FUTURE EXTENSIONS TO THE NATIVE INTERFACE

Rickard Green

rickard@erlang.org

Erlang Factory San Francisco Bay Area 2011

EXISTING NATIVE INTERFACE

- Drivers
  - Provides native code via a set of callbacks
  - Executed in context of a port
  - Ports similar to processes
- NIF
  - Native code
  - Executed in context of a process

COMMUNICATING WITH PORTS

Message To Port	Message From Port
`{Pid, {command, Data}}`	`{Port, {data, Data}}`
`{Pid, close}`	`{Port, closed}`
`{Pid, {connect, NewPid}}`	`{Port, connected}`

Port BIFs
`port_command(Port, Data)`
`port_close(Port)`
`port_connect(Port, NewPid)`
`port_control(Port, Operation, Data)`
`erlang:port_call(Port, Operation, Data)`
`port_info(Port) / port_info(Port, Item)`

DRIVER CALLBACKS

```
typedef struct erl_drv_entry {
    int (*init)(void);
    ErlDrvData (*start)(ErlDrvPort port, char *command);
    void (*stop)(ErlDrvData drv_data);
    void (*output)(ErlDrvData drv_data, char *buf, int len);
    void (*ready_input)(ErlDrvData drv_data, ErlDrvEvent event);
    void (*ready_output)(ErlDrvData drv_data, ErlDrvEvent event);
    char *driver_name;
    void (*finish)(void);
    void *handle;
    int (*control)(ErlDrvData drv_data, unsigned int command, char *buf, int len, char **rbuf, int rlen);
    void (*timeout)(ErlDrvData drv_data);
    void (*outputv)(ErlDrvData drv_data, ErlIOVec *ev);
    void (*ready_async)(ErlDrvData drv_data, ErlDrvThreadData thread_data);
    void (*flush)(ErlDrvData drv_data);
    int (*call)(ErlDrvData drv_data, unsigned int command, char *buf, int len, char **rbuf, int rlen, unsigned int *flags);
    void (*event)(ErlDrvData drv_data, ErlDrvEvent event, ErlDrvEventData event_data);
    int extended_marker;
    int major_version;
    int minor_version;
    int driver_flags;
    void *handle2;
    void (*process_exit)(ErlDrvData drv_data, ErlDrvMonitor *monitor);
    void (*stop_select)(ErlDrvEvent event, void* reserved);
} ErlDrvEntry;
```


ERLANG

DRIVER CALLBACKS

Driver Callback	Operation	Call Reason
output() outputv()	Port ! {Pid, {command, Data}} port_command()	Process communication
control()	port_control()	
call()	erlang:port_call()	
timeout()	driver_set_timer()	Event subscribtion
process_exit()	driver_monitor_process()	
ready_async()	driver_async()	
ready_input() ready_output() event() stop_select()	driver_select()	

ERLANG

ISSUES WITH PORTS

- Cannot handle arbitrary protocols
  - Only understand a fixed set of messages
    - `{Pid, {command, Data}}`
  - For example, cannot handle the I/O protocol
  - Need to be paired with a process
    - may imply unnecessary scheduling
- Lacks functionality compared to processes
  - Cannot be monitored
  - No remote operations
- Inefficient compared to NIFs
  - Argument processing
  - Heap management
- Own implementation
  - Doesn't benefit from optimizations nor new functionality made for processes

NATIVE IMPLEMENTED FUNCTIONS

- Easy to implement
- Efficient
  - Direct call into native code
  - Reads and writes directly from/to process heap
- Can ***not*** replace ports
  - Only code
  - No fixed execution context
  - Cannot subscribe for events

```
ERL_NIF_TERM hello_nif_world(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[])
{
    return enif_make_string(env, "Hello NIF world!", ERL_NIF_LATIN1);
}
```


ERLANG

NATIVE PROCESSES

- Indistinguishable from ordinary processes
- Minimal amount of native code
  - Execute both Erlang code and native code
- Share implementation with ordinary processes
- As efficient as NIFs
- Share implementation with NIFs

