NIFs in Elixir

Native Implemented Functions in the Erlang VM

Ahmad Sattar Atta

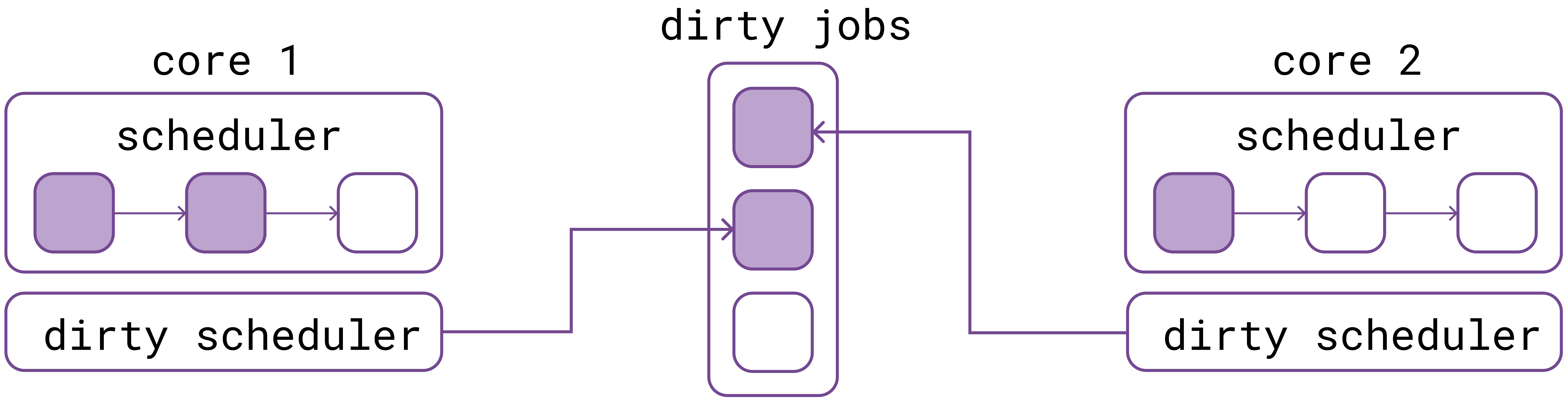
Content

Use cases in the wild

- Dataframe manipulation (Explorer)
- Serialization/deserialization (Jsonrs)
- Code execution runtimes (wasmex, extism)
- Hashing/cryptography/compression
- Specialized data structures (Discord.SortedSet)
- CPU-heavy tasks
- Low-level I/O (Elixir Circuits)
- Bindings to other libraries (OpenCV, libsodium)

Overview

- NIF (Native Implemented Functions)
- Dynamically loadable shared library
- Erlang Term
- Reductions (max ~4000 reductions or 1ms for 19.2+)
- Scheduler



Scheduler implications

- Example with the *--erl '+S 1:1'* config
- 3 second timeout on single scheduler

```
$ time ./scheduler.exs --
..
0.34s user 0.08s
system 12% cpu 3.347 total
```

```
$ time ./scheduler.exs -- d
..
0.33s user 0.09s
system 126% cpu 0.325 total
```

```rust
#[rustler::nif]
fn sleep(duration_ms: u64) -> () {
    thread::sleep(..);
}
```

```rust
#[rustler::nif(schedule = "DirtyCpu")]
fn sleep_dirty(duration_ms: u64) -> () {
    thread::sleep(..);
}
```

NIF C library

- Make C functions available for calling within the BEAM
- Allocate memory blobs
- Read and write Erlang terms
- Interact with BEAM processes

```c
static ERL_NIF_TERM sensor_reading(
  ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]) {
  // .. sensor reading code
  return enif_make_tuple(env, 2, enif_make_atom(env, "ok"), reading);
}
static ErlNifFunc nif_funcs[] = {
  // name, arity, fn pointer, flags
  {"sensor_reading", 0, do_sensor_reading, 0}
};
ERL_NIF_INIT(Elixir.Some.Module, nif_funcs, NULL,NULL,NULL,NULL)
```

Rust & Zig examples

- Ease of use
- API abstractions

```rust
#[rustler::nif]
fn alloc_vec() → Vec<u16> {
    vec![0, 1, 2, 3]
}
```

```zig
const std = @import("std");
const ArrayList = std.ArrayList;
const beam = @import("beam");

pub fn alloc_vec(env: beam.env) !beam.term {
  const slice = try beam.allocator.alloc(u16, 4);
  defer beam.allocator.free(slice);

  for (slice) |*item, index| {
      item.* = @intCast(u16, index);
  }

  return beam.make(env, slice, .{});
}
```

Actual Rustler usage

```rust
#[rustler::nif]
fn apply_patch(json_string: ResourceArc<JsonString>,
               json_patch_string: &str
) -> Result<(), Error> {
    let entity_patch: EntityUpdate = serde_json::from_str(json_patch_string)
        .map_err(|_e| {
            Error::Term(Box::new(atoms::json_deserialize_error()))
        })?;
    let mut json_string = json_string.as_mut();

    patch(&mut json_string, &entity_patch.patch)
        .map(|_| ())
        .map_err(|_| Error::Term(Box::new(atoms::malformed_patch())))
}
```

Demo

Thank you!