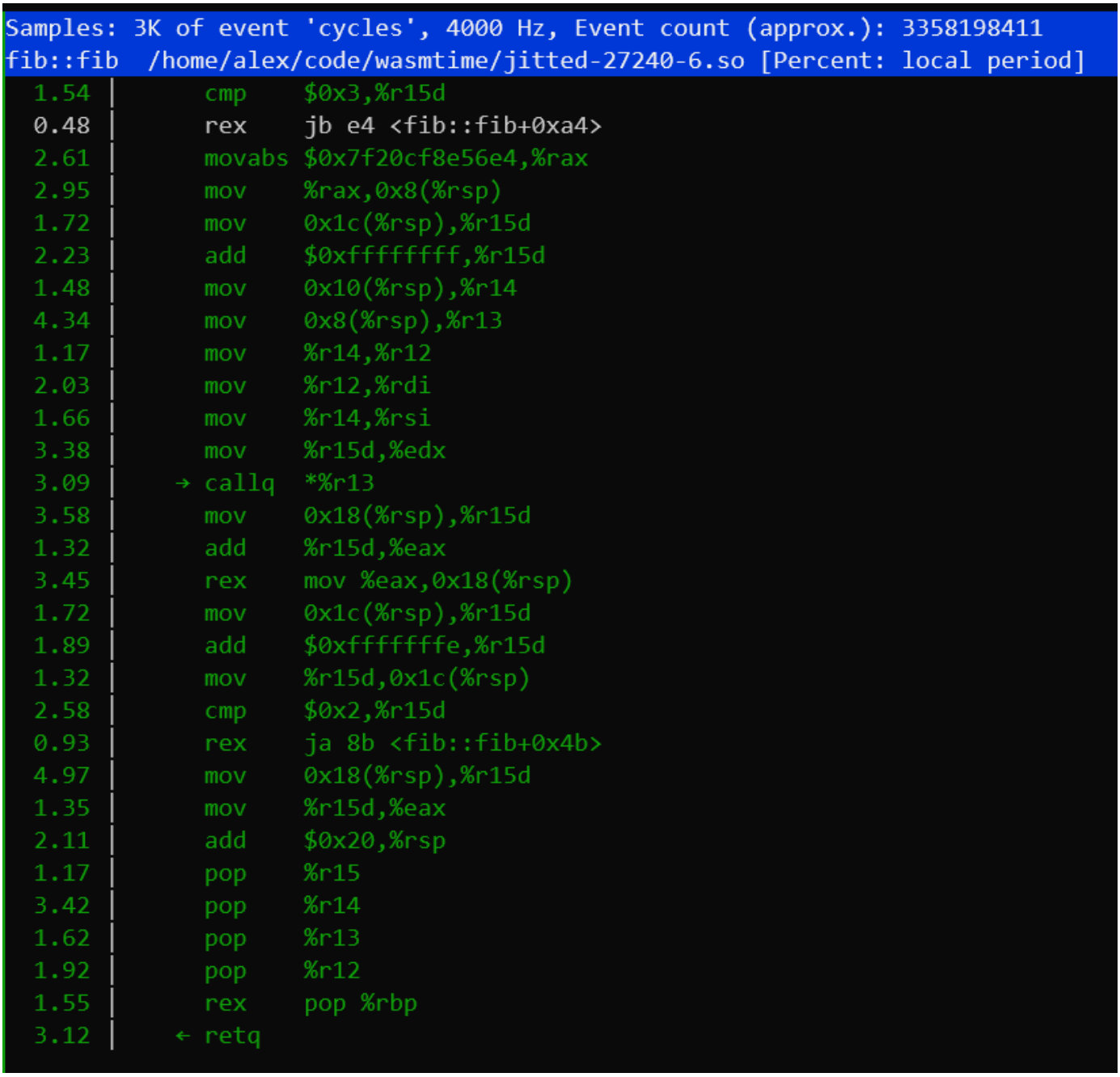
And we should have all our information now! We can execute `perf report` for example to see that 99% of our runtime (as expected) is spent in our `fib` function. Note that the symbol has been demangled to `fib::fib` which is what the Rust symbol is:

```
$ perf report --input perf.jit.data
```



Alternatively we could also use `perf annotate` to take a look at the disassembly of the `fib` function, seeing what the JIT generated:

```
$ perf annotate --input perf.jit.data
```


Samples: 3K of event 'cycles', 4000 Hz, Event count (approx.): 3358198411
fib::fib /home/alex/code/wasmtime/jitted-27240-6.so [Percent: local period]
1.54 cmp $0x3,%r15d
0.48 rex jb e4 <fib::fib+0xa4>
2.61 movabs $0x7f20cf8e56e4,%rax
2.95 mov %rax,0x8(%rsp)
1.72 mov 0x1c(%rsp),%r15d
2.23 add $0xffffffff,%r15d
1.48 mov 0x10(%rsp),%r14
4.34 mov 0x8(%rsp),%r13
1.17 mov %r14,%r12
2.03 mov %r12,%rdi
1.66 mov %r14,%rsi
3.38 mov %r15d,%edx
3.09 → callq *%r13
3.58 mov 0x18(%rsp),%r15d
1.32 add %r15d,%eax
3.45 rex mov %eax,0x18(%rsp)
1.72 mov 0x1c(%rsp),%r15d
1.89 add $0xfffffffe,%r15d
1.32 mov %r15d,0x1c(%rsp)
2.58 cmp $0x2,%r15d
0.93 rex ja 8b <fib::fib+0x4b>
4.97 mov 0x18(%rsp),%r15d
1.35 mov %r15d,%eax
2.11 add $0x20,%rsp
1.17 pop %r15
3.42 pop %r14
1.62 pop %r13
1.92 pop %r12
1.55 rex pop %rbp
3.12 ← retq

Using VTune

VTune is a popular performance profiling tool that targets both 32-bit and 64-bit x86 architectures. The tool collects profiling data during runtime and then, either through the command line or GUI, provides a variety of options for viewing and analyzing that data. VTune Profiler is available in both commercial and free options. The free, downloadable version is available here and is backed by a community forum for support. This version is appropriate for detailed analysis of your Wasm program.

VTune support in Wasmtime is provided through the JIT profiling APIs from the `ittapi` library. This library provides code generators (or the runtimes that use them) a way to report JIT activities. The APIs are implemented in a static library (see `ittapi` source) which Wasmtime links to when VTune support is specified through the `vtune` Cargo feature flag; this feature is not enabled by default. When the VTune collector is run, the `ittapi` library collects Wasmtime's reported JIT activities. This connection to `ittapi` is provided by the `ittapi-rs` crate.

For more information on VTune and the analysis tools it provides see its documentation.

Turn on VTune support

For JIT profiling with VTune, Wasmtime currently builds with the `vtune` feature enabled by default. This ensures the compiled binary understands how to inform the `ittapi` library of JIT events. But it must still be enabled at runtime--enable runtime support based on how you use Wasmtime:

- **Rust API** - call the [`Config::profiler`] method with `ProfilingStrategy::VTune` to enable profiling of your wasm modules.
- **C API** - call the `wasmtime_config_profiler_set` API with a `WASMTIME_PROFILING_STRATEGY_VTUNE` value.
- **Command Line** - pass the `--profile=vtune` flag on the command line.

Profiling Wasmtime itself

Note that VTune is capable of profiling a single process or all system processes. Like `perf`, VTune is capable of profiling the Wasmtime runtime itself without any added support. However, the `ittapi` APIs also provide an interface for marking the start and stop of code regions for easy isolation in the VTune Profiler. Support for these APIs is expected to be added in the future.

Example: Getting Started

With VTune properly installed, if you are using the CLI execute:

```
$ cargo build
$ vtune -run-pass-thru=--no-altstack -collect hotspots target/debug/wasmtime --
profile=vtune foo.wasm
```

This command tells the VTune collector (`vtune`) to collect hot spot profiling data as Wasmtime is executing `foo.wasm`. The `--profile=vtune` flag enables VTune support in Wasmtime so that the collector is also alerted to JIT events that take place during runtime. The first time this is run, the result of the command is a results directory `r000hs/` which contains profiling data for Wasmtime and the execution of `foo.wasm`. This data can then be read and displayed via the command line or via the VTune GUI by importing the result.

Example: CLI Collection

Using a familiar algorithm, we'll start with the following Rust code:

```
fn main() {
    let n = 45;
    println!("fib({}) = {}", n, fib(n));
}

fn fib(n: u32) -> u32 {
    if n <= 2 {
        1
    } else {
        fib(n - 1) + fib(n - 2)
    }
}
```

We compile the example to Wasm:

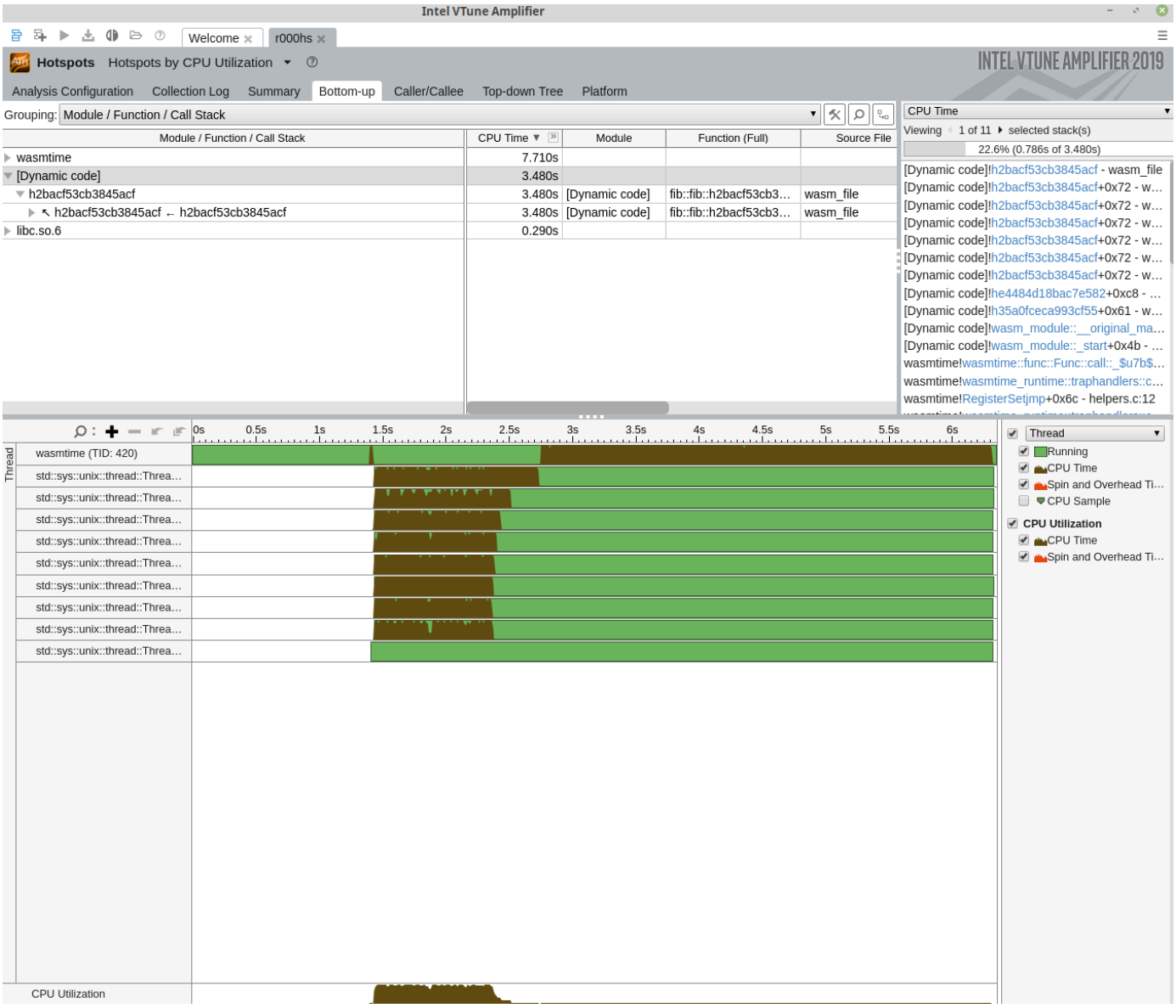
```
$ rustc --target wasm32-wasip1 fib.rs -C opt-level=z -C lto=yes
```

Then we execute the Wasmtime runtime (built with the `vtune` feature and executed with the `--profile=vtune` flag to enable reporting) inside the VTune CLI application, `vtune`, which must already be installed and available on the path. To collect hot spot profiling information, we execute:

```
$ rustc --target wasm32-wasip1 fib.rs -C opt-level=z -C lto=yes
$ vtune -run-pass-thru=--no-altstack -v -collect hotspots target/debug/wasmtime
--profile=vtune fib.wasm
fib(45) = 1134903170
amplxe: Collection stopped.
amplxe: Using result path /home/jlb6740/wasmtime/r000hs
amplxe: Executing actions  7 % Clearing the database
amplxe: The database has been cleared, elapsed time is 0.239 seconds.
amplxe: Executing actions 14 % Updating precomputed scalar metrics
amplxe: Raw data has been loaded to the database, elapsed time is 0.792
seconds.
amplxe: Executing actions 19 % Processing profile metrics and debug information
...
Top Hotspots
Function
Module          CPU Time
------------------------------------------------------------------------------
-------------  -------------  --------
h2bacf53cb3845acf
[Dynamic code]    3.480s
__memmove_avx_unaligned_erms
libc.so.6         0.222s
cranelift_codegen::ir::instructions::InstructionData::opcode::hee6f5b6a72fc684e
wasmtime          0.122s
core::ptr::slice_from_raw_parts::hc5cb6f1b39a0e7a1
wasmtime          0.066s
_$LT$usize$u20$as$u20$core..slice..SliceIndex$LT$$u5b$T$u5d$$GT$$GT$::get::h70c
7f142eeeee8bd  wasmtime          0.066s
```

Example: Importing Results into GUI

Results directories created by the `vtune` CLI can be imported in the VTune GUI by clicking "Open > Result". Below is a visualization of the collected data as seen in VTune's GUI:



Example: GUI Collection

VTune can collect data in multiple ways (see `vtune` CLI discussion above); another way is to use the VTune GUI directly. A standard work flow might look like:

- Open VTune Profiler
- "Configure Analysis" with
  - "Application" set to `/path/to/wasmtime` (e.g., `target/debug/wasmtime`)
  - "Application parameters" set to `--profile=vtune /path/to/module.wasm`
  - "Working directory" set as appropriate
  - Enable "Hardware Event-Based Sampling," which may require some system configuration, e.g. `sysctl -w kernel.perf_event_paranoid=0`
- Start the analysis

Using `samply` on Linux/macOS

One profiler supported by Wasmtime is `samply` for Linux and macOS. As of 17th July 2023, the latest version of samply (on crates.io) is 0.11.0 which does not seem to support perfmaps. To use this, you either need a newer version of samply, if by the time you read this, a newer version has been released, or you can build samply from source.

Profiling with `perfmap`

Simple profiling support with `samply` generates a "perfmap" file that the `samply` CLI will automatically look for, when running into unresolved symbols. This requires runtime support from Wasmtime itself, so you will need to manually change a few things to enable profiling support in your application. Enabling runtime support depends on how you're using Wasmtime:

- **Rust API** - you'll want to call the [`Config::profiler`] method with `ProfilingStrategy::PerfMap` to enable profiling of your wasm modules.
- **C API** - you'll want to call the `wasmtime_config_profiler_set` API with a `WASMTIME_PROFILING_STRATEGY_PERFMAP` value.
- **Command Line** - you'll want to pass the `--profile=perfmap` flag on the command line.

Once perfmap support is enabled, you'll use `samply record` like usual to record your application's performance.

For example if you're using the CLI, you'll execute:

```
$ samply record wasmtime --profile=perfmap foo.wasm
```

This will record your application's performance and open the Firefox profiler UI to view the results. It will also dump its own profile data to a json file (called `profile.json`) in the current directory.

Note that support for perfmap is still relatively new in Wasmtime, so if you have any problems, please don't hesitate to file an issue!

Using Wasmtime's cross-platform profiler

The guest profiling strategy enables in-process sampling and will write the captured profile to a file which can be viewed at https://profiler.firefox.com/.

To use this profiler with the Wasmtime CLI, pass the `--profile=guest[,path[,interval]]` flag.

- `path` is where to write the profile, `wasmtime-guest-profile.json` by default
- `interval` is the duration between samples, 10ms by default

When used with `-W timeout=N`, the timeout will be rounded up to the nearest multiple of the profiling interval.

Wasm memcheck (wmemcheck)

wmemcheck provides the ability to check for invalid mallocs, reads, and writes inside a Wasm module, as long as Wasmtime is able to make certain assumptions (`malloc` and `free` functions are visible and your program uses only the default allocator). This is analogous to the Valgrind tool's memory checker (memcheck) tool for native programs.