NATIVE PROCESS CREATION

- `spawn()`
- Convert to native process in NIF
  - Call `enif_become_native_process()`
  - Process will enter a native event loop and will not return from NIF
  - Code is part of a NIF library

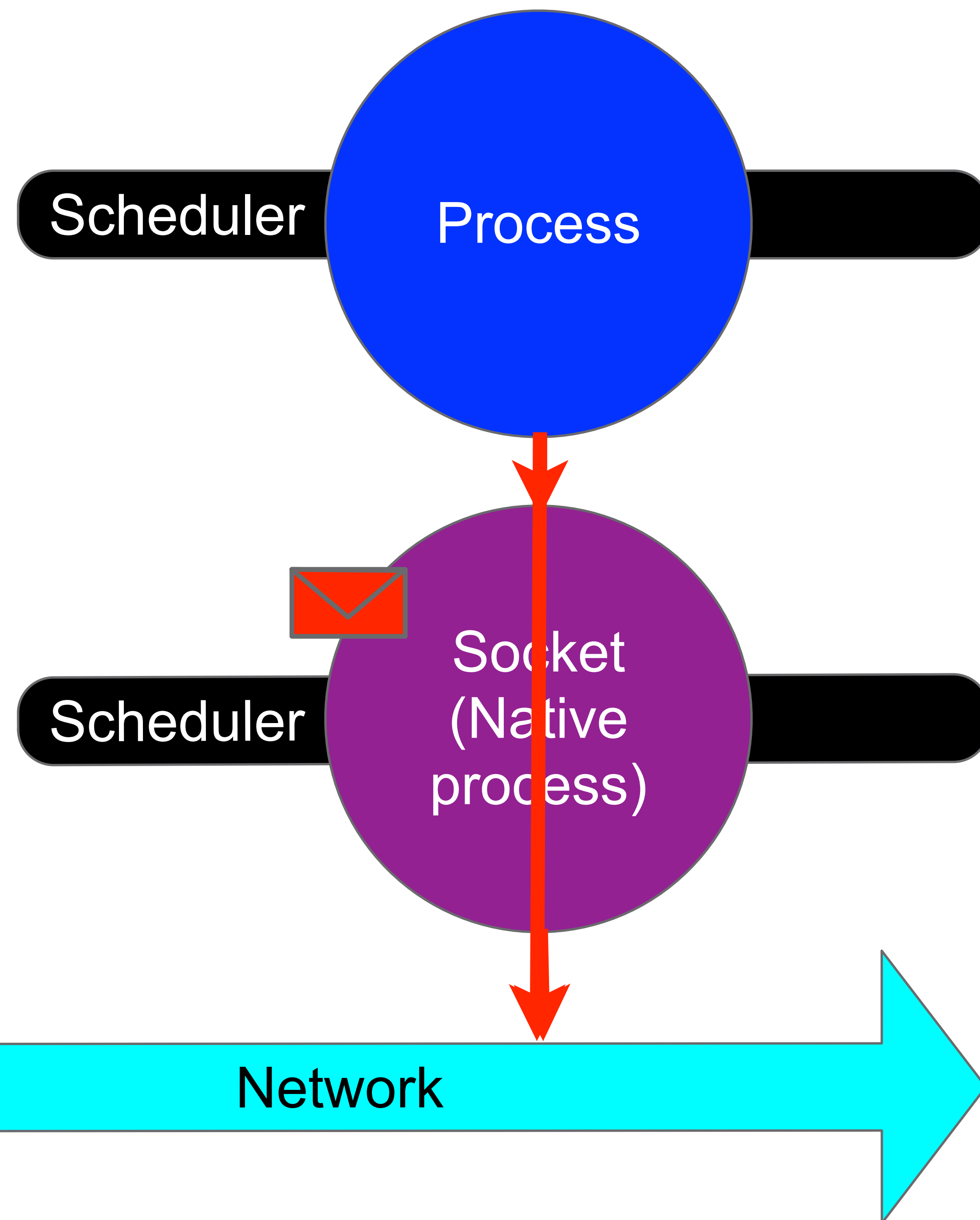
```
ERL_NIF_TERM enif_become_native_process(ErlNifEnv *env, ErlNProcSetup *setup);

typedef struct {
    ...
    void *state;
    void (*io_event)(ErlNifEnv *,
                     void *,
                     ErlNProcIoEventType,
                     ErlNProcIoDescr);
    void (*event)(ErlNifEnv *,
                  void *,
                  ErlNProcEventType,
                  ErlNifEnv *,
                  int,
                  ERL_NIF_TERM *);
} ErlNProcSetup;
```

