Enduro/X Core - Feature #45

Nodejs support

04/01/2016 10:35 AM - Madars

Status:	New	**Start date:**	04/01/2016
Priority:	Normal (Code 4)	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			

Description

Enduro/X shall be ported to NodeJS with first class support (ATMI client & serve wtih carray, string, json and UBF buffers).

History

#1 - 04/01/2016 10:36 AM - Madars

Ideas for libuv support: https://github.com/oracle/node-oracledb
https://github.com/oracle/node-oracledb/blob/master/src/njs/src/njsConnection.cpp#L438

#2 - 04/04/2016 02:43 PM - Madars

According to https://nikhilm.github.io/uvbook/threads.html uv_async_send() can be used to notify Nodejs about c++ side event. Search in google "c++ callback node.js".

An Introduction to libuv
THREADS

« Networking :: Contents :: Processes »
Threads

Wait a minute? Why are we on threads? Aren't event loops supposed to be the way to do web-scale programming? Well... no. Threads are still the medium in which processors do their jobs. Threads are therefore mighty useful sometimes, even though you might have to wade through various synchronization primitives.

Threads are used internally to fake the asynchronous nature of all of the system calls. libuv also uses threads to allow you, the application, to perform a task asynchronously that is actually blocking, by spawning a thread and collecting the result when it is done.

Today there are two predominant thread libraries: the Windows threads implementation and POSIX's pthreads. libuv's thread API is analogous to the pthreads API and often has similar semantics.

A notable aspect of libuv's thread facilities is that it is a self contained section within libuv. Whereas other features intimately depend on the event loop and callback principles, threads are complete agnostic, they block