How to use:

1. When building Wasmtime, add the CLI flag "--features wmemcheck" to compile with wmemcheck configured.

   ```
   cargo build --features wmemcheck
   ```

2. When running your wasm module, add the CLI flag "-W wmemcheck".

   ```
   wasmtime run -W wmemcheck test.wasm
   ```

If your program executes an invalid operation (load or store to non-allocated address, double-free, or an internal error in malloc that allocates the same memory twice) you will see an error that looks like a Wasm trap. For example, given the program

```c
#include <stdlib.h>

int main() {
    char* p = malloc(1024);
    *p = 0;
    free(p);
    *p = 0;
}
```

compiled with WASI-SDK via

```
$ /opt/wasi-sdk/bin/clang -o test.wasm test.c
```

you can observe the memory checker working like so:

```
$ wasmtime run -W wmemcheck ./test.wasm
Error: failed to run main module `./test.wasm`

Caused by:
    0: failed to invoke command default
    1: error while executing at wasm backtrace:
           0:  0x103 - <unknown>!__original_main
           1:   0x87 - <unknown>!_start
           2: 0x2449 - <unknown>!_start.command_export
    2: Invalid store at addr 0x10610 of size 1
```

Building a Minimal Wasmtime embedding

Wasmtime embeddings may wish to optimize for binary size and runtime footprint to fit on a small system. This documentation is intended to guide some features of Wasmtime and how to best produce a minimal build of Wasmtime.

Building a minimal CLI

> *Note*: the exact numbers in this section were last updated on 2024-12-12 on a Linux x86_64 host. For up-to-date numbers consult the artifacts in the `dev` release of Wasmtime where the `min/lib/libwasmtime.so` binary represents the culmination of these steps.

Many Wasmtime embeddings go through the `wasmtime` crate as opposed to the Wasmtime C API `libwasmtime.so`, but to start out let's take a look at minimizing the dynamic library as a case study. By default the C API is relatively large:

```
$ cargo build -p wasmtime-c-api
$ ls -lh ./target/debug/libwasmtime.so
-rwxrwxr-x 2 alex alex 260M Dec 12 07:46 target/debug/libwasmtime.so
```

The easiest size optimization is to compile with optimizations. This will strip lots of dead code and additionally generate much less debug information by default

```
$ cargo build -p wasmtime-c-api --release
$ ls -lh ./target/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 19M Dec 12 07:46 target/release/libwasmtime.so
```

Much better, but still relatively large! The next thing that can be done is to disable the default features of the C API. This will remove all optional functionality from the crate and strip it down to the bare bones functionality.

```
$ cargo build -p wasmtime-c-api --release --no-default-features
$ ls -lh ./target/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 2.1M Dec 12 07:47 target/release/libwasmtime.so
```

Note that this library is stripped to the bare minimum of functionality which notably means it does not have a compiler for WebAssembly files. This means that compilation is no longer supported meaning that `*.cwasm` files must used to create a module. Additionally error messages will be worse in this mode as less contextual information is provided.

The final Wasmtime-specific optimization you can apply is to disable logging statements. Wasmtime and its dependencies make use of the `log` crate and `tracing` crate for debugging and diagnosing. For a minimal build this isn't needed though so this can all be disabled through Cargo features to shave off a small amount of code. Note that for custom embeddings you'd need to replicate the `disable-logging` feature which sets the `max_level_off` feature for the `log` and `tracing` crate.

```
$ cargo build -p wasmtime-c-api --release --no-default-features --features
disable-logging
$ ls -lh ./target/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 2.1M Dec 12 07:49 target/release/libwasmtime.so
```

At this point the next line of tricks to apply to minimize binary size are general tricks-of-the-trade for Rust programs and are no longer specific to Wasmtime. For example the first thing that can be done is to optimize for size rather than speed via rustc's `s` optimization level. This uses Cargo's environment-variable based configuration via the `CARGO_PROFILE_RELEASE_OPT_LEVEL=s` environment variable to configure this.

```
$ export CARGO_PROFILE_RELEASE_OPT_LEVEL=s
$ cargo build -p wasmtime-c-api --release --no-default-features --features
disable-logging
$ ls -lh ./target/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 2.4M Dec 12 07:49 target/release/libwasmtime.so
```

Note that the size has increased here slightly instead of going down. Optimizing for speed-vs-size can affect a number of heuristics in LLVM so it's best to test out locally what's best for your embedding. Further examples below continue to pass this flag since by the end it will produce a smaller binary than the default optimization level of "3" for release mode. You may wish to also try an optimization level of "2" and see which produces a smaller build for you.

After optimizations levels the next compilation setting to configure is Rust's "panic=abort" mode where panics translate to process aborts rather than unwinding. This removes landing pads from code as well as unwind tables from the executable.

```
$ export CARGO_PROFILE_RELEASE_OPT_LEVEL=s
$ export CARGO_PROFILE_RELEASE_PANIC=abort
$ cargo build -p wasmtime-c-api --release --no-default-features --features
disable-logging
$ ls -lh ./target/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 2.0M Dec 12 07:49 target/release/libwasmtime.so
```

Next, if the compile time hit is acceptable, LTO can be enabled to provide deeper opportunities for compiler optimizations to remove dead code and deduplicate. Do note that this will take a significantly longer amount of time to compile than previously. Here LTO is configured with `CARGO_PROFILE_RELEASE_LTO=true`.

```
$ export CARGO_PROFILE_RELEASE_OPT_LEVEL=s
$ export CARGO_PROFILE_RELEASE_PANIC=abort
$ export CARGO_PROFILE_RELEASE_LTO=true
$ cargo build -p wasmtime-c-api --release --no-default-features --features
disable-logging
$ ls -lh ./target/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 1.2M Dec 12 07:50 target/release/libwasmtime.so
```

Similar to LTO above rustc can be further instructed to place all crates into their own single object file instead of multiple by default. This again increases compile times. Here that's done with `CARGO_PROFILE_RELEASE_CODEGEN_UNITS=1`.

```
$ export CARGO_PROFILE_RELEASE_OPT_LEVEL=s
$ export CARGO_PROFILE_RELEASE_PANIC=abort
$ export CARGO_PROFILE_RELEASE_LTO=true
$ export CARGO_PROFILE_RELEASE_CODEGEN_UNITS=1
$ cargo build -p wasmtime-c-api --release --no-default-features --features
disable-logging
$ ls -lh ./target/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 1.2M Dec 12 07:50 target/release/libwasmtime.so
```

Note that with LTO using a single codegen unit may only have marginal benefit. If not using LTO, however, a single codegen unit will likely provide benefit over the default 16 codegen units.

One final flag before getting to nightly features is to strip debug information from the standard library. In `--release` mode Cargo by default doesn't generate debug information for local crates, but the Rust standard library may have debug information still included with it. This is configured via `CARGO_PROFILE_RELEASE_STRIP=debuginfo`

```
$ export CARGO_PROFILE_RELEASE_OPT_LEVEL=s
$ export CARGO_PROFILE_RELEASE_PANIC=abort
$ export CARGO_PROFILE_RELEASE_LTO=true
$ export CARGO_PROFILE_RELEASE_CODEGEN_UNITS=1
$ export CARGO_PROFILE_RELEASE_STRIP=debuginfo
$ cargo build -p wasmtime-c-api --release --no-default-features --features
disable-logging
$ ls -lh ./target/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 1.2M Dec 12 07:50 target/release/libwasmtime.so
```

Next, if your use case allows it, the Nightly Rust toolchain provides a number of other options to minimize the size of binaries. Note the usage of `+nightly` here to the `cargo` command to use a Nightly toolchain (assuming your local toolchain is installed with rustup). Also note that due to the nature of nightly the exact flags here may not work in the future. Please open an issue with Wasmtime if these commands don't work and we'll update the documentation.

The first nightly feature we can leverage is to remove filename and line number information in panics with `-Zlocation-detail=none`

```
$ export CARGO_PROFILE_RELEASE_OPT_LEVEL=s
$ export CARGO_PROFILE_RELEASE_PANIC=abort
$ export CARGO_PROFILE_RELEASE_LTO=true
$ export CARGO_PROFILE_RELEASE_CODEGEN_UNITS=1
$ export CARGO_PROFILE_RELEASE_STRIP=debuginfo
$ export RUSTFLAGS="-Zlocation-detail=none"
$ cargo +nightly build -p wasmtime-c-api --release --no-default-features --
features disable-logging
$ ls -lh ./target/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 1.2M Dec 12 07:51 target/release/libwasmtime.so
```

Further along the line of nightly features the next optimization will recompile the standard library without unwinding information, trimming out a bit more from the standard library. This uses the `-Zbuild-std` flag to Cargo. Note that this additionally requires `--target` as well which will need to be configured for your particular platform.

```
$ export CARGO_PROFILE_RELEASE_OPT_LEVEL=s
$ export CARGO_PROFILE_RELEASE_PANIC=abort
$ export CARGO_PROFILE_RELEASE_LTO=true
$ export CARGO_PROFILE_RELEASE_CODEGEN_UNITS=1
$ export CARGO_PROFILE_RELEASE_STRIP=debuginfo
$ export RUSTFLAGS="-Zlocation-detail=none"
$ cargo +nightly build -p wasmtime-c-api --release --no-default-features --
features disable-logging \
    -Z build-std=std,panic_abort --target x86_64-unknown-linux-gnu
$ ls -lh target/x86_64-unknown-linux-gnu/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 941K Dec 12 07:52 target/x86_64-unknown-linux-
gnu/release/libwasmtime.so
```

Next the Rust standard library has some optional features in addition to Wasmtime, such as printing of backtraces. This may not be required in minimal environments so the features of the standard library can be disabled with the `-Zbuild-std-features=` flag which configures the set of enabled features to be empty.

```
$ export CARGO_PROFILE_RELEASE_OPT_LEVEL=s
$ export CARGO_PROFILE_RELEASE_PANIC=abort
$ export CARGO_PROFILE_RELEASE_LTO=true
$ export CARGO_PROFILE_RELEASE_CODEGEN_UNITS=1
$ export CARGO_PROFILE_RELEASE_STRIP=debuginfo
$ export RUSTFLAGS="-Zlocation-detail=none"
$ cargo +nightly build -p wasmtime-c-api --release --no-default-features --
features disable-logging \
    -Z build-std=std,panic_abort --target x86_64-unknown-linux-gnu \
    -Z build-std-features=
$ ls -lh target/x86_64-unknown-linux-gnu/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 784K Dec 12 07:53 target/x86_64-unknown-linux-
gnu/release/libwasmtime.so
```

And finally, if you can enable the `panic_immediate_abort` feature of the Rust standard library to shrink panics even further. Note that this comes at a cost of making bugs/panics very difficult to debug.

```
$ export CARGO_PROFILE_RELEASE_OPT_LEVEL=s
$ export CARGO_PROFILE_RELEASE_PANIC=abort
$ export CARGO_PROFILE_RELEASE_LTO=true
$ export CARGO_PROFILE_RELEASE_CODEGEN_UNITS=1
$ export CARGO_PROFILE_RELEASE_STRIP=debuginfo
$ export RUSTFLAGS="-Zlocation-detail=none"
$ cargo +nightly build -p wasmtime-c-api --release --no-default-features --
features disable-logging \
    -Z build-std=std,panic_abort --target x86_64-unknown-linux-gnu \
    -Z build-std-features=panic_immediate_abort
$ ls -lh target/x86_64-unknown-linux-gnu/release/libwasmtime.so
-rwxrwxr-x 2 alex alex 698K Dec 12 07:54 target/x86_64-unknown-linux-
gnu/release/libwasmtime.so
```

Minimizing further

Above shows an example of taking the default `cargo build` result of 260M down to a 700K binary for the `libwasmtime.so` binary of the C API. Similar steps can be done to reduce the size of the `wasmtime` CLI executable as well. This is currently the smallest size with the source code as-is, but there are more size reductions which haven't been implemented yet.

This is a listing of some example sources of binary size. Some sources of binary size may not apply to custom embeddings since, for example, your custom embedding might already not use WASI and might already not be included.

- Unused functionality in the C API - building `libwasmtime.{a,so}` can show a misleading file size because the linker is unable to remove unused code. For example `libwasmtime.so` contains all code for the C API but your embedding may not be using all of the symbols present so in practice the final linked binary will often be much smaller than `libwasmtime.so`. Similarly `libwasmtime.a` is forced to contain the entire C API so its size is likely much larger than a linked application. For a minimal embedding it's recommended to link against `libwasmtime.a` with `--gc-sections` as a linker flag and evaluate the size of your own application.

- Formatting strings in Wasmtime - Wasmtime makes extensive use of formatting strings for error messages and other purposes throughout the implementation. Most of this is intended for debugging and understanding more when something goes wrong, but much of this is not necessary for a truly minimal embedding. In theory much of this could be conditionally compiled out of the Wasmtime project to produce a smaller executable. Just how much of the final binary size is accounted for by formatting string is unknown, but it's well known in Rust that `std::fmt` is not the slimmest of modules.

- CLI: WASI implementation - currently the CLI includes all of WASI. This includes two separate implementations of WASI - one for preview2 and one for preview1. This accounts for 1M+ of space which is a significant chunk of the remaining ~2M. While removing just preview2 or preview1 would be easy enough with a Cargo feature, the

resulting executable wouldn't be able to do anything. Something like a plugin feature for the CLI, however, would enable removing WASI while still being a usable executable. Note that the C API's implementation of WASI can be disabled because custom host functionality can be provided.

- CLI: Argument parsing - as a command line executable `wasmtime` contains parsing of command line arguments which currently uses the `clap` crate. This contributes ~200k of binary size to the final executable which would likely not be present in a custom embedding of Wasmtime. While this can't be removed from Wasmtime it's something to consider when evaluating the size of CI artifacts.

- Cranelift vs Winch - the "min" builds on CI exclude Cranelift from their binary footprint but this comes at a cost of the final binary not supporting compilation of wasm modules. If this is required then no effort has yet been put into minimizing the code size of Cranelift itself. One possible tradeoff that can be made though is to choose between the Winch baseline compiler vs Cranelift. Winch should be much smaller from a compiled footprint point of view while not sacrificing everything in terms of performance. Note though that Winch is still under development.

Above are some future avenues to take in terms of reducing the binary size of Wasmtime and various tradeoffs that can be made. The Wasmtime project is eager to hear embedder use cases/profiles if Wasmtime is not suitable for binary size reasons today. Please feel free to open an issue and let us know and we'd be happy to discuss more how best to handle a particular use case.

Building Wasmtime for a Custom Platform

Wasmtime supports a wide range of functionality by default on major operating systems such as Windows, macOS, and Linux, but this functionality is not necessarily present on all platforms (much less custom platforms). Most of Wasmtime's features are gated behind either platform-specific configuration flags or Cargo feature flags. The `wasmtime` crate for example documents important crate features which likely want to be disabled for custom platforms.

Not all of Wasmtime's features are supported on all platforms, but many are enabled by default. For example the `parallel-compilation` crate feature requires the host platform to have threads, or in other words the Rust `rayon` crate must compile for your platform. If the `parallel-compilation` feature is disabled, though, then `rayon` won't be compiled. For a custom platform, one of the first things you'll want to do is to disable the default features of the `wasmtime` crate (or C API).

Some important features to be aware of for custom platforms are:

- `runtime` - you likely want to enable this feature since this includes the runtime to actually execute WebAssembly binaries.

- `cranelift` and `winch` - you likely want to disable these features. This primarily cuts down on binary size. Note that you'll need to use `*.cwasm` artifacts so wasm files will need to be compiled outside of the target platform and transferred to them.

- `signals-based-traps` - without this feature Wasmtime won't rely on host OS signals (e.g. segfaults) at runtime and will instead perform manual checks to avoid signals. This increases portability at the cost of runtime performance. For maximal portability leave this disabled.

When compiling Wasmtime for an unknown platform, for example "not Windows" or "not Unix", then Wasmtime will need some symbols to be provided by the embedder to operate correctly. The header file at `examples/min-platform/embedding/wasmtime-platform.h` describes the symbols that the Wasmtime runtime requires to work which your platform will need to provide. Some important notes about this are:

- `wasmtime_{setjmp,longjmp}` are required for trap handling at this time. These are thin wrappers around the standard `setjmp` and `longjmp` symbols you'll need to provide. An example implementation looks like this. In the future this dependency is likely going to go away as trap handling and unwinding is migrated to compiled code (e.g. Cranelift) itself.

- `wasmtime_tls_{get,set}` are required for the runtime to operate. Effectively a single pointer of TLS storage is necessary. Whether or not this is actually stored in TLS is up to the embedder, for example storage in `static` memory is ok if the embedder knows it won't be using threads.

- `WASMTIME_SIGNALS_BASED_TRAPS` - if this `#define` is given (e.g. the `signals-based-traps` feature was enabled at compile time), then your platform must have the concept of virtual memory and support `mmap`-like APIs and signal handling. Many APIs in this header are disabled if `WASMTIME_SIGNALS_BASED_TRAPS` is turned off which is why it's more portable, but if you enable this feature all of these APIs must be implemented.