EVENT CALLBACK

```
void event_callback(ErlNifEnv *proc_env,
                    void *state,
                    ErlNProcEventType type,
                    ErlNifEnv *arg_env,
                    int argc,
                    ERL_NIF_TERM *argv);
```

enif_make_copy()

```
typedef enum {
    ERL_NPROC_EVENT_MESSAGE = 0,
    ERL_NPROC_EVENT_TERMINATE = 1,
    ERL_NPROC_EVENT_NEW_CODE = 2,
    ...
} ErlNProcEventType;
```

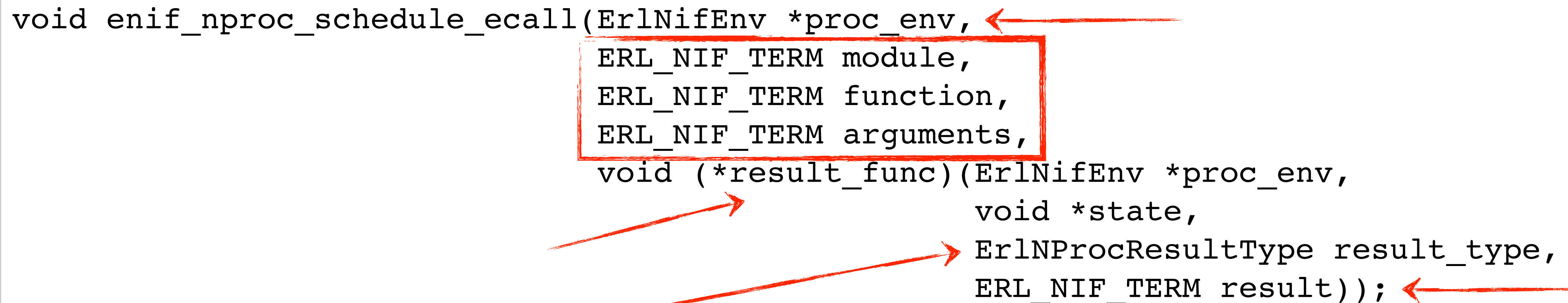


IO-EVENT CALLBACK

```
void io_event_callback(ErlNifEnv *proc_env,
                       void *state,
                       ErlNProcIoEventType type,
                       ErlNProcIoDescr io_descr);

typedef enum {
    ERL_NPROC_IO_EVENT_INPUT_READY = 0,
    ERL_NPROC_IO_EVENT_OUTPUT_READY = 1,
    ERL_NPROC_IO_EVENT_STOP_SELECT = 2
} ErlNativeProcIoEventType;
```

```
void enif_nproc_select(ErlNifEnv *env, ErlNProcIoDescr io_descr, int mode, int on);

#define ERL_NPROC_IO_EVENT_READ   1
#define ERL_NPROC_IO_EVENT_WRITE  2
#define ERL_NPROC_IO_EVENT_USE    4
```

EXECUTING ERLANG CODE



```
void enif_nproc_schedule_ecall(ErlNifEnv *proc_env,
                               ERL_NIF_TERM module,
                               ERL_NIF_TERM function,
                               ERL_NIF_TERM arguments,
                               void (*result_func)(ErlNifEnv *proc_env,
                                                   void *state,
                                                   ErlNProcResultType result_type,
                                                   ERL_NIF_TERM result));

typedef enum {
    ERL_NPROC_RESULT_RETURN = 0,
    ERL_NPROC_RESULT_THROW = 1,
    ERL_NPROC_RESULT_ERROR = 2,
    ERL_NPROC_RESULT_EXIT = 3
} ErlNProcResultType;
```