You can find an example in the `wasmtime` repository of building a minimal embedding. Note that for Rust code you'll be using `#![no_std]` and you'll need to provide a memory allocator and a panic handler as well. The memory alloator will likely get hooked up to your platform's memory allocator and the panic handler mostly just needs to abort.

Building Wasmtime for a custom platform is not a turnkey process right now, there are a number of points that need to be considered:

- For a truly custom platform you'll probably want to create a custom Rust target. This means that Nightly Rust will be required.

- Wasmtime depends on the availability of a memory allocator (e.g. `malloc`). Wasmtime assumes that failed memory allocation aborts execution (except for the case of allocating linear memories and growing them).

- Not all features for Wasmtime can be built for custom targets. For example WASI support does not work on custom targets. When building Wasmtime you'll probably want `--no-default-features` and will then want to incrementally add features back in as needed.

The `examples/min-platform` directory has an example of building this minimal embedding and some necessary steps. Combined with the above features about producing a minimal build currently produces a 400K library on Linux.

Stability

... more coming soon

Release Process

Wasmtime's release process was originally designed in an RFC and this page is intended to serve as documentation for the current process as-is today. The high-level summary of Wasmtime's release process is:

- A new major version of Wasmtime will be made available once a month.
- Security bugs and correctness fixes will be backported to the latest two releases of Wasmtime and issued as patch releases.

Once a month Wasmtime will issue a new major version. This will be issued with a semver-major version update, such as 4.0.0 to 5.0.0. The precise schedule of Wasmtime's release is currently an automated PR is sent to bump the version on the 5th of every month and a release is made when the PR is merged. The PR typically gets merged within a few days.

Each major release of Wasmtime reserves the right to break both behavior and API backwards-compatibility. This is not expected to happen frequently, however, and any breaking change will follow these criteria:

- Minor breaking changes, either behavior or with APIs, will be documented in the `RELEASES.md` release notes. Minor changes will require some degree of consensus but are not required to go through the entire RFC process.

- Major breaking changes, such as major refactorings to the API, will be required to go through the RFC process. These changes are intended to be broadly communicated to those interested and provides an opportunity to give feedback about embeddings. Release notes will clearly indicate if any major breaking changes through accepted RFCs are included in a release.

Patch releases of Wasmtime will only be issued for security and critical correctness issues for on-by-default behavior in the previous releases. If Wasmtime is currently at version 5.0.0 then 5.0.1 and 4.0.1 will be issued as patch releases if a bug is found. Patch releases are guaranteed to maintain API and behavior backwards-compatibility and are intended to be trivial for users to upgrade to.

Patch releases for Cranelift will be made for any miscompilations found by Cranelift, even those that Wasmtime itself may not exercise. Due to the current release process a patch release for Cranelift will issue a patch release for Wasmtime as well.

What's released?

At this time the release process of Wasmtime encompasses:

- The `wasmtime` Rust crate

- The C API of Wasmtime
- The `wasmtime` CLI tool through the `wasmtime-cli` Rust crate

Other projects maintained by the Bytecode Alliance will also likely be released, with the same version numbers, with the main Wasmtime project soon after a release is made, such as:

- `wasmtime-dotnet`
- `wasmtime-py`
- `wasmtime-go`
- `wasmtime-cpp`
- `wasmtime-rb`

Note, though, that bugs and security issues in these projects do not at this time warrant patch releases for Wasmtime.

Tiers of Support in Wasmtime

Wasmtime's support for platforms and features can be distinguished with three different tiers of support. The description of these tiers are intended to be inspired by the Rust compiler's support tiers for targets but are additionally tailored for Wasmtime. Wasmtime's tiered support additionally applies not only to platforms/targets themselves but additionally features implemented within Wasmtime itself.

The purpose of this document is to provide a means by which to evaluate the inclusion of new features and support for existing features within Wasmtime. This should not be used to "lawyer" a change into Wasmtime on a precise technical detail or similar since this document is itself not 100% precise and will change over time.

Current Tier Status

For explanations of what each tier means see below.

Tier 1

Category	Description
Target	`x86_64-apple-darwin`
Target	`x86_64-pc-windows-msvc`
Target	`x86_64-unknown-linux-gnu`
Compiler Backend	Cranelift
WebAssembly Proposal	`mutable-globals`
WebAssembly Proposal	`sign-extension-ops`
WebAssembly Proposal	`nontrapping-float-to-int-conversion`
WebAssembly Proposal	`multi-value`
WebAssembly Proposal	`bulk-memory`
WebAssembly Proposal	`reference-types`
WebAssembly Proposal	`simd`
WebAssembly Proposal	`component-model`
WebAssembly Proposal	`relaxed-simd`
WebAssembly Proposal	`multi-memory`
WebAssembly Proposal	`threads`
WebAssembly Proposal	`tail-call`
WASI Proposal	`wasi-io`

Category	Description
WASI Proposal	`wasi-clocks`
WASI Proposal	`wasi-filesystem`
WASI Proposal	`wasi-random`
WASI Proposal	`wasi-sockets`
WASI Proposal	`wasi-http`
WASI Proposal	`wasi_snapshot_preview1`
WASI Proposal	`wasi_unstable`

Tier 2

Category	Description	Missing Tier 1 Requirements
Target	`aarch64-unknown-linux-gnu`	Continuous fuzzing
Target	`aarch64-apple-darwin`	Continuous fuzzing
Target	`s390x-unknown-linux-gnu`	Continuous fuzzing
Target	`x86_64-pc-windows-gnu`	Clear owner of the target
Target	Support for `#![no_std]`	Support beyond CI checks
WebAssembly Proposal	`memory64`	Unstable wasm proposal
WebAssembly Proposal	`function-references`	Unstable wasm proposal
WebAssembly Proposal	`wide-arithmetic`	Unstable wasm proposal

Tier 3

Category	Description	Missing Tier 2 Requirements
Target	`aarch64-pc-windows-msvc`	CI testing, unwinding, full-time maintainer
Target	`riscv64gc-unknown-linux-gnu`	full-time maintainer
Target	`wasm32-wasip1` [1]	Supported but not tested
Target	`aarch64-linux-android`	CI testing, full-time maintainer

Category	Description	Missing Tier 2 Requirements
Target	`x86_64-linux-android`	CI testing, full-time maintainer
Target	`x86_64-unknown-linux-musl` [2]	CI testing, full-time maintainer
Target	`x86_64-unknown-illumos`	CI testing, full-time maintainer
Target	`x86_64-unknown-freebsd`	CI testing, full-time maintainer
Compiler Backend	Winch on x86_64	WebAssembly proposals (`simd` , `relaxed-simd` , `tail-call` , `reference-types` , `threads`)
Compiler Backend	Winch on aarch64	Complete implementation
WebAssembly Proposal	`gc`	Complete implementation
WASI Proposal	`wasi-nn`	More expansive CI testing
WASI Proposal	`wasi-threads`	More CI, unstable proposal
WASI Proposal	`wasi-config`	unstable proposal
WASI Proposal	`wasi-keyvalue`	unstable proposal
misc	Non-Wasmtime Cranelift usage [3]	CI testing, full-time maintainer
misc	DWARF debugging [4]	CI testing, full-time maintainer, improved quality

[3] This is intended to encompass features that Cranelift supports as a general-purpose code generator such as integer value types other than `i32` and `i64` , non-Wasmtime calling conventions, code model settings, relocation restrictions, etc. These features aren't covered by Wasmtime's usage of Cranelift because the WebAssembly instruction set doesn't leverage them. This means that they receive far less testing and fuzzing than the parts of Cranelift exercised by Wasmtime.

[4] Currently there is no active maintainer of DWARF debugging support and support is currently best-effort. Additionally there are known shortcomings and bugs. At this time there's no developer time to improve the situation here as well.

[1] Wasmtime's `cranelift` and `winch` features can be compiled to WebAssembly but not the `runtime` feature at this time. This means that Wasmtime-compiled-to-wasm can itself compile wasm but cannot execute wasm.

[2] Binary artifacts for MUSL are dynamically linked, not statically linked, meaning that they are not suitable for "run on any linux distribution" style use cases. Wasmtime does not have static binary artifacts at this time and that will require building from source.

Unsupported features and platforms

While this is not an exhaustive list, Wasmtime does not currently have support for the following features. Note that this is intended to document Wasmtime's current state and does not mean Wasmtime does not want to ever support these features; rather design discussion and PRs are welcome for many of the below features to figure out how best to implement them and at least move them to Tier 3 above.

- Target: ARM 32-bit
- Target: AArch64 FreeBSD
- Target: NetBSD/OpenBSD
- Target: i686 (32-bit Intel targets)
- Target: MIPS
- Target: SPARC
- Target: PowerPC
- Target: RISC-V 32-bit
- WebAssembly Proposals: see documentation here
- WASI proposal: `proxy-wasm`
- WASI proposal: `wasi-blob-store`
- WASI proposal: `wasi-crypto`
- WASI proposal: `wasi-data`
- WASI proposal: `wasi-distributed-lock-service`
- WASI proposal: `wasi-grpc`
- WASI proposal: `wasi-message-queue`
- WASI proposal: `wasi-parallel`
- WASI proposal: `wasi-pubsub`
- WASI proposal: `wasi-sql`
- WASI proposal: `wasi-url`

Tier Details

Wasmtime's precise definitions of tiers are not guaranteed to be constant over time, so these descriptions are likely to change over time. Tier 1 is classified as the highest level of support, confidence, and correctness for a component. Each tier additionally encompasses all the guarantees of previous tiers.

Features classified under a particular tier may already meet the criteria for later tiers as well. In situations like this it's not intended to use these guidelines to justify removal of a feature at any one point in time. Guidance is provided here for phasing out unmaintained features but it should be clear under what circumstances work "can be avoided" for each tier.

Tier 3 - Not Production Ready

The general idea behind Tier 3 is that this is the baseline for inclusion of code into the Wasmtime project. This is not intended to be a catch-all "if a patch is sent it will be merged" tier. Instead the goal of this tier is to outline what is expected of contributors adding new features to Wasmtime which might be experimental at the time of addition. This is intentionally not a relaxed tier of restrictions but already implies a significant commitment of effort to a feature being included within Wasmtime.

Tier 3 features include:

- Inclusion of a feature does not impose unnecessary maintenance overhead on other components/features. Some examples of additions to Wasmtime which would not be accepted are:

    - An experimental feature doubles the time of CI for all PRs.
    - A change which makes it significantly more difficult to architecturally change Wasmtime's internal implementation.
    - A change which makes building Wasmtime more difficult for unrelated developers.

    In general Tier 3 features are off-by-default at compile time but still tested-by-default on CI.

- New features of Wasmtime cannot have major known bugs at the time of inclusion. Landing a feature in Wasmtime requires the feature to be correct and bug-free as best can be evaluated at the time of inclusion. Inevitably bugs will be found and that's ok, but anything identified during review must be addressed.

- Code included into the Wasmtime project must be of an acceptable level of quality relative to the rest of the code in Wasmtime.

- There must be a path to a feature being finished at the time of inclusion. Adding a new backend to Cranelift for example is a significant undertaking which may not be able to be done in a single PR. Partial implementations of a feature are acceptable so long as there's a clear path forward and schedule for completing the feature.

- New components in Wasmtime must have a clearly identified owner who is willing to be "on the hook" for review, updates to the internals of Wasmtime, etc. For example a new backend in Cranelift would need to have a maintainer who is willing to respond to changes in Cranelift's interfaces and the needs of Wasmtime.

This baseline level of support notably does not require any degree of testing, fuzzing, or verification. As a result components classified as Tier 3 are generally not production-ready as they have not been battle-tested much.

Features classified as Tier 3 may be disabled in CI or removed from the repository as well. If a Tier 3 feature is preventing development of other features then the owner will be notified.

If no response is heard from within a week then the feature will be disabled in CI. If no further response happens for a month then the feature may be removed from the repository.

Tier 2 - Almost Production Ready

This tier is meant to encompass features and components of Wasmtime which are well-maintained, tested well, but don't necessarily meet the stringent criteria for Tier 1. Features in this category may already be "production ready" and safe to use.

Tier 2 features include:

- Tests are run in CI for the Wasmtime project for this feature and everything passes. For example a Tier 2 platform runs in CI directly or via emulation. Features are otherwise fully tested on CI.
- Complete implementations for anything that's part of Tier 1. For example all Tier 2 targets must implement all of the Tier 1 WebAssembly proposals, and all Tier 2 features must be implemented on all Tier 1 targets.
- All existing developers are expected to handle minor changes which affect Tier 2 components. For example if Cranelift's interfaces change then the developer changing the interface is expected to handle the changes for Tier 2 architectures so long as the affected part is relatively minor. Note that if a more substantial change is required to a Tier 2 component then that falls under the next bullet.
- Maintainers of a Tier 2 feature are responsive (reply to requests within a week) and are available to accommodate architectural changes that affect their component. For example more expansive work beyond the previous bullet where contributors can't easily handle changes are expected to be guided or otherwise implemented by Tier 2 maintainers.
- Major changes otherwise requiring an RFC that affect Tier 2 components are required to consult Tier 2 maintainers in the course of the RFC. Major changes to Tier 2 components themselves do not require an RFC, however.

Features at this tier generally are not turned off or disabled for very long. Maintainers are already required to be responsive to changes and will be notified of any unrelated change which affects their component. It's recommended that if a component breaks for one reason or another due to an unrelated change that the maintainer either contributes to the PR-in-progress or otherwise has a schedule for the implementation of the feature.

Tier 1 - Production Ready

This tier is intended to be the highest level of support in Wasmtime for any particular feature, indicating that it is suitable for production environments. This conveys a high level

of confidence in the Wasmtime project about the specified features.

Tier 1 features include:

- Continuous fuzzing is required for WebAssembly proposals. This means that any WebAssembly proposal must have support in the `wasm-smith` crate and existing fuzz targets must be running and exercising the new code paths. Where possible differential fuzzing should also be implemented to compare results with other implementations.
- Continuous fuzzing is required for the architecture of supported targets. For example currently there are three x86_64 targets that are considered Tier 1 but only `x86_64-unknown-linux-gnu` is fuzzed.
- CVEs and security releases will be performed as necessary for any bugs found in features and targets.
- Major changes affecting this component may require help from maintainers with specialized expertise, but otherwise it should be reasonable to expect most Wasmtime developers to be able to maintain Tier 1 features.
- Major changes affecting Tier 1 features require an RFC and prior agreement on the change before an implementation is committed.
- WebAssembly proposals meet all stabilization requirements.

A major inclusion point for this tier is intended to be the continuous fuzzing of Wasmtime. This implies a significant commitment of resources for fixing issues, hardware to execute Wasmtime, etc. Additionally this tier comes with the broadest expectation of "burden on everyone else" in terms of what changes everyone is generally expected to handle.

Features classified as Tier 1 are rarely, if ever, turned off or removed from Wasmtime.

Platform Support

This page is intended to give a high-level overview of Wasmtime's platform support along with some aspirations of Wasmtime. For more details see the documentation on tiers of stability which has specific information about what's supported in Wasmtime on a per-matrix-combination basis.

Wasmtime strives to support hardware that anyone wants to run WebAssembly on. Maintainers of Wasmtime support a number of "major" platforms themselves but porting work may be required to support platforms that maintainers are not themselves familiar with. Out-of-the box Wasmtime supports:

- Linux x86_64, aarch64, s390x, and riscv64
- macOS x86_64, aarch64
- Windows x86_64

Other platforms such as Android, iOS, and the BSD family of OSes are not built-in yet. PRs for porting are welcome and maintainers are happy to add more entries to the CI matrix for these platforms.

Compiler Support

Cranelift supports x86_64, aarch64, s390x, and riscv64. No 32-bit platform is currently supported. Building a new backend for Cranelift is a relatively large undertaking which maintainers are willing to help with but it's recommended to reach out to Cranelift maintainers first to discuss this.

Winch supports x86_64. The aarch64 backend is in development. Winch is built on Cranelift's support for emitting instructions so Winch's possible backend list is currently limited to what Cranelift supports.

Usage of the Cranelift or Winch requires a host operating system which supports creating executable memory pages on-the-fly. Support for statically linking in a single precompiled module is not supported at this time.

Both Cranelift and Winch can be used either in AOT or JIT mode. In AOT mode one process precompiles a module/component and then loads it into another process. In JIT mode this is all done within the same process.

Neither Cranelift nor Winch support tiering at this time in the sense of having a WebAssembly module start from a Winch compilation and automatically switch to a Cranelift compilation. Modules are either entirely compiled with Winch or Cranelift.

Interpreter support

At this time `wasmtime` does not have a mode in which it simply interprets WebAssembly code. It is desired to add support for an interpreter, however, and this will have minimal system dependencies. It is planned that the system will need to support some form of dynamic memory allocation, but other than that not much else will be needed.

Support for `#![no_std]`

The `wasmtime` crate supports being build on no_std platforms in Rust, but only for a subset of its compile-time Cargo features. Currently supported Cargo features are:

- `runtime`
- `gc`
- `component-model`

This notably does not include the `default` feature which means that when depending on Wasmtime you'll need to specify `default-features = false`. This also notably does not include Cranelift or Winch at this time meaning that no_std platforms must be used in AOT mode where the module is precompiled elsewhere.

Wasmtime's support for no_std requires the embedder to implement the equivalent of a C header file to indicate how to perform basic OS operations such as allocating virtual memory. This API can be found as `wasmtime-platform.h` in Wasmtime's release artifacts or at `examples/min-platform/embedding/wasmtime-platform.h` in the source tree. Note that this API is not guaranteed to be stable at this time, it'll need to be updated when Wasmtime is updated.

Wasmtime's runtime will use the symbols defined in this file meaning that if they're not defined then a link-time error will be generated. Embedders are required to implement these functions in accordance with their documentation to enable Wasmtime to run on custom platforms.

Wasm Proposals

This document is intended to describe the current status of WebAssembly proposals in Wasmtime. For information about implementing a proposal in Wasmtime see the associated documentation.

WebAssembly proposals that want to be tier 2 or above are required to check all boxes in this matrix. An explanation of each matrix column is below.

On-by-default proposals

Proposal	Phase 4	Tests	Finished	Fuzzed	API	C API
`mutable-globals`	✅	✅	✅	✅	✅	✅
`sign-extension-ops`	✅	✅	✅	✅	✅	✅
`nontrapping-fptoint`	✅	✅	✅	✅	✅	✅
`multi-value`	✅	✅	✅	✅	✅	✅
`bulk-memory`	✅	✅	✅	✅	✅	✅
`reference-types`	✅	✅	✅	✅	✅	✅
`simd`	✅	✅	✅	✅	✅	✅
`component-model`	❌[1]	✅	✅	⚠️[2]	✅	❌[3]
`relaxed-simd`	✅	✅	✅	✅	✅	✅
`multi-memory`	✅	✅	✅	✅	✅	✅
`threads`	✅	✅	✅[4]	❌[5]	✅	✅
`tail-call`	✅	✅	✅	✅	✅	✅
`extended-const`	✅	✅	✅	✅	✅	✅

[1] The `component-model` proposal is not at phase 4 in the standardization process but it is still enabled-by-default in Wasmtime.

[2] Various shapes of components are fuzzed but full-on fuzzing along the lines of `wasm-smith` are not implemented for components.

[5] Fuzzing with threads is an open implementation question that is expected to get fleshed out as the `shared-everything-threads` proposal advances.

[3] Support for the C API for components is desired by many embedders but does not currently have anyone lined up to implement it.

[4] There are known issues with shared memories and the implementation/API in Wasmtime, for example they aren't well integrated with resource-limiting features in `Store`. Additionally `shared` memories aren't supported in the pooling allocator.

Off-by-default proposals

Proposal	Phase 4	Tests	Finished	Fuzzed	API	C API
`memory64`	❌	✅	✅	✅	✅	✅
`function-references`	✅	✅	❌	❌	✅	❌
`gc` [6]	✅	✅	❌ [7]	❌	✅	❌
`wide-arithmetic`	❌	✅	✅	✅	✅	✅
`custom-page-sizes`	❌	✅	✅	✅	✅	❌

[6] There is also a tracking issue for the GC proposal.

[7] The implementation of GC has known performance issues which can affect non-GC code when the GC proposal is enabled.

Unimplemented proposals

Proposal	Tracking Issue
`branch-hinting`	#9463
`exception-handling`	#3427
`flexible-vectors`	#9464
`memory-control`	#9467
`stack-switching`	#9465
`shared-everything-threads`	#9466

Feature requirements

For each column in the above tables, this is a further explanation of its meaning:

- **Phase 4** - The proposal must be in phase 4, or greater, of the WebAssembly standardization process.
- **Tests** - All spec tests must be passing in Wasmtime and where appropriate Wasmtime-specific tests, for example for the API, should be passing. Tests must pass at least for Cranelift on all tier 1 platforms, but missing other platforms is otherwise acceptable.
- **Finished** - No open questions, design concerns, or serious known bugs. The implementation should be complete to the extent that is possible. Support must be implemented for all tier 1 targets and compiler backends.
- **Fuzzed** - Has been fuzzed for at least a week minimum. We are also confident that the fuzzers are fully exercising the proposal's functionality. The `module_generation_uses_expected_proposals` test in the `wasmtime-fuzzing` crate must be updated to include this proposal.

  > For example, it would *not* have been enough to simply enable reference types in the `compile` fuzz target to enable that proposal by default. Compiling a module that uses reference types but not instantiating it nor running any of its functions doesn't exercise any of the GC implementation and does not run the inline fast paths for `table` operations emitted by the JIT. Exercising these things was the motivation for writing the custom fuzz target for `table.{get,set}` instructions.

  One indication of the status of fuzzing is this file which controls module configuration during fuzzing.
- **API** - The proposal's functionality is exposed in the `wasmtime` crate's API. At minimum this is `Config::wasm_the_proposal` but proposals such as `gc` also add new types to the API.
- **C API** - The proposal's functionality is exposed in the C API.

Security

One of WebAssembly (and Wasmtime's) main goals is to execute untrusted code in a safe manner inside of a sandbox. WebAssembly is inherently sandboxed by design (must import all functionality, etc). This document is intended to cover the various sandboxing implementation strategies that Wasmtime has as they are developed. This has also been documented in a historical blog post too.

At this time Wasmtime implements what's necessary for the WebAssembly specification, for example memory isolation between instances. Additionally the safe Rust API is intended to mitigate accidental bugs in hosts.

Different sandboxing implementation techniques will also come with different tradeoffs in terms of performance and feature limitations, and Wasmtime plans to offer users choices of which tradeoffs they want to make.

WebAssembly Core

The core WebAssembly spec has several features which create a unique sandboxed environment:

- The callstack is inaccessible. Unlike most native execution environments, return addresses from calls and spilled registers are not stored in memory accessible to applications. They are stored in memory that only the implementation has access to, which makes traditional stack-smashing attacks targeting return addresses impossible.
- Pointers, in source languages which have them, are compiled to offsets into linear memory, so implementations details such as virtual addresses are hidden from applications. And all accesses within linear memory are checked to ensure they stay in bounds.
- All control transfers—direct and indirect branches, as well as direct and indirect calls—are to known and type-checked destinations, so it's not possible to accidentally call into the middle of a function or branch outside of a function.
- All interaction with the outside world is done through imports and exports. There is no raw access to system calls or other forms of I/O; the only thing a WebAssembly instance can do is what is available through interfaces it has been explicitly linked with.
- There is no undefined behavior. Even where the WebAssembly spec permits multiple possible behaviors, it doesn't permit arbitrary behavior.

Defense-in-depth

While WebAssembly is designed to be sandboxed bugs or issues inevitably arise so Wasmtime also implements a number of mitigations which are not required for correct execution of WebAssembly but can help mitigate issues if bugs are found:

- Linear memories by default are preceded with a 2GB guard region. WebAssembly has no means of ever accessing this memory but this can protect against accidental sign-extension bugs in Cranelift where if an offset is accidentally interpreted as a signed 32-bit offset instead of an unsigned offset it could access memory before the addressable memory for WebAssembly.

- Wasmtime uses explicit checks to determine if a WebAssembly function should be considered to stack overflow, but it still uses guard pages on all native thread stacks. These guard pages are never intended to be hit and will abort the program if they're hit. Hitting a guard page within WebAssembly indicates a bug in host configuration or a bug in Cranelift itself.

- Where it can Wasmtime will zero memory used by a WebAssembly instance after it's finished. This is not necessary unless the memory is actually reused for instantiation elsewhere but this is done to prevent accidental leakage of information between instances in the face of other bugs. This applies to linear memories, tables, and the memory used to store instance information itself.

- The choice of implementation language, Rust, for Wasmtime is also a defense in protecting the authors for Wasmtime from themselves in addition to protecting embedders from themselves. Rust helps catch mistakes when writing Wasmtime itself at compile time. Rust additionally enables Wasmtime developers to create an API that means that embedders can't get it wrong. For example it's guaranteed that Wasmtime won't segfault when using its public API, empowering embedders with confidence that even if the embedding has bugs all of the security guarantees of WebAssembly are still upheld.

- Wasmtime is in the process of implementing control-flow-integrity mechanisms to leverage hardware state for further guaranteeing that WebAssembly stays within its sandbox. In the event of a bug in Cranelift this can help mitigate the impact of where control flow can go to.

Filesystem Access

Wasmtime implements the WASI APIs for filesystem access, which follow a capability-based security model, which ensures that applications can only access files and directories they've been given access to. WASI's security model keeps users safe today, and also helps us prepare for shared-nothing linking and nanoprocesses in the future.

Wasmtime developers are intimately engaged with the WASI standards process, libraries, and tooling development, all along the way too.

Terminal Output

If untrusted code is allowed to print text which is displayed to a terminal, it may emit ANSI-style escape sequences and other control sequences which, depending on the terminal the user is using and how it is configured, can have side effects including writing to files, executing commands, injecting text into the stream as if the user had typed it, or reading the output of previous commands. ANSI-style escape sequences can also confuse or mislead users, making other vulnerabilities easier to exploit.

Our first priority is to protect users, so Wasmtime now filters writes to output streams when they are connected to a terminal to translate escape sequences into inert replacement sequences.

Some applications need ANSI-style escape sequences, such as terminal-based editors and programs that use colors, so we are also developing a proposal for the WASI Subgroup for safe and portable ANSI-style escape sequence support, which we hope to post more about soon.

Spectre

Wasmtime implements a few forms of basic spectre mitigations at this time:

- Bounds checks when accessing entries in a function table (e.g. the `call_indirect` instruction) are mitigated.
- The `br_table` instruction is mitigated to ensure that speculation goes to a deterministic location.
- Wasmtime's default configuration for linear memory means that bounds checks will not be present for memory accesses due to the reliance on page faults to instead detect out-of-bounds accesses. When Wasmtime is configured with "dynamic" memories, however, Cranelift will insert spectre mitigation for the bounds checks performed for all memory accesses.

Mitigating Spectre continues to be a subject of ongoing research, and Wasmtime will likely grow more mitigations in the future as well.

Disclosure Policy

The disclosure policy for security issues in Wasmtime is documented on the Bytecode Alliance website.

What is considered a security vulnerability?

If you are still unsure whether an issue you are filing is a security vulnerability or not after reading this page, always err on the side of caution and report it as a security vulnerability!

Bugs must affect a tier 1 platform or feature to be considered a security vulnerability.

The security of the host and integrity of the sandbox when executing Wasm is paramount. Anything that undermines the Wasm execution sandbox is a security vulnerability.

On the other hand, execution that diverges from Wasm semantics (such as computing incorrect values) are not considered security vulnerabilities so long as they remain confined within the sandbox. This has a couple repercussions that are worth highlighting:

- Even though it is safe from the *host's* point of view, an incorrectly computed value could lead to classic memory unsafety bugs from the *Wasm guest's* point of view, such as corruption of its `malloc`'s free list or reading past the end of a source-level array.
- Wasmtime embedders should never blindly trust values from the guest — no matter how trusted the guest program is, even if it was written by the embedders themselves — and should always validate these values before performing unsafe operations on behalf of the guest.

Denials of service when *executing* Wasm (either originating inside compiled Wasm code or Wasmtime's runtime subroutines) are considered security vulnerabilities. For example, if you configure Wasmtime to run Wasm guests with the async fuel mechanism, and then executing the Wasm goes into an infinite loop that never yields, that is considered a security vulnerability.

Denials of service when *compiling* Wasm, however, are not considered security vulnerabilities. For example, an infinite loop during register allocation is not a security vulnerability.

Any kind of memory unsafety (e.g. use-after-free bugs, out-of-bounds memory accesses, etc...) in the host is always a security vulnerability.

Cheat Sheet: Is this bug considered a security vulnerability?

Type of bug	At Wasm Compile Time	At Wasm Execution Time
Sandbox escape	-	Yes

Type of bug	At Wasm Compile Time	At Wasm Execution Time
Uncaught out-of-bounds memory access	-	Yes
Uncaught out-of-bounds table access	-	Yes
Failure to uphold Wasm's control-flow integrity	-	Yes
File system access outside of the WASI file system's mapped directories	-	Yes
Use of a WASI resource without having been given the associated WASI capability	-	Yes
Etc...	-	Yes
Divergence from Wasm semantics (without escaping the sandbox)	-	No
Computing incorrect value	-	No
Raising errant trap	-	No
Etc...	-	No
Memory unsafety	Yes	Yes
Use-after-free	Yes	Yes
Out-of-bounds memory access	Yes	Yes
Use of uninitialized memory	Yes	Yes

Type of bug	At Wasm Compile Time	At Wasm Execution Time
Etc...	Yes	Yes
Denial of service	No	Yes
Panic	No	Yes
Process abort	No	Yes
Uninterruptible infinite loops	No	Yes
User-controlled memory exhaustion	No	Yes
Uncontrolled recursion over user-supplied input	No	Yes
Etc...	No	Yes

Note that we still want to fix every bug mentioned above even if it is not a security vulnerability! We appreciate when issues are filed for non-vulnerability bugs, particularly when they come with test cases and steps to reproduce!

Vulnerability Runbook

This document outlines how Wasmtime maintainers should respond to a security vulnerability found in Wasmtime. This is intended to be a Wasmtime-specific variant of the runbook RFC originally created. More details are available in the RFC in some specific steps.

Vulnerabilities and advisories are all primarily coordinated online through GitHub Advisories on the Wasmtime repository. Anyone can make an advisory on Wasmtime, and once created anyone can be added to an advisory. Once an advisory is created these steps are followed:

1. An **Incident Manager** is selected. By default this is the Wasmtime maintainer that opened the advisory. If a contributor opened the advisory then it's by default the first responder on the advisory. The incident manager can, at any time, explicitly hand off this role to another maintainer.

2. **Fill out the advisory details**. This step involves filling out all the fields on the GitHub Advisory page such as:

    - Description - the description field's initial placeholder has the various sections to fill out. At this point at least a brief description of the impact should be filled out. This will get fleshed out more later too.
    - Affected versions - determine which previously released versions of Wasmtime are affected by this issue.
    - Severity - use the CVSS calculator to determine the severity of this vulnerability.

3. **Collaborate on a fix**. This should be done in a private fork created for the security advisory. This is also when any collaborators who can help with the development of the fix should also be invited. At this time only the `main` branch needs to have a fix.

4. **Finalize vulnerability details and patched versions**. After a fix has been developed and the vulnerability is better understood at this point the description of the advisory should be fully filled out and be made ready to go to the public. This is also when the incident manager should determine the number of versions of Wasmtime to patch. The latest two versions are required, and older versions are optional.

5. **Request a CVE**. Use the Big Green Button on the advisory to request a CVE number from GitHub staff.

6. **Send advanced disclosure email**. The incident manager will decide on a disclosure date, typically no more than a week away, and send mail to sec-announce@bytecodealliance.org about the upcoming security release. An example mail looks like this

7. **Add more stakeholders** (optional). Users interested in getting advanced notice about this vulnerability may respond to the mailing list post. The incident manager will add them to the security advisory.

8. **Prepare PRs for patch releases**. This will involve creating more pull requests in the private fork attached to the advisory. Each version of Wasmtime being patched should have a PR ready-to-go which cleanly applies. Be sure to write release notes on the PR for each release branch.

9. **The full test suite should be run locally for `main`**. Locally try to run as much of the CI matrix as you can. You probably won't be able to run all of it, and that's ok, but try to get the ones that may have common failures. This is required because CI doesn't run on private forks.

10. **Open version bump PRs on the public repository**. Use the [online trigger][ci-trigger] for this workflow to open PRs for all versions that are going to be patched. DO NOT include patch notes or release notes for this fix. Use this time to fix CI by landing PRs to the release branches separate from the version bump PR. DO NOT merge the version bump PR.

11. **Manually make PRs on release day**. DO NOT merge via the security advisory. This has generally not worked well historically because there's too many CI failures and branch protections. On the day of the release make public PRs from all of the previously-created PRs on the private fork. You'll need to push the changes to your own personal repository for this, but that's ok since it's time to make things public anyway. Merge all PRs (including to `main`) once CI passes.

12. **Merge version bump PRs**. Once the fixes have all been merged and CI is green merge all the version bump PRs. That will trigger the automatic release process which will automatically publish to crates.io and publish the release.

13. **Publish the GitHub Advisories**. Delete the private forks and hit that Big Green Button to publish the advisory.

14. **Send mail about the security release**. Send another around of mail to sec-announce@bytecodealliance.org describing the security release. This mail looks like this.