- May only be called by callbacks
- Erlang code may call NIFs
- Continuations of a callback call
- No new callback calls until done

NATIVE PROCESS CODE CHANGE

- Load new module
- Call NIF in new module using `enif_nproc_schedule_ecall()`
- Call `enif_native_process_code_change()` from NIF

```
ERL_NIF_TERM enif_native_process_code_change(ErlNifEnv *env, ErlNProcSetup *setup);

typedef struct {
    ...
    void *state;
    void (*io_event)(ErlNifEnv *,
                     void *,
                     ErlNProcIoEventType,
                     ErlNProcIoDescr);
    void (*event)(ErlNifEnv *,
                  void *,
                  ErlNProcEventType,
                  ErlNifEnv *,
                  int,
                  ERL_NIF_TERM *);
} ErlNProcSetup;
```

```
void *enif_get_nproc_state(ErlNifEnv *env);
```

REVERTING BACK TO AN ORDINARY PROCESS

```
void enif_nproc_become_ordinary_process(ErlNifEnv *env);
```

- Call `enif_nproc_become_ordinary_process()`
- May only be called in callback
- Exits native event loop and return to Erlang code from NIF that converted to native process

REASONS TO WRITE NATIVE CODE

- Optimizing
- Access functionality not available in Erlang
  - Operating System services
    - Sockets
    - Files
    - ...
  - Third party libraries
    - ZLib
    - OpenSSL
    - wxWidgets
    - ...

PROBLEMATIC NATIVE CODE

- Unstable code
  - Runtime system crash
  - Internal runtime system inconsistency
- Code that run for long periods of time
  - Destroy responsiveness of runtime system (also in the SMP runtime system)

UNSTABLE NATIVE CODE

- Fix if possible
- Runtime system executing native code is vulnerable
- Isolate by running in separate runtime system
  - Unstable third party libraries
  - Debugging
- Distribution transparency makes this easier
  - Ports aren't distribution transparent
  - Native processes are

LONG EXECUTION OF NATIVE CODE

- Make preemptable by dividing a large workload into a number of smaller workloads
  - Preferred but not always possible
    - Third party libraries
    - Blocking wait
- Threading
  - Async thread pool (drivers only)
  - Own driver/NIF threads

PREEMPTABLE NIF AND DRIVER

```
large_workload(Arg) ->
    case large_workload_nif(Arg) of
       {continue, Reductions, NewState} ->
           erlang:bump_reductions(Reductions),
           large_workload(NewState);
       {done, Reductions, Result} ->
           erlang:bump_reductions(Reductions),
           Result
    end.
```

```
static void large_workload_output_callback(ErlDrvData drv_data,
                                           char *buf, int len)
{
    int not_done;
    MyDrvData *data = (ErlDrvData *) drv_data;
    /* Do work... */
    if (not_done) driver_set_timer(data->port, 0);
}

static void timeout_callback(ErlDrvData drv_data)
{
    int not_done;
    MyDrvData *data = (ErlDrvData *) drv_data;
    /* Do more work... */
    if (not_done) driver_set_timer(data->port, 0);
}
```

PREEMPTABLE NATIVE PROCESS CALLBACK

```
void large_workload_callback(ErlNifEnv *proc_env, void *state, ErlNProcEventType type,
                             ErlNifEnv *arg_env, int argc, ERL_NIF_TERM *argv)
{
    ERL_NIF_TERM erl_state;
    /* Parse args, etc... */
    do_work(proc_env, state,
            ERL_NPROC_RESULT_RETURN, erl_state);
}

static void do_work(ErlNifEnv *env,
                    void *state,
                    ErlNProcResultType rtype,
                    ERL_NIF_TERM erl_state)
{
    int not_done, reductions;
    ERL_NIF_TERM new_erl_state;
    /* Do work... */
    if (not_done)
        enif_nproc_schedule_ecall(enif_make_atom(env, "my_module"),
                                  enif_make_atom(env, "reschedule_point"),
                                  enif_make_list(env, 2,
                                                 enif_make_int(env, reductions),
                                                 new_erl_state),
                                  do_work);
}
```