You'll want to pay close attention to CI on release day. There's likely going to be CI failures with the fix for the vulnerability for some build configurations or platforms and such. It should be easy to fix though so mostly try to stay on top of it. Additionally be sure to carefully watch the publish process to crates.io. It's possible to hit rate limits in crate publication which necessitates a retry of the job later. You can also try publishing locally too from the release branch, but it's best to do it through CI.

Contributing

We're excited to work on Wasmtime and/or Cranelift together with you! This guide should help you get up and running with Wasmtime and Cranelift development. But first, make sure you've read the Code of Conduct!

Wasmtime and Cranelift are very ambitious projects with many goals, and while we're confident we can achieve some of them, we see many opportunities for people to get involved and help us achieve even more.

Join Our Chat

We chat about Wasmtime and Cranelift development on Zulip — join us!. You can also join specific streams:

- #wasmtime
- #cranelift

If you're having trouble building Wasmtime or Cranelift, aren't sure why a test is failing, or have any other questions, feel free to ask on Zulip. Not everything we hope to do with these projects is reflected in the code or documentation yet, so if you see things that seem missing or that don't make sense, or even that just don't work the way you expect them to, we're also interested to hear about that!

As always, you're more than welcome to open an issue too!

Finally, we have biweekly project meetings, hosted on Zoom, for Wasmtime and Cranelift. For more information, see our meetings agendas/minutes repository. Please feel free to contact us via Zulip if you're interested in joining!

Finding Something to Hack On

If you're looking for something to do, these are great places to start:

- Issues labeled "good first issue" — these issues tend to be simple, what needs to be done is well known, and are good for new contributors to tackle. The goal is to learn Wasmtime's development workflow and make sure that you can build and test Wasmtime.

- Issues labeled "help wanted" — these are issues that we need a little help with!

If you're unsure if an issue is a good fit for you or not, feel free to ask in a comment on the issue, or in chat.

Mentoring

We're happy to mentor people, whether you're learning Rust, learning about compiler backends, learning about machine code, learning about wasm, learning about how Cranelift does things, or all together at once.

We categorize issues in the issue tracker using a tag scheme inspired by Rust's issue tags. For example, the E-easy marks good beginner issues, and E-rust marks issues which likely require some familiarity with Rust, though not necessarily Cranelift-specific or even compiler-specific experience. E-compiler-easy marks issues good for beginners who have some familiarity with compilers, or are interested in gaining some :-).

See also the full list of Cranelift labels.

Also, we encourage people to just look around and find things they're interested in. This a good time to get involved, as there aren't a lot of things set in stone yet.

Architecture of Wasmtime

This document is intended to give an overview of the implementation of Wasmtime. This will explain the purposes of the various `wasmtime-*` crates that the main `wasmtime` crate depends on. For even more detailed information it's recommended to review the code itself and find the comments contained within.

The `wasmtime` crate

The main entry point for Wasmtime is the `wasmtime` crate itself. Wasmtime is designed such that the `wasmtime` crate is nearly a 100% safe API (safe in the Rust sense) modulo some small and well-documented functions as to why they're `unsafe`. The `wasmtime` crate provides features and access to WebAssembly primitives and functionality, such as compiling modules, instantiating them, calling functions, etc.

At this time the `wasmtime` crate is the first crate that is intended to be consumed by users. First in this sense means that everything `wasmtime` depends on is thought of as an internal dependency. We publish crates to crates.io but put very little effort into having a "nice" API for internal crates or worrying about breakage between versions of internal crates. This primarily means that all the other crates discussed here are considered internal dependencies of Wasmtime and don't show up in the public API of Wasmtime at all. To use some Cargo terminology, all the `wasmtime-*` crates that `wasmtime` depends on are "private" dependencies.

Additionally at this time the safe/unsafe boundary between Wasmtime's internal crates is not the most well-defined. There are methods that should be marked `unsafe` which aren't, and `unsafe` methods do not have exhaustive documentation as to why they are `unsafe`. This is an ongoing matter of improvement, however, where the goal is to have safe methods be actually safe in the Rust sense, as well as having documentation for `unsafe` methods which clearly lists why they are `unsafe`.

Important concepts

To preface discussion of more nitty-gritty internals, it's important to have a few concepts in the back of your head. These are some of the important types and their implications in Wasmtime:

- `wasmtime::Engine` - this is a global compilation context which is sort of the "root context". An `Engine` is typically created once per program and is expected to be shared across many threads (internally it's atomically reference counted). Each `Engine`

stores configuration values and other cross-thread data such as type interning for `Module` instances. The main thing to remember for `Engine` is that any mutation of its internals typically involves acquiring a lock, whereas for `Store` below no locks are necessary.

- `wasmtime::Store` - this is the concept of a "store" in WebAssembly. While there's also a formal definition to go off of, it can be thought of as a bag of related WebAssembly objects. This includes instances, globals, memories, tables, etc. A `Store` does not implement any form of garbage collection of the internal items (there is a `gc` function but that's just for `externref` values). This means that once you create an `Instance` or a `Table` the memory is not actually released until the `Store` itself is deallocated. A `Store` is sort of a "context" used for almost all wasm operations. `Store` also contains instance handles which recursively refer back to the `Store`, leading to a good bit of aliasing of pointers within the `Store`. The important thing for now, though, is to know that `Store` is a unit of isolation. WebAssembly objects are always entirely contained within a `Store`, and at this time nothing can cross between stores (except scalars if you manually hook it up). In other words, wasm objects from different stores cannot interact with each other. A `Store` cannot be used simultaneously from multiple threads (almost all operations require `&mut self`).

- `wasmtime::runtime::vm::InstanceHandle` - this is the low-level representation of a WebAssembly instance. At the same time this is also used as the representation for all host-defined objects. For example if you call `wasmtime::Memory::new` it'll create an `InstanceHandle` under the hood. This is a very `unsafe` type that should probably have all of its functions marked `unsafe` or otherwise have more strict guarantees documented about it, but it's an internal type that we don't put much thought into for public consumption at this time. An `InstanceHandle` doesn't know how to deallocate itself and relies on the caller to manage its memory. Currently this is either allocated on-demand (with `malloc`) or in a pooling fashion (using the pooling allocator). The `deallocate` method is different in these two paths (as well as the `allocate` method).

  An `InstanceHandle` is laid out in memory with some Rust-owned values first capturing the dynamic state of memories/tables/etc. Most of these fields are unused for host-defined objects that serve one purpose (e.g. a `wasmtime::Table::new`), but for an instantiated WebAssembly module these fields will have more information. After an `InstanceHandle` in memory is a `VMContext`, which will be discussed next. `InstanceHandle` values are the main internal runtime representation and what the `crate::runtime::vm` code works with. The `wasmtime::Store` holds onto all these `InstanceHandle` values and deallocates them at the appropriate time. From the runtime perspective it simplifies things so the graph of wasm modules communicating to each other is reduced to simply `InstanceHandle` values all talking to themselves.

- `crate::runtime::vm::VMContext` - this is a raw pointer, within an allocation of an `InstanceHandle`, that is passed around in JIT code. A `VMContext` does not have a structure defined in Rust (it's a 0-sized structure) because its contents are dynamically

determined based on the `VMOffsets`, or the source wasm module it came from. Each `InstanceHandle` has a "shape" of a `VMContext` corresponding with it. For example a `VMContext` stores all values of WebAssembly globals, but if a wasm module has no globals then the size of this array will be 0 and it won't be allocated. The intention of a `VMContext` is to be an efficient in-memory representation of all wasm module state that JIT code may access. The layout of `VMContext` is dynamically determined by a module and JIT code is specialized for this one structure. This means that the structure is efficiently accessed by JIT code, but less efficiently accessed by native host code. A non-exhaustive list of purposes of the `VMContext` is to:

  - Store WebAssembly instance state such as global values, pointers to tables, pointers to memory, and pointers to other JIT functions.
  - Separate wasm imports and local state. Imported values have pointers stored to their actual values, and local state has the state defined inline.
  - Hold a pointer to the stack limit at which point JIT code will trigger a stack overflow.
  - Hold a pointer to a `VMExternRefActivationsTable` for fast-path insertion of `externref` values into the table.
  - Hold a pointer to a `*mut dyn crate::runtime::vm::Store` so store-level operations can be performed in libcalls.

  A comment about the layout of a `VMContext` can be found in the `vmoffsets.rs` file.

- `wasmtime::Module` - this is the representation of a compiled WebAssembly module. At this time Wasmtime always assumes that a wasm module is always compiled to native JIT code. `Module` holds the results of said compilation, and currently Cranelift can be used for compiling. It is a goal of Wasmtime to support other modes of representing modules but those are not implemented today just yet, only Cranelift is implemented and supported.

- `wasmtime_environ::Module` - this is a descriptor of a wasm module's type and structure without holding any actual JIT code. An instance of this type is created very early on in the compilation process, and it is not modified when functions themselves are actually compiled. This holds the internal type representation and state about functions, globals, etc. In a sense this can be thought of as the result of validation or typechecking a wasm module, although it doesn't have information such as the types of each opcode or minute function-level details like that.

Compiling a module

With a high-level overview and some background information of types, this will next walk through the steps taken to compile a WebAssembly module. The main entry point for this is the `wasmtime::Module::from_binary` API. There are a number of other entry points that

deal with surface-level details like translation from text-to-binary, loading from the filesystem, etc.

Compilation is roughly broken down into a few phases:

1. First compilation walks over the WebAssembly module validating everything except function bodies. This synchronous pass over a wasm module creates a `wasmtime_environ::Module` instance and additionally prepares for function compilation. Note that with the module linking proposal one input module may end up creating a number of output modules to process. Each module is processed independently and all further steps are parallelized on a per-module basis. Note that parsing and validation of the WebAssembly module happens with the `wasmparser` crate. Validation is interleaved with parsing, validating parsed values before using them.

2. Next all functions within a module are validated and compiled in parallel. No inter-procedural analysis is done and each function is compiled as its own little island of code at this time. This is the point where the meat of Cranelift is invoked on a per-function basis.

3. The compilation results at this point are all woven into a `wasmtime_jit::CompilationArtifacts` structure. This holds module information (`wasmtime_environ::Module`), compiled JIT code (stored as an ELF image), and miscellaneous other information about functions such as platform-agnostic unwinding information, per-function trap tables (indicating which JIT instructions can trap and what the trap means), per-function address maps (mapping from JIT addresses back to wasm offsets), and debug information (parsed from DWARF information in the wasm module). These results are inert and can't actually be executed, but they're appropriate at this point to serialize to disk or begin the next phase...

4. The final step is to actually place all code into a form that's ready to get executed. This starts from the `CompilationArtifacts` of the previous step. Here a new memory mapping is allocated and the JIT code is copied into this memory mapping. This memory mapping is then switched from read/write to read/execute so it's actually executable JIT code at this point. This is where various hooks like loading debuginfo, informing JIT profilers of new code, etc, all happens. At this point a `wasmtime_jit::CompiledModule` is produced and this is itself wrapped up in a `wasmtime::Module`. At this point the module is ready to be instantiated.

A `wasmtime::Module` is an atomically-reference-counted object where upon instantiation into a `Store`, the `Store` will hold a strong reference to the internals of the module. This means that all instances of a `wasmtime::Module` share the same compiled code. Additionally a `wasmtime::Module` is one of the few objects that lives outside of a `wasmtime::Store`. This means that `wasmtime::Module`'s reference counting is its own form of memory management.

Note that the property of sharing a module's compiled code across all instantiations has interesting implications on what the compiled code can assume. For example Wasmtime implements a form of type interning, but the interned types happen at a few different levels. Within a module we deduplicate function types, but across modules in a `Store` types need to be represented with the same value. This means that if the same module is instantiated into many stores its same function type may take on many values, so the compiled code can't assume a particular value for a function type. (more on type information later). The general gist though is that compiled code leans relatively heavily on the `VMContext` for contextual input because the JIT code is intended to be so widely reusable.

Trampolines

An important aspect to also cover for compilation is the creation of trampolines. Trampolines in this case refer to code executed by Wasmtime to enter WebAssembly code. The host may not always have prior knowledge about the signature of the WebAssembly function that it wants to call. Wasmtime JIT code is compiled with native ABIs (e.g. params/results in registers according to System V on Unix), which means that a Wasmtime embedding doesn't have an easy way to enter JIT code.

This problem is what the trampolines compiled into a module solve, which is to provide a function with a known ABI that will call into a function with a specific other type signature/ABI. Wasmtime collects all the exported functions of a module and creates a set of their type signatures. Note that exported in this context actually means "possibly exported" which includes things like insertion into a global/function table, conversion to a `funcref`, etc. A trampoline is generated for each of these type signatures and stored along with the JIT code for the rest of the module.

These trampolines are then used with the `wasmtime::Func::call` API where in that specific case because we don't know the ABI of the target function the trampoline (with a known ABI) is used and has all the parameters/results passed through the stack.

Another point of note is that trampolines are not deduplicated at this time. Each compiled module contains its own set of trampolines, and if two compiled modules have the same types then they'll have different copies of the same trampoline.

Type Interning and `VMSharedSignatureIndex`

One important point to talk about with compilation is the `VMSharedSignatureIndex` type and how it's used. The `call_indirect` opcode in wasm compares an actual function's signature against the function signature of the instruction, trapping if the signatures mismatch. This is implemented in Wasmtime as an integer comparison, and the comparison happens on a `VMSharedSignatureIndex` value. This index is an intern'd representation of a function type.

The scope of interning for `VMSharedSignatureIndex` happens at the `wasmtime::Engine` level. Modules are compiled into an `Engine`. Insertion of a `Module` into an `Engine` will assign a `VMSharedSignatureIndex` to all of the types found within the module.

The `VMSharedSignatureIndex` values for a module are local to that one instantiation of a `Module` (and they may change on each insertion of a `Module` into a different `Engine`). These are used during the instantiation process by the runtime to assign a type ID effectively to all functions for imports and such.

Instantiating a module

Once a module has been compiled it's typically then instantiated to actually get access to the exports and call wasm code. Instantiation always happens within a `wasmtime::Store` and the created instance (plus all exports) are tied to the `Store`.

Instantiation itself (`crates/wasmtime/src/instance.rs`) may look complicated, but this is primarily due to the implementation of the Module Linking proposal. The rough flow of instantiation looks like:

1. First all imports are type-checked. The provided list of imports is cross-referenced with the list of imports recorded in the `wasmtime_environ::Module` and all types are verified to line up and match (according to the core wasm specification's definition of type matching).

2. Each `wasmtime_environ::Module` has a list of initializers that need to be completed before instantiation is finished. For MVP wasm this only involves loading the import into the correct index array, but for module linking this could involve instantiating other modules, handling `alias` fields, etc. In any case the result of this step is a `crate::runtime::vm::Imports` array which has the values for all imported items into the wasm module. Note that in this case an import is typically some sort of raw pointer to the actual state plus the `VMContext` of the instance that was imported from. The final result of this step is an `InstanceAllocationRequest`, which is then submitted to the configured instance allocator, either on-demand or pooling.

3. The `InstanceHandle` corresponding to this instance is allocated. How this is allocated depends on the strategy (malloc for on-demand, slab allocation for pooling). In addition to initialization of the fields of `InstanceHandle` this also initializes all the fields of the `VMContext` for this handle (which as mentioned above is adjacent to the `InstanceHandle` allocation after it in memory). This does not process any data segments, element segments, or the `start` function at this time.

4. At this point the `InstanceHandle` is stored within the `Store`. This is the "point of no return" where the handle must be kept alive for the same lifetime as the `Store` itself. If an initialization step fails then the instance may still have had its functions, for

example, inserted into an imported table via an element segment. This means that even if we fail to initialize this instance its state could still be visible to other instances/objects so we need to keep it alive regardless.

5. The final step is performing wasm-defined instantiation. This involves processing element segments, data segments, the `start` function, etc. Most of this is just translating from Wasmtime's internal representation to the specification's required behavior.

Another part worth pointing out for instantiating a module is that a `ModuleRegistry` is maintained within a `Store` of all instantiated modules into the store. The purpose of this registry is to retain a strong reference to items in the module needed to run instances. This includes the JIT code primarily but also has information such as the `VMSharedSignatureIndex` registration, metadata about function addresses and such, etc. Much of this data is stored into a `GLOBAL_MODULES` map for later access during traps.

Traps

Once instances have been created and wasm starts running most things are fairly standard. Trampolines are used to enter wasm (or we can enter with a known ABI if using `wasmtime::TypedFunc`) and JIT code generally does what it does to execute wasm. An important aspect of the implementation to cover, however, is traps.

Wasmtime today implements traps with `longjmp` and `setjmp`. The `setjmp` function cannot be defined in Rust (even unsafely -- (https://github.com/rust-lang/rfcs/issues/2625) so the `crates/wasmtime/src/runtime/vm/helpers.c` file actually calls setjmp/longjmp. Note that in general the operation of `longjmp` is not safe to execute in Rust because it skips stack-based destructors, so after `setjmp` when we call back into Rust to execute wasm we need to be careful in Wasmtime to not have any significant destructors on the stack once wasm is called.

Traps can happen from a few different sources:

- Explicit traps - these can happen when a host call returns a trap, for example. These bottom out in `raise_user_trap` or `raise_lib_trap`, both of which immediately call `longjmp` to go back to the wasm starting point. Note that these, like when calling wasm, have to have callers be very careful to not have any destructors on the stack.

- Signals - this is the main vector for trap. Basically we use segfault and illegal instructions to implement traps in wasm code itself. Segfaults arise when linear memory accesses go out of bounds and illegal instructions are how the wasm `unreachable` instruction is implemented. In both of these cases Wasmtime installs a platform-specific signal handler to catch the signal, inspect the state of the signal, and then handle it. Note that Wasmtime tries to only catch signals that happen from JIT

code itself as to not accidentally cover up other bugs. Exiting a signal handler happens via `longjmp` to get back to the original wasm call-site.

The general idea is that Wasmtime has very tight control over the stack frames of wasm (naturally via Cranelift) and also very tight control over the code that executes just before we enter wasm (aka before the `setjmp`) and just after we reenter back into wasm (aka frames before a possible `longjmp`).

The signal handler for Wasmtime uses the `GLOBAL_MODULES` map populated during instantiation to determine whether a program counter that triggered a signal is indeed a valid wasm trap. This should be true except for cases where the host program has another bug that triggered the signal.

A final note worth mentioning is that Wasmtime uses the Rust `backtrace` crate to capture a stack trace when a wasm exception occurs. This forces Wasmtime to generate native platform-specific unwinding information to correctly unwind the stack and generate a stack trace for wasm code. This does have other benefits as well such as improving generic sampling profilers when used with Wasmtime.

Linear Memory

Linear memory in Wasmtime is implemented effectively with `mmap` (or the platform equivalent thereof), but there are some subtle nuances that are worth pointing out here too. The implementation of linear memory is relatively configurable which gives rise to a number of situations that both the runtime and generated code need to handle.

First there are a number of properties about linear memory which can be configured:

- `wasmtime::Config::static_memory_maximum_size`
- `wasmtime::Config::static_memory_guard_size`
- `wasmtime::Config::dynamic_memory_guard_size`
- `wasmtime::Config::guard_before_linear_memory`

The methods on `Config` have a good bit of documentation to go over some nitty-gritty, but the general gist is that Wasmtime has two modes of memory: static and dynamic. Static memories represent an address space reservation that never moves and pages are committed to represent memory growth. Dynamic memories represent allocations where the committed portion exactly matches the wasm memory's size and growth happens by allocating a bigger chunk of memory.

The guard size configuration indicates the size of the guard region that happens after linear memory. This guard size affects whether generated JIT code emits bounds checks or not. Bounds checks are elided if out-of-bounds addresses provably encounter the guard pages.

The `guard_before_linear_memory` configuration additionally places guard pages in front of linear memory as well as after linear memory (the same size on both ends). This is only used to protect against possible Cranelift bugs and otherwise serves no purpose.

The defaults for Wasmtime on 64-bit platforms are:

- 4GB static maximum size meaning all 32-bit memories are static and 64-bit memories are dynamic.
- 2GB static guard size meaning all loads/stores with less than 2GB offset don't need bounds checks with 32-bit memories.
- Guard pages before linear memory are enabled.

Altogether this means that 32-bit linear memories result in an 8GB virtual address space reservation by default in Wasmtime. With the pooling allocator where we know that linear memories are contiguous this results in a 6GB reservation per memory because the guard region after one memory is the guard region before the next.

Note that 64-bit memories (the memory64 proposal for WebAssembly) can be configured to be static but will never be able to elide bounds checks at this time. This configuration is possible through the `static_memory_forced` configuration option. Additionally note that support for 64-bit memories in Wasmtime is functional but not yet tuned at this time so there's probably still some performance work and better defaults to manage.

Tables and `externref`

WebAssembly tables contain reference types, currently either `funcref` or `externref`. A `funcref` in Wasmtime is represented as `*mut VMCallerCheckedFuncRef` and an `externref` is represented as `VMExternRef` (which is internally `*mut VMExternData`). Tables are consequently represented as vectors of pointers. Table storage memory management by default goes through Rust's `Vec` which uses `malloc` and friends for memory. With the pooling allocator this uses preallocated memory for storage.

As mentioned previously `Store` has no form of internal garbage collection for wasm objects themselves so a `funcref` table in wasm is pretty simple in that there's no lifetime management of any of the pointers stored within, they're simply assumed to be valid for as long as the table is in use.

For tables of `externref` the story is more complicated. The `VMExternRef` is a version of `Arc<dyn Any>` but specialized in Wasmtime so JIT code knows where the offset of the reference count field to directly manipulate it is. Furthermore tables of `externref` values need to manage the reference count field themselves, since the pointer stored in the table is required to have a strong reference count allocated to it.

GC and `externref`

Wasmtime implements the `externref` type of WebAssembly with an atomically-reference-counted pointer. Note that the atomic part is not needed by wasm itself but rather from the Rust embedding environment where it must be safe to send `ExternRef` values to other threads. Wasmtime also does not come with a cycle collector so cycles of host-allocated `ExternRef` objects will leak.

Despite reference counting, though, a `Store::gc` method exists. This is an implementation detail of how reference counts are managed while wasm code is executing. Instead of managing the reference count of an `externref` value individually as it moves around on the stack Wasmtime implements "deferred reference counting" where there's an overly conservative list of `ExternRef` values that may be in use, and periodically a GC is performed to make this overly conservative list a precise one. This leverages the stack map support of Cranelift plus the backtracing support of `backtrace` to determine live roots on the stack. The `Store::gc` method forces the possibly-overly-conservative list to become a precise list of `externref` values that are actively in use on the stack.

Index of crates

The main Wasmtime internal crates are:

- `wasmtime` - the safe public API of `wasmtime`.
    - `wasmtime::runtime::vm` - low-level runtime implementation of Wasmtime. This is where `VMContext` and `InstanceHandle` live. This module used to be a crate, but has since been folding into `wasmtime`.
- `wasmtime-environ` - low-level compilation support. This is where translation of the `Module` and its environment happens, although no compilation actually happens in this crate (although it defines an interface for compilers). The results of this crate are handed off to other crates for actual compilation.
- `wasmtime-cranelift` - implementation of function-level compilation using Cranelift.

Note that at this time Cranelift is a required dependency of wasmtime. Most of the types exported from `wasmtime-environ` use cranelift types in their API. One day it's a goal, though, to remove the required cranelift dependency and have `wasmtime-environ` be a relatively standalone crate.

In addition to the above crates there are some other miscellaneous crates that `wasmtime` depends on:

- `wasmtime-cache` - optional dependency to manage default caches on the filesystem. This is enabled in the CLI by default but not enabled in the `wasmtime` crate by default.

- `wasmtime-fiber` - implementation of stack-switching used by `async` support in Wasmtime
- `wasmtime-debug` - implementation of mapping wasm dwarf debug information to native dwarf debug information.
- `wasmtime-profiling` - implementation of hooking up generated JIT code to standard profiling runtimes.
- `wasmtime-obj` - implementation of creating an ELF image from compiled functions.

Building

This section describes everything required to build and run Wasmtime.

Prerequisites

Before we can actually build Wasmtime, we'll need to make sure these things are installed first.

Git Submodules

The Wasmtime repository contains a number of git submodules. To build Wasmtime and most other crates in the repository, you have to ensure that those are initialized with this command:

```
git submodule update --init
```

The Rust Toolchain

Install the Rust toolchain here. This includes `rustup`, `cargo`, `rustc`, etc...

`libclang` (optional)

The `wasmtime-fuzzing` crate transitively depends on `bindgen`, which requires that your system has a `libclang` installed. Therefore, if you want to hack on Wasmtime's fuzzing infrastructure, you'll need `libclang`. Details on how to get `libclang` and make it available for `bindgen` are here.

Building the `wasmtime` CLI

To make an unoptimized, debug build of the `wasmtime` CLI tool, go to the root of the repository and run this command:

```
cargo build
```

The built executable will be located at `target/debug/wasmtime`.

To make an optimized build, run this command in the root of the repository:

```
cargo build --release
```

The built executable will be located at `target/release/wasmtime`.

You can also build and run a local `wasmtime` CLI by replacing `cargo build` with `cargo run`.

Building the Wasmtime C API

See `crates/c-api/README.md` for details.

Building Other Wasmtime Crates

You can build any of the Wasmtime crates by appending `-p wasmtime-whatever` to the `cargo build` invocation. For example, to build the `wasmtime-environ` crate, execute this command:

```
cargo build -p wasmtime-environ
```

Alternatively, you can `cd` into the crate's directory, and run `cargo build` there, without needing to supply the `-p` flag:

```
cd crates/environ/
cargo build
```

Testing

This section describes how to run Wasmtime's tests and add new tests.

Before continuing, make sure you can build Wasmtime successfully. Can't run the tests if you can't build it!

Installing `wasm32` Targets

To compile the tests, you'll need the `wasm32-wasip1` and `wasm32-unknown-unknown` targets installed, which, assuming you're using rustup.rs to manage your Rust versions, can be done as follows:

```
rustup target add wasm32-wasip1 wasm32-unknown-unknown
```

Running Tests

Depending on what you're modifying there's a few commands you may be the most interested:

- `cargo test` - used to run the `tests/*` folder at the top-level. This tests the CLI and contains most tests for the `wasmtime` crate itself. This will also run all spec tests. Note that this does not run all tests in the repository, but it's generally a good starting point.
- `cargo test -p cranelift-tools` - used if you're working on Cranelift and this will run all the tests at `cranelift/filetests/filetests`. You can also, within the `cranelift` folder, run `cargo run test ./filetests` to run these tests.
- `cargo test -p wasmtime-wasi` - this will run all WASI tests for the `wasmtime-wasi` crate.

At this time not all of the crates in the Wasmtime workspace can be tested, so running all tests is a little non-standard. To match what CI does and run all tests you'll need to execute

```
./ci/run-tests.sh
```

Testing a Specific Crate

You can test a particular Wasmtime crate with `cargo test -p wasmtime-whatever`. For example, to test the `wasmtime-environ` crate, execute this command:

```
cargo test -p wasmtime-environ
```

Alternatively, you can `cd` into the crate's directory, and run `cargo test` there, without needing to supply the `-p` flag:

```
cd crates/environ/
cargo test
```

Running the Wasm Spec Tests

The spec testsuite itself is in a git submodule, so make sure you've checked it out and initialized its submodule:

```
git submodule update --init
```

When the submodule is checked out, Wasmtime runs the Wasm spec testsuite as part of testing the `wasmtime-cli` crate at the crate root, meaning in the root of the repository you can execute:

```
cargo test --test wast
```

You can pass an additional CLI argument to act as a filter on which tests to run. For example to only run the spec tests themselves (excluding handwritten Wasmtime-specific tests) and only in Cranelift you can run:

```
cargo test --test wast Cranelift/tests/spec
```

Note that in general spec tests are executed regardless of whether they pass or not. In `tests/wast.rs` there's a `should_fail` function which indicates the expected result of the test. When adding new spec tests or implementing features this function will need to be updated as tests change from failing to passing.

Running WASI Integration Tests

WASI integration tests can be run separately from all other tests which can be useful when working on the `wasmtime-wasi` crate. This can be done by executing this command:

```
cargo test -p wasmtime-wasi
```

Similarly if you're testing HTTP-related functionality you can execute:

```
cargo test -p wasmtime-wasi-http
```

Note that these tests will compile programs in `crates/test-programs` to run.

Adding New Tests

Adding Rust's `#[test]`-Style Tests

For very "unit-y" tests, we add `test` modules in the same `.rs` file as the code that is being tested. These `test` modules are configured to only get compiled during testing with `#[cfg(test)]`.

```
// some code...

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn some_test_for_that_code() {
        // ...
    }
}
```

If you're writing a unit test and a `test` module doesn't already exist, you can create one.

For more "integration-y" tests, each crate supports a separate `tests` directory within the crate, and put the tests inside there. Most integration tests in Wasmtime are located in the root `tests/*.rs` location, notably `tests/all/*.rs`. This tests much of the `wasmtime` crate for example and facilitates `cargo test` at the repository root running most tests.

Some tests make more sense to live per-crate, though. For example, many WASI tests are at `crates/wasi/tests/*.rs`. For adding a test feel free to add it wherever feels best, there's not really a strong reason to put it in one place over another. While it's easiest to add to existing tests it's ok to add a new `tests` directory with tests too.

Adding Specification-Style Wast Tests

We use the spec testsuite as-is and without custom patches or a forked version via a submodule at `tests/spec_testsuite`. This probably isn't what you want to modify when adding a new Wasmtime test!

When you have a Wasmtime-specific test that you'd like to write in Wast and use the Wast-style assertions, you can add it to our "misc testsuite". The misc testsuite uses the same syntax and assertions as the spec testsuite, but lives in `tests/misc_testsuite`. Feel free to add new tests to existing `tests/misc_testsuite/*.wast` files or create new ones as needed. These tests are run from the crate root:

```
cargo test --test wast
```

If you have a new test that you think really belongs in the spec testsuite, make sure it makes sense for every Wasm implementation to run your test (i.e. it isn't Wasmtime-specific) and send a pull request upstream. Once it is accepted in the upstream repo, it'll make its way to the test-specific mirror at WebAssembly/testsuite and then we can update our git submodule and we'll start running the new tests.

Adding WASI Integration Tests

When you have a WASI-specific test program that you'd like to include as a test case to run against our WASI implementation, you can add it to our `test-programs` crate. In particular, you should drop a main-style Rust source file into `crates/test-programs/src/bin/PREFIX_some_new_test.rs`. Here the `PREFIX` indicates what test suite it's going to run as. For example `preview2_*` tests are run as part of `wasmtime-wasi` crate tests. The `cli_*` tests are run as part of `tests/all/cli_tests.rs`. It's probably easiest to use a preexisting prefix. The `some_new_test` name is arbitrary and is selected as appropriate by you.

One a test file is added you'll need to add some code to execute the tests as well. For example if you added a new test `crates/test-programs/src/bin/cli_my_test.rs` then you'll need to add a new function to `tests/all/cli_tests.rs` such as:

```
#[test]
fn my_test() {
    // ...
}
```

The path to the compiled WebAssembly of your test case will be available in a Rust-level `const` named `CLI_MY_TEST`. There is also a component version at `CLI_MY_TEST_COMPONENT`. These are used to pass as arguments to `wasmtime`-the-CLI for example or you can use `Module::from_file`.

When in doubt feel free to copy existing tests and then modify them to suit your needs.

Fuzzing

Test Case Generators and Oracles

Test case generators and oracles live in the `wasmtime-fuzzing` crate, located in the `crates/fuzzing` directory.

A *test case generator* takes raw, unstructured input from a fuzzer and translates that into a test case. This might involve interpreting the raw input as "DNA" or pre-determined choices through a decision tree and using it to generate an in-memory data structure, or it might be a no-op where we interpret the raw bytes as if they were Wasm.

An *oracle* takes a test case and determines whether we have a bug. For example, one of the simplest oracles is to take a Wasm binary as an input test case, validate and instantiate it, and (implicitly) check that no assertions failed or segfaults happened. A more complicated oracle might compare the result of executing a Wasm file with and without optimizations enabled, and make sure that the two executions are observably identical.

Our test case generators and oracles strive to be fuzzer-agnostic: they can be reused with libFuzzer or AFL or any other fuzzing engine or driver.

libFuzzer and `cargo fuzz` Fuzz Targets

We combine a test case generator and one more more oracles into a *fuzz target*. Because the target needs to pipe the raw input from a fuzzer into the test case generator, it is specific to a particular fuzzer. This is generally fine, since they're only a couple of lines of glue code.

Currently, all of our fuzz targets are written for libFuzzer and `cargo fuzz`. They are defined in the `fuzz` subdirectory.

See `fuzz/README.md` for details on how to run these fuzz targets and set up a corpus of seed inputs.

Continuous Integration (CI)

The Wasmtime and Cranelift projects heavily rely on Continuous Integration (CI) to ensure everything keeps working and keep the final end state of the code at consistently high quality. The CI setup for this repository is relatively involved and extensive, and so it's worth covering here how it's organized and what's expected of contributors.

All CI currently happens on GitHub Actions and is configured in the `.github` directory of the repository.

PRs and CI

Currently on sample of the full CI test suite is run on every Pull Request. CI on PRs is intended to be relatively quick and catch the majority of mistakes and errors. By default the test suite is run on x86_64 Linux but this may change depending on what files the PR is modifying. The intention is to run "mostly relevant" CI on a PR by default.

PR authors are expected to fix CI failures in their PR, unless the CI failure is systemic and unrelated to the PR. In that case other maintainers should be alerted to ensure that the problem can be addressed. Some reviewers may also wait to perform a review until CI is green on the PR as otherwise it may indicate changes are needed.