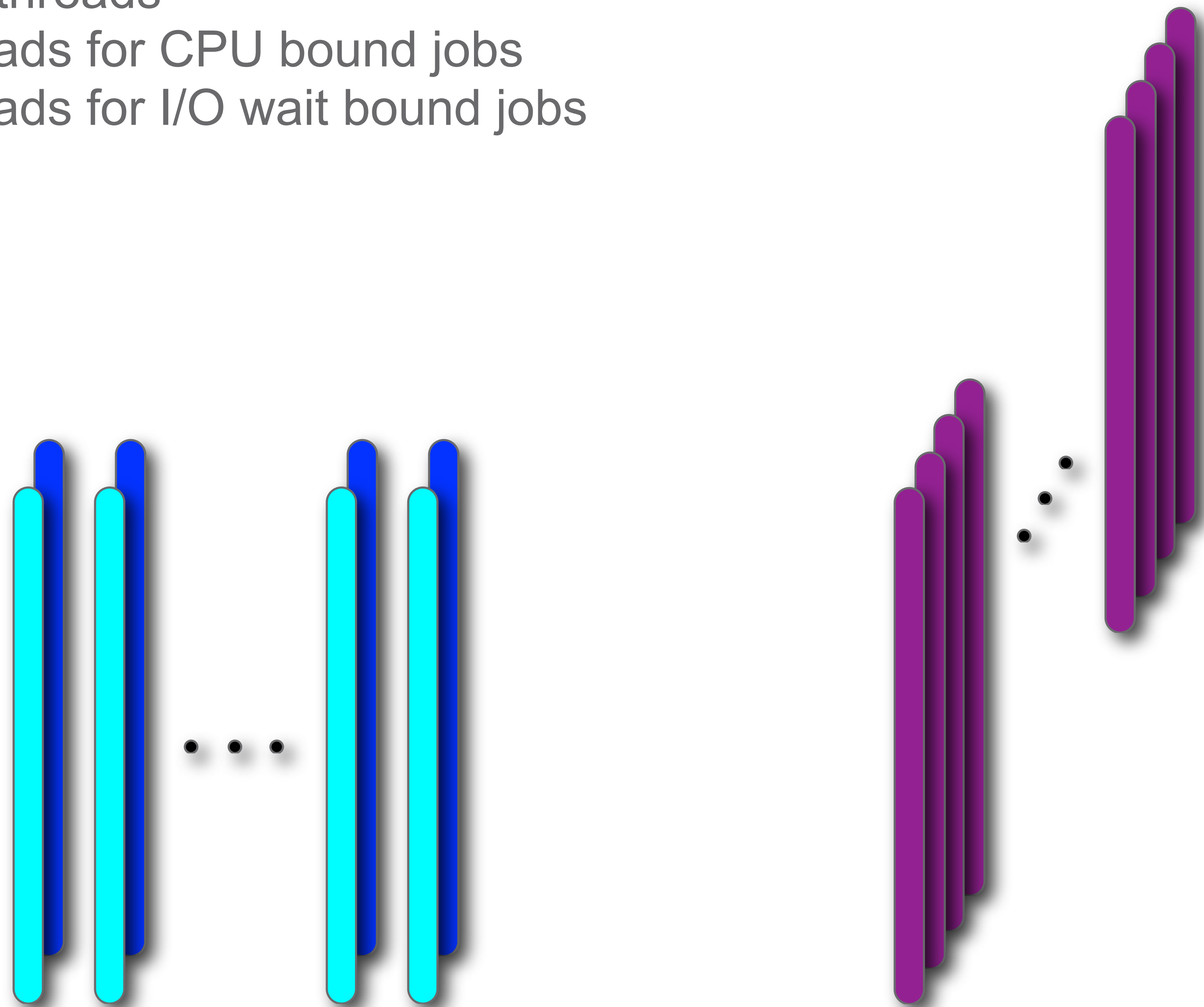
```
-module(my_module).
-export([reschedule_point/2]).

reschedule_point(Reds, State) ->
    erlang:bump_reductions(Reds),
    State.
```

EASE MANAGEMENT OF LONG JOBS

- Ease threading
- Two sets of long jobs
  - CPU bound jobs
  - I/O wait bound jobs
- When executed on other threads than schedulers threads
  - CPU bound have large impact on priority of other processes
  - I/O wait bound have a small or no impact on other processes
- Two set of thread pools

DIRTY SCHEDULERS

- Ordinary scheduler threads
- Dirty scheduler threads for CPU bound jobs
- Dirty scheduler threads for I/O wait bound jobs



DIRTY JOBS

- Thread safety
  - SMP runtime system; as on any scheduler thread
  - Non-SMP runtime system
    - Limited set of functions can be used
    - Behavior slightly different
- Scheduling
  - Borrowing reductions for CPU bound jobs
    - May be automatically calculated on some platforms
    - Native code need to bump reductions
    - Punishment
- ERTS internal functionality

USING DIRTY SCHEDULERS

```
#define ERTS_DIRTY_JOB_FLG_CPU_BOUND 1
#define ERTS_DIRTY_JOB_FLG_IO_WAIT_BOUND 2

/* Dirty NIFs and Native Process Callbacks */
int enif_is_dirty_job(ErlNifEnv *env);
int enif_is_on_dirty_scheduler(ErlNifEnv *env);
int enif_have_dirty_schedulers(void);

ERL_NIF_TERM enif_schedule_dirty_nif(ErlNifEnv *env, int flags,
                                     ERL_NIF_TERM (*dirty_nif)(ErlNifEnv *, int
                                                                ERL_NIF_TERM *));
ERL_NIF_TERM enif_schedule_dirty_nif_finalizer(ErlNifEnv *env,
                                               ERL_NIF_TERM (*dirty_nif_finalizer)(
                                                                  ErlNifEnv *,
                                                                  ERL_NIF_TERM));

void enif_nproc_schedule_dirty_event(...);
void enif_schedule_dirty_event_finalizer(...);

void enif_nproc_schedule_dirty_io_event(...);
void enif_schedule_dirty_io_event_finalizer(...);

/* Dirty Driver Callbacks */

...
```

EXAMPLE OF DIRTY SCHEDULER USE

```
static ERL_NIF_TERM perhaps_long_job(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[])
{
    if (will_be_short_job(env, argc, argv))
        return do_job(env, argc, argv);
    else
        return enif_schedule_dirty_nif(env, ENIF_DIRTY_JOB_CPU_BOUND, do_job);
}

static ERL_NIF_TERM do_job(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[])
{
    ERL_NIF_TERM result;
    /* Do work... */
    return enif_schedule_dirty_nif_finalizer(env, result, do_job_finalizer);
}

static ERL_NIF_TERM do_job_finalizer(ErlNifEnv *env, ERL_NIF_TERM result)
{
    ERL_NIF_TERM modified_result;
    /* Finalize... */
    return modified_result;
}
```


ERICSSON

ERLANG

SUMMARY

- Native processes make drivers obsolete
  - Will be able to handle any protocol
  - Possible to minimize the need for native code
  - Will have access to the same functionality as ordinary processes
  - More efficient
  - Distribution transparent
  - Share implementations
- Dirty schedulers
  - Now easy to handle any code that misbehaves by monopolizing a scheduler thread
  - Ordinary schedulers should never be harassed by misbehaving jobs
- Designed by the OTP-team; based on ideas hatched by Tony Rogvall
- When will these features show up?
  - Hopefully R15, but this is ***no*** promise

ERICSSON