The Wasmtime repository uses GitHub's Merge Queue feature to merge PRs which. Entry in to the merge queue requires green CI on the PR beforehand. Maintainers who have approved a PR will flag it for entry into the merge queue, and the PR will automatically enter the merge queue once CI is green.

When entering the merge queue a PR will have the full test suite executed which may include tests that weren't previously run on the PR. This may surface new failures, and contributors are expected to fix these failures as well.

To force PRs to execute the full test suite, which takes longer than the default test suite for PRs, then contributors can place the string "prtest:full" somewhere in any commit of the PR. From that point on the PR will automatically run the full test suite as-if it were in the merge queue. Note that when going through the merge queue this will rerun tests.

Tests run on CI

While this may not be fully exhaustive, the general idea of all the checks we run on CI looks like this:

- Code formatting - we run `cargo fmt -- --check` on CI to ensure that all code in the repository is formatted with rustfmt. All PRs are expected to be formatted with the latest stable version of rustfmt.
- Book documentation tests - code snippets (Rust ones at least) in the book documentation (the `docs` folder) are tested on CI to ensure they are working.
- Crate tests - the moral equivalent of `cargo test --all` and `cargo test --all --release` is executed on CI. This means that all workspace crates have their entire test suite run, documentation tests and all, in both debug and release mode. Additionally we execute all crate tests on macOS, Windows, and Linux, to ensure that everything works on all the platforms.
- Fuzz regression tests - we take a random sampling of the fuzz corpus and run it through the fuzzers. This is mostly intended to be a pretty quick regression test and testing the fuzzers still build, most of our fuzzing happens on oss-fuzz. Found issues are recorded in the oss-fuzz bug tracker

While we do run more tests here and there, this is the general shape of what you can be expected to get tested on CI for all commits and all PRs. You can of course always feel free to expand our CI coverage by editing the CI files themselves, we always like to run more tests!

Artifacts produced on CI

Our CI system is also responsible for producing all binary releases and documentation of Wasmtime and Cranelift. Currently this consists of:

- Tarballs of the `wasmtime` CLI - produced for macOS, Windows, and Linux we try to make these "binary compatible" wherever possible, for example producing the Linux build in a really old CentOS container to have a very low glibc requirement.
- Tarballs of the Wasmtime C API - produced for the same set of platforms as the CLI above.
- Book and API documentation - the book is rendered with `mdbook` and we also build all documentation with `cargo doc`.
- A source code tarball which is entirely self-contained. This source tarball has all dependencies vendored so the network is not needed to build it.
- WebAssembly adapters for the component model to translate `wasi_snapshot_preview1` to WASI Preview 2.

Artifacts are produced as part of the full CI suite. This means that artifacts are not produced on a PR by default but can be requested via "prtest:full". All runs through the merge queue

though, which means all merges to `main`, will produce a full suite of artifacts. The latest artifacts are available through Wasmtime's `dev` release and downloads are also available for recent CI runs through the CI page in GitHub Actions.

Reducing Test Cases

When reporting a bug, or investing a bug report, in Wasmtime it is easier for everyone involved when there is a test case that reproduces the bug. It is even better when that test case is as small as possible, so that developers don't need to wade through megabytes of unrelated Wasm that isn't necessary to showcase the bug. The process of taking a large test case and stripping out the unnecessary bits is called *test case reduction*.

The `wasm-tools shrink` tool can automatically reduce Wasm test cases when given

1. the original, unreduced test case, and
2. a predicate script to determine whether the bug reproduces on a given reduced test case candidate.

If the test case causes Wasmtime to segfault, the script can run Wasmtime and check its exit code. If the test case produces a different result in Wasmtime vs another Wasm engine, the script can run both engines and compare their results. It is also often useful to `grep` through the candidate's WAT disassembly to make sure that relevant features and instructions are present.

Case Study: Issue #7779

A bug was reported involving the `memory.init` instruction. The attached test case was larger than it needed to be and contained a bunch of functions and other things that were irrelevant. A perfect use case for `wasm-tools shrink`!

First, we needed a predicate script to identify the reported buggy behavior. The script is given the candidate test case as its first argument and must exit zero if the candidate exhibits the bug and non-zero otherwise.

```
#!/usr/bin/env bash

# Propagate failure: exit non-zero if any individual command exits non-zero.
set -e

# Disassembly the candidate into WAT. Make sure the `memory.init` instruction
# is present, since the bug report is about that instruction. Additionally,
make
# sure it is referencing the same data segment.
wasm-tools print $1 | grep -q 'memory.init 2'

# Make sure that the data segment in question remains unchanged, as mutating
its
# length can change the semantics of the `memory.init` instruction.
wasm-tools print $1 | grep -Eq '\(data \(;2;\) \(i32\.const 32\)
"\\01\\02\\03\\04\\05\\06\\07\\08\\ff"\)'

# Make sure that the `_start` function that contains the `memory.init` is still
# exported, so that running the Wasm will run the `memory.init` instruction.
wasm-tools print $1 | grep -Eq '\(export "_start" \(func 0\)\)'

# Run the testcase in Wasmtime and make sure that it traps the same way as the
# original test case.
cargo run --manifest-path ~/wasmtime/Cargo.toml -- run $1 2>&1 \
    | grep -q 'wasm trap: out of bounds memory access'
```

Note that this script is a little fuzzy! It just checks for `memory.init` and a particular trap. That trap can correctly occur according to Wasm semantics when `memory.init` is given certain inputs! This means we need to double check that the reduced test case actually exhibits a real bug and its inputs haven't been transformed into something that Wasm semantics specify should indeed trap. Sometimes writing very precise predicate scripts is difficult, but we do the best we can and usually it works out fine.

With the predicate script in hand, we can automatically reduce the original test case:

```
$ wasm-tools shrink predicate.sh test-case.wasm
369 bytes (1.07% smaller)
359 bytes (3.75% smaller)
357 bytes (4.29% smaller)
354 bytes (5.09% smaller)
344 bytes (7.77% smaller)
...
118 bytes (68.36% smaller)
106 bytes (71.58% smaller)
94 bytes (74.80% smaller)
91 bytes (75.60% smaller)
90 bytes (75.87% smaller)

test-case.shrunken.wasm :: 90 bytes (75.87% smaller)
================================================================================
=
(module
  (type (;0;) (func))
  (func (;0;) (type 0)
    (local i32 f32 i64 f64)
    i32.const 0
    i32.const 9
    i32.const 0
    memory.init 2
  )
  (memory (;0;) 1 5)
  (export "_start" (func 0))
  (data (;0;) (i32.const 8) "")
  (data (;1;) (i32.const 16) "")
  (data (;2;) (i32.const 32) "\01\02\03\04\05\06\07\08\ff")
)
================================================================================
=
```

In this case, the arguments to the original `memory.init` instruction haven't changed, and neither has the relevant data segment, so the reduced test case should exhibit the same behavior as the original.

In the end, it was determined that Wasmtime was behaving as expected, but the presence of the reduced test case makes it much easier to make that determination.

Cross Compiling

When contributing to Wasmtime and Cranelift you may run into issues that only reproduce on a different architecture from your development machine. Luckily, `cargo` makes cross compilation and running tests under QEMU pretty easy.

This guide will assume you are on an x86-64 with Ubuntu/Debian as your OS. The basic approach (with commands, paths, and package names appropriately tweaked) applies to other Linux distributions as well.

On Windows you can install build tools for AArch64 Windows, but targeting platforms like Linux or macOS is not easy. While toolchains exist for targeting non-Windows platforms you'll have to hunt yourself to find the right one.

On macOS you can install, through Xcode, toolchains for iOS but the main `x86_64-apple-darwin` is really the only easy target to install. You'll need to hunt for toolchains if you want to compile for Linux or Windows.

Install Rust Targets

First, use `rustup` to install Rust targets for the other architectures that Wasmtime and Cranelift support:

```
$ rustup target add \
    s390x-unknown-linux-gnu \
    riscv64gc-unknown-linux-gnu \
    aarch64-unknown-linux-gnu
```

Install GCC Cross-Compilation Toolchains

Next, you'll need to install a `gcc` for each cross-compilation target to serve as a linker for `rustc`.

```
$ sudo apt install \
    gcc-s390x-linux-gnu \
    gcc-riscv64-linux-gnu \
    gcc-aarch64-linux-gnu
```

Install `qemu`

You will also need to install `qemu` to emulate the cross-compilation targets.

```
$ sudo apt install qemu-user
```

Configure Cargo

The final bit to get out of the way is to configure `cargo` to use the appropriate `gcc` and `qemu` when cross-compiling and running tests for other architectures.

Add this to `.cargo/config.toml` in the Wasmtime repository (or create that file if none already exists).

```
[target.aarch64-unknown-linux-gnu]
linker = "aarch64-linux-gnu-gcc"
runner = "qemu-aarch64 -L /usr/aarch64-linux-gnu -E
LD_LIBRARY_PATH=/usr/aarch64-linux-gnu/lib -E WASMTIME_TEST_NO_HOG_MEMORY=1"

[target.riscv64gc-unknown-linux-gnu]
linker = "riscv64-linux-gnu-gcc"
runner = "qemu-riscv64 -L /usr/riscv64-linux-gnu -E
LD_LIBRARY_PATH=/usr/riscv64-linux-gnu/lib -E WASMTIME_TEST_NO_HOG_MEMORY=1"

[target.s390x-unknown-linux-gnu]
linker = "s390x-linux-gnu-gcc"
runner = "qemu-s390x -L /usr/s390x-linux-gnu -E LD_LIBRARY_PATH=/usr/s390x-
linux-gnu/lib -E WASMTIME_TEST_NO_HOG_MEMORY=1"
```

Cross-Compile Tests and Run Them!

Now you can use `cargo build`, `cargo run`, and `cargo test` as you normally would for any crate inside the Wasmtime repository, just add the appropriate `--target` flag!

A few examples:

- Build the `wasmtime` binary for `aarch64`:

  ```
  $ cargo build --target aarch64-unknown-linux-gnu
  ```

- Run the tests under `riscv` emulation:

  ```
  $ cargo test --target riscv64gc-unknown-linux-gnu
  ```

- Run the `wasmtime` binary under `s390x` emulation:

```
$ cargo run --target s390x-unknown-linux-gnu -- compile example.wasm
```

Coding guidelines

For the most part, Wasmtime and Cranelift follow common Rust conventions and pull request (PR) workflows, though we do have a few additional things to be aware of.

`rustfmt`

All PRs must be formatted according to rustfmt, and this is checked in the continuous integration tests. You can format code locally with:

```
$ cargo fmt
```

at the root of the repository. You can find more information about rustfmt online too, such as how to configure your editor.

Compiler Warnings and Lints

Wasmtime promotes all compiler warnings to errors in CI, meaning that the `main` branch will never have compiler warnings for the version of Rust that's being tested on CI. Compiler warnings change over time, however, so it's not always guaranteed that Wasmtime will build with zero warnings given an arbitrary version of Rust. If you encounter compiler warnings on your version of Rust please feel free to send a PR fixing them.

During local development, however, compiler warnings are simply warnings and the build and tests can still succeed despite the presence of warnings. This can be useful because warnings are often quite prevalent in the middle of a refactoring, for example. By the time you make a PR, though, we'll require that all warnings are resolved or otherwise CI will fail and the PR cannot land.

Compiler lints are controlled through the `[workspace.lints.rust]` table in the `Cargo.toml` at the root of the Wasmtime repository. A few allow-by-default lints are enabled such as `trivial_numeric_casts`, and you're welcome to enable more lints as applicable. Lints can additionally be enabled on a per-crate basis such as placing this in a `src/lib.rs` file:

```
#![warn(trivial_numeric_casts)]
```

Using `warn` here will allow local development to continue while still causing CI to promote this warning to an error.

Clippy

All PRs are gated on `cargo clippy` passing for all workspace crates and targets. All clippy lints, however, are allow-by-default and thus disabled. The Wasmtime project selectively enables Clippy lints on an opt-in basis. Lints can be controlled for the entire workspace via `[workspace.lints.clippy]`:

```
[workspace.lints.clippy]
# ...
manual_strip = 'warn'
```

or on a per-crate or module basis by using attributes:

```
#![warn(clippy::manual_strip)]
```

In Wasmtime we've found that the default set of Clippy lints is too noisy to productively use other Clippy lints, hence the allow-by-default behavior. Despite this though there are numerous useful Clippy lints which are desired for all crates or in some cases for a single crate or module. Wasmtime encourages contributors to enable Clippy lints they find useful through workspace or per-crate configuration.

Like compiler warnings in the above section all Clippy warnings are turned into errors in CI. This means that `cargo clippy` should always produce no warnings on Wasmtime's `main` branch if you're using the same compiler version that CI does (typically current stable Rust). This means, however, that if you enable a new Clippy lint for the workspace you'll be required to fix the lint for all crates in the workspace to land the PR in CI.

Clippy can be run locally with:

```
$ cargo clippy --workspace --all-targets
```

Contributors are welcome to enable new lints and send PRs for this. Feel free to reach out if you're not sure about a lint as well.

Minimum Supported `rustc` Version

Wasmtime and Cranelift support the latest three stable releases of Rust. This means that if the latest version of Rust is 1.72.0 then Wasmtime supports Rust 1.70.0, 1.71.0, and 1.72.0. CI will test by default with 1.72.0 and there will be one job running the full test suite on Linux x86_64 on 1.70.0.

Some of the CI jobs depend on nightly Rust, for example to run rustdoc with nightly features, however these use pinned versions in CI that are updated periodically and the general repository does not depend on nightly features.

Updating Wasmtime's MSRV is done by editing the `rust-version` field in the workspace root's `Cargo.toml`

Dependencies of Wasmtime

Wasmtime and Cranelift have a higher threshold than default for adding dependencies to the project. All dependencies are required to be "vetted" through the `cargo vet` tool. This is checked on CI and will run on all modifications to `Cargo.lock`.

A "vet" for Wasmtime is not a meticulous code review of a dependency for correctness but rather it is a statement that the crate does not contain malicious code and is safe for us to run during development and (optionally) users to run when they run Wasmtime themselves. Wasmtime's vet entries are used by other organizations which means that this isn't simply for our own personal use. Wasmtime additionally uses vet entries from other organizations as well which means we don't have to vet everything ourselves.

New vet entries are required to be made by trusted contributors to Wasmtime. This is all configured in the `supply-chain` folder of Wasmtime. These files generally aren't hand-edited though and are instead managed through the `cargo vet` tool itself. Note that our `supply-chain/audits.toml` additionally contains entries which indicates that authors are trusted as opposed to vets of individual crates. This lowers the burden of updating version of a crate from a trusted author.

When put together this means that contributions to Wasmtime and Cranelift which update existing dependencies or add new dependencies will not be mergeable by default (CI will fail). This is expected from our project's configuration and this situation will be handled one of a few ways:

Note that this process is not in place to prevent new dependencies or prevent updates, but rather it ensures that development of Wasmtime is done with a trusted set of code that has been reviewed by trusted parties. We welcome dependency updates and new functionality, so please don't be too alarmed when contributing and seeing a failure of `cargo vet` on CI!

`cargo vet` for Contributors

If you're a contributor to Wasmtime and you've landed on this documentation, hello and thanks for your contribution! Here's some guidelines for changing the set of dependencies in Wasmtime:

- If a new dependency is being added it might be worth trying to slim down what's required or avoiding the dependency altogether. Avoiding new dependencies is best when reasonable, but it is not always reasonable to do so. This is left to the judgement of the author and reviewer.

- When updating dependencies this should be done for a specific purpose relevant to the PR-at-hand. For example if the PR implements a new feature then the dependency update should be required for the new feature. Otherwise it's best to leave dependency updates to their own PRs. It's ok to update dependencies "just for the update" but we prefer to have that as separate PRs.

Dependency additions or updates require action on behalf of project maintainers so we ask that you don't run `cargo vet` yourself or update the `supply-chain` folder yourself. Instead a maintainer will review your PR and perform the `cargo vet` entries themselves. Reviewers will typically make a separate pull request to add `cargo vet` entries and once that lands yours will be added to the queue.

`cargo vet` for Maintainers

Maintainers of Wasmtime are required to explicitly vet and approve all dependency updates and modifications to Wasmtime. This means that when reviewing a PR you should ensure that contributors are not modifying the `supply-chain` directory themselves outside of commits authored by other maintainers. Otherwise though to add vet entries this is done through one of a few methods:

- For a PR where maintainers themselves are modifying dependencies the `cargo vet` entries can be included inline with the PR itself by the author. The reviewer knows that the author of the PR is themself a maintainer.

- PRs that "just update dependencies" are ok to have at any time. You can do this in preparation for a future feature or for a future contributor. This more-or-less is the same as the previous categories.

- For contributors who should not add vet entries themselves maintainers should review the PR and add vet entries either in a separate PR or as part of the contributor's PR itself. As a separate PR you'll check out the branch, run `cargo vet`, then rebase away the contributor's commits and push your `cargo vet` commit alone to merge. For pushing directly to the contributor's own PR be sure to read the notes below.

Note for the last case it's important to ensure that if you push directly to a contributor's PR any future updates pushed by the contributor either contain or don't overwrite your vet entries. Also verify that if the PR branch is rebased or force-pushed, the details of your previously pushed vetting remain the same: e.g., versions were not bumped and descriptive reasons remain the same. If pushing a vetting commit to a contributor's PR and also asking for more changes, request that the contributor make the requested fixes in an additional commit rather than force-pushing a rewritten history, so your existing vetting commit remains untouched. These guidelines make it easier to verify no tampering has occurred.

Policy for adding `cargo vet` entries

For maintainers this is intended to document the project's policy on adding `cargo vet` entries. The goal of this policy is to not make dependency updates so onerous that they never happen while still achieving much of the intended benefit of `cargo vet` in protection against supply-chain style attacks.

- For dependencies **that receive at least 10,000 downloads a day** on crates.io it's ok to add an entry to `exemptions` in `supply-chain/config.toml`. This does not require careful review or review at all of these dependencies. The assumption here is that a supply chain attack against a popular crate is statistically likely to be discovered relatively quickly. Changes to `main` in Wasmtime take at least 2 weeks to be released due to our release process, so the assumption is that popular crates that are victim of a supply chain attack would be discovered during this time. This policy additionally greatly helps when updating dependencies on popular crates that are common to see without increasing the burden too much on maintainers.
- For other dependencies a manual vet is required. The `cargo vet` tool will assist in adding a vet by pointing you towards the source code, as published on crates.io, to be browsed online. Manual review should be done to ensure that "nothing nefarious" is happening. For example `unsafe` should be inspected as well as use of ambient system capabilities such as `std::fs`, `std::net`, or `std::process`, and build scripts. Note that you're not reviewing for correctness, instead only for whether a supply-chain attack appears to be present.

This policy intends to strike a rough balance between usability and security. It's always recommended to add vet entries where possible, but the first bullet above can be used to update an `exemptions` entry or add a new entry. Note that when the "popular threshold" is used **do not add a vet entry** because the crate is, in fact, not vetted. This is required to go through an `[[exemptions]]` entry.

Development Process

We use issues for asking questions (open one here!) and tracking bugs and unimplemented features, and pull requests (PRs) for tracking and reviewing code submissions.

Before submitting a PR

Consider opening an issue to talk about it. PRs without corresponding issues are appropriate for fairly narrow technical matters, not for fixes to user-facing bugs or for feature implementations, especially when those features might have multiple implementation strategies that usefully could be discussed.

Our issue templates might help you through the process.

When submitting PRs

- Please answer the questions in the pull request template. They are the minimum information we need to know in order to understand your changes.
- Write clear commit messages that start with a one-line summary of the change (and if it's difficult to summarize in one line, consider splitting the change into multiple PRs), optionally followed by additional context. Good things to mention include which areas of the code are affected, which features are affected, and anything that reviewers might want to pay special attention to.
- If there is code which needs explanation, prefer to put the explanation in a comment in the code, or in documentation, rather than in the commit message. Commit messages should explain why the new version is better than the old.
- Please include new test cases that cover your changes, if you can. If you're not sure how to do that, we'll help you during our review process.
- For pull requests that fix existing issues, use issue keywords. Note that not all pull requests need to have accompanying issues.
- When updating your pull request, please make sure to re-request review if the request has been cancelled.

Focused commits or squashing

We are not picky about how your git commits are structured. When we merge your PR, we will squash all of your commits into one, so it's okay if you add fixes in new commits.

We appreciate it if you can organize your work into separate commits which each make one focused change, because then we can more easily understand your changes during review. But we don't require this.

Once someone has reviewed your PR, it's easier for us if you *don't* rebase it when making further changes. Instead, at that point we prefer that you make new commits on top of the already-reviewed work.

That said, sometimes we may need to ask you to rebase for various technical reasons. If you need help doing that, please ask!

Review and merge

Anyone may submit a pull request, and anyone may comment on or review others' pull requests. However, one review from somebody in the Core Team is required before the Core Team merges it.

Even Core Team members must create PRs and get review from another Core Team member for every change, including minor work items such as version bumps, removing warnings, etc.

Implementing WebAssembly Proposals

Adding New Support for a Wasm Proposal

The following checkboxes enumerate the steps required to add support for a new WebAssembly proposal to Wasmtime. They can be completed over the course of multiple pull requests.

- [ ] Implement support for the proposal in the `wasm-tools` repository. (example)
    - [ ] `wast` - text parsing
    - [ ] `wasmparser` - binary decoding and validation
    - [ ] `wasmprinter` - binary-to-text
    - [ ] `wasm-encoder` - binary encoding
    - [ ] `wasm-smith` - fuzz test case generation
- [ ] Update Wasmtime to use these `wasm-tools` crates, but leave the new proposal unimplemented for now (implementation comes in subsequent PRs). (example)
- [ ] Add `Config::wasm_your_proposal` to the `wasmtime` crate.
- [ ] Implement the proposal in `wasmtime`, gated behind this flag.
- [ ] Add `-Wyour-proposal` to the `wasmtime-cli-flags` crate.
- [ ] Update `tests/wast.rs` to spec tests should pass for this proposal.
- [ ] Write custom tests in `tests/misc_testsuite/*.wast` for this proposal.
- [ ] Enable the proposal in the fuzz targets.
    - [ ] Write a custom fuzz target, oracle, and/or test case generator for fuzzing this proposal in particular.

      > For example, we wrote a custom generator, oracle, and fuzz target for exercising `table.{get,set}` instructions and their interaction with GC while implementing the reference types proposal.
- [ ] Expose the proposal's new functionality in the `wasmtime` crate's API.

  > For example, the bulk memory operations proposal introduced a `table.copy` instruction, and we exposed its functionality as the `wasmtime::Table::copy`

method.

- [ ] Expose the proposal's new functionality in the C API.

  This may require extensions to the standard C API, and if so, should be defined in `wasmtime.h` and prefixed with `wasmtime_`.

- [ ] Update `docs/stability-tiers.md` with an implementation status of the proposal.

For information about the status of implementation of various proposals in Wasmtime see the associated documentation.

Adding component functionality to WASI

The cap-std repository contains crates which implement the capability-based version of the Rust standard library and extensions to that functionality. Once the functionality has been added to the relevant crates of that repository, they can be added into wasmtime by including them in the preview2 directory of the wasi crate.

Currently, WebAssembly modules which rely on preview2 ABI cannot be directly executed by the wasmtime command. The following steps allow for testing such changes.

1. Build wasmtime with the changes `cargo build --release`
2. Create a simple Webassembly module to test the new component functionality by compiling your test code to the `wasm32-wasip1` build target.
3. Build the wasi-preview1-component-adapter as a command adapter. `cargo build -p wasi-preview1-component-adapter --target wasm32-wasip1 --release --features command --no-default-features`
4. Use wasm-tools to convert the test module to a component. `wasm-tools component new --adapt wasi_snapshot_preview1=wasi_snapshot_preview1.command.wasm -o component.wasm path/to/test/module`
5. Run the test component created in the previous step with the locally built wasmtime. `wasmtime -W component-model=y -S preview2=y component.wasm`

Maintainer Guidelines

This section describes procedures and expectations for Core Team members. It may be of interest if you just want to understand how we work, or if you are joining the Core Team yourself.

Code Review

We only merge changes submitted as GitHub Pull Requests, and only after they've been approved by at least one Core Team reviewer who did not author the PR. This section covers expectations for the people performing those reviews. These guidelines are in addition to expectations which apply to everyone in the community, such as following the Code of Conduct.

It is our goal to respond to every contribution in a timely fashion. Although we make no guarantees, we aim to usually provide some kind of response within about one business day.

That's important because we appreciate all the contributions we receive, made by a diverse collection of people from all over the world. One way to show our appreciation, and our respect for the effort that goes into contributing to this project, is by not keeping contributors waiting. It's no fun to submit a pull request and then sit around wondering if anyone is ever going to look at it.

That does not mean we will review every PR promptly, let alone merge them. Some contributions have needed weeks of discussion and changes before they were ready to merge. For some other contributions, we've had to conclude that we could not merge them, no matter how much we appreciate the effort that went into them.

What this does mean is that we will communicate with each contributor to set expectations around the review process. Some examples of good communication are:

- "I intend to review this but I can't yet. Please leave me a message if I haven't responded by (a specific date in the near future)."
- "I think (a specific other contributor) should review this."
- "I'm having difficulty reviewing this PR because of (a specific reason, if it's something the contributor might reasonably be able to help with). Are you able to change that? If not, I'll ask my colleagues for help (or some other concrete resolution)."

If you are able to quickly review the PR, of course, you can just do that.

You can find open Wasmtime pull requests for which your review has been requested with this search:

https://github.com/bytecodealliance/wasmtime/pulls?q=is:open+type:pr+user-review-requested:@me

Auto-assigned reviewers

We automatically assign a reviewer to every newly opened pull request. We do this to avoid the problem of diffusion of responsibility, where everyone thinks somebody else will respond to the PR, so nobody does.

To be in the pool of auto-assigned reviewers, a Core Team member must commit to following the above goals and guidelines around communicating in a timely fashion.

We don't ask everyone to make this commitment. In particular, we don't believe it's fair to expect quick responses from unpaid contributors, although we gratefully accept any review work they do have time to contribute.

If you are in the auto-assignment pool, remember: **You are not necessarily expected to review the pull requests which are assigned to you.** Your only responsibility is to ensure that contributors know what to expect from us, and to arrange that *somebody* reviews each PR.

We have several different teams that reviewers may be auto-assigned from. You should be in teams where you are likely to know who to re-assign a PR to, if you can't review it yourself. The teams are determined by the `CODEOWNERS` file at the root of the Wasmtime repository. But despite the name, membership in these teams is *not* about who is an authority or "owner" in a particular area. So rather than creating a team for each fine-grained division in the repository such as individual target architectures or WASI extensions, we use a few coarse-grained teams:

- wasmtime-compiler-reviewers: Cranelift and Winch
- wasmtime-core-reviewers: Wasmtime, including WASI
- wasmtime-fuzz-reviewers: Fuzz testing targets
- wasmtime-default-reviewers: Anything else, including CI and documentation

Ideally, auto-assigned reviewers should be attending the regular Wasmtime or Cranelift meetings, as appropriate for the areas they're reviewing. This is to help these reviewers stay aware of who is working on what, to more easily hand off PRs to the most relevant reviewer for the work. However, this is only advice, not a hard requirement.

If you are not sure who to hand off a PR review to, you can look at GitHub's suggestions for reviewers, or look at `git log` for the paths that the PR affects. You can also just ask other Core Team members for advice.

General advice

This is a collection of general advice for people who are reviewing pull requests. Feel free to take any that you find works for you and ignore the rest. You can also open pull requests to suggest more references for this section.

The Gentle Art of Patch Review suggests a "Three-Phase Contribution Review" process:

1. Is the idea behind the contribution sound?
2. Is the contribution architected correctly?
3. Is the contribution polished?

Phase one should be a quick check for whether the pull request should move forward at all, or needs a significantly different approach. If it needs significant changes or is not going to be accepted, there's no point reviewing in detail until those issues are addressed.

On the other end, it's a good idea to defer reviewing for typos or bikeshedding about variable names until phase three. If there need to be significant structural changes, entire paragraphs or functions might disappear, and then any minor errors that were in them won't matter.

The full essay has much more advice and is recommended reading.

Release Process

This is intended to serve as documentation for Wasmtime's release process. It's largely an internal checklist for those of us performing a Wasmtime release, but others might be curious in this as well!

Releasing a major version

Major versions of Wasmtime are released once-a-month. Most of this is automatic and all that needs to be done is to merge GitHub PRs that CI will generate. At a high-level the structure of Wasmtime's release process is:

- On the 5th of every month a new `release-X.Y.Z` branch is created with the current contents of `main`.
- On the 20th of every month this release branch is published to crates.io and release artifacts are built.

This means that Wasmtime releases are always at least two weeks behind development on `main` and additionally happen once a month. The lag time behind `main` is intended to give time to fuzz changes on `main` as well as allow testing for any users using `main`. It's expected, though, that most consumers will likely use the release branches of wasmtime.

A detailed list of all the steps in the release automation process are below. The steps requiring interactions are **bolded**, otherwise everything else is automatic and this is documenting what automation does.

1. On the 5th of every month, (configured via `.github/workflows/release-process.yml`) a CI job will run and do these steps:
    - Download the current `main` branch
    - Push the `main` branch to `release-X.Y.Z`
    - Run `./scripts/publish.rs` with the `bump` argument
    - Commit the changes
    - Push these changes to a temporary `ci/*` branch
    - Open a PR with this branch against `main`
    - This step can also be triggered manually with the `main` branch and the `cut` argument.
2. **A maintainer of Wasmtime merges this PR**
    - It's intended that this PR can be immediately merged as the release branch has been created and all it's doing is bumping the version.
3. **Time passes and the `release-X.Y.Z` branch is maintained**
    - All changes land on `main` first, then are backported to `release-X.Y.Z` as necessary.

4. On the 20th of every month (same CI job as before) another CI job will run performing:
    - Reset to `release-X.Y.Z`
    - Update the release date of `X.Y.Z` to today in `RELEASES.md`
    - Add a special marker to the commit message to indicate a tag should be made.
    - Open a PR against `release-X.Y.Z` for this change
    - This step can also be triggered manually with the `main` branch and the `release-latest` argument.
5. **A maintainer of Wasmtime merges this PR**
    - When merged, will trigger the next steps due to the marker in the commit message. A maintainer should double-check there are no open security issues, but otherwise it's expected that all other release issues are resolved by this point.
6. The main CI workflow at `.github/workflow/main.yml` has special logic at the end such that pushes to the `release-*` branch will scan the git logs of pushed changes for the special marker added by `release-process.yml`. If found and CI passes a tag is created and pushed.
7. Once a tag is created the `.github/workflows/publish-*` workflows run. One publishes all crates as-is to crates.io and the other will download all artifacts from the `main.yml` workflow and then upload them all as an official release.

If all goes well you won't have to read up much on this and after hitting the Big Green Button for the automatically created PRs everything will merrily carry on its way.

Releasing a patch version

Making a patch release is somewhat more manual than a major version, but like before there's automation to help guide the process as well and take care of more mundane bits.

This is a list of steps taken to perform a patch release for 2.0.1 for example. Like above human interaction is indicated with **bold** text in these steps.

1. **Necessary changes are backported to the `release-2.0.0` branch from `main`**
    - All changes must land on `main` first (if applicable) and then get backported to an older branch. Release branches should already exist from the above major release steps.
    - CI may not have been run in some time for release branches so it may be necessary to backport CI fixes and updates from `main` as well.
    - When merging backports maintainers need to double-check that the `PUBLIC_CRATES` listed in `scripts/publish.rs` do not have semver-API-breaking changes (in the strictest sense). All security fixes must be done in such a way that the API doesn't break between the patch version and the original version.
    - Don't forget to write patch notes in `RELEASES.md` for backported changes.
2. **The patch release process is triggered manually with the `release-2.0.0` branch and the `release-patch` argument**

    - This will run the `release-process.yml` workflow. The `scripts/publish.rs` script will be run with the `bump-patch` argument.
    - The changes will be committed with a special marker indicating a release needs to be made.
    - A PR will be created from a temporary `ci/*` branch to the `release-2.0.0` branch which, when merged, will trigger the release process.
3. **Review the generated PR and merge it**
    - This will resume from step 6 above in the major release process where the special marker in the commit message generated by CI will trigger a tag to get pushed which will further trigger the rest of the release process.
    - Please make sure to update the `RELEASES.md` at this point to include the `Released on` date by pushing directly to the branch associated with the PR.

Releasing a security patch

For security releases see the documentation of the vulnerability runbook.

Releasing Notes

Release notes for Wasmtime are written in the `RELEASES.md` file in the root of the repository. Management of this file looks like:

- (theoretically) All changes on `main` which need to write an entry in `RELEASES.md`.
- When the `main` branch gets a version the `RELEASES.md` file is emptied and replaced with `ci/RELEASES-template.md`. An entry for the upcoming release is added to the bulleted list at the bottom.
- (realistically) After a `release-X.Y.Z` branch is created release notes are updated and edited on the release branch.

This means that `RELEASES.md` only has release notes for the release branch that it is on. Historical release notes can be found through links at the bottom to previous copies of `RELEASES.md`

Governance

... more coming soon

Contributor Covenant Code of Conduct

Note: this Code of Conduct pertains to individuals' behavior. Please also see the Organizational Code of Conduct.

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of

unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the Bytecode Alliance CoC team at report@bytecodealliance.org. The CoC team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The CoC team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the Bytecode Alliance's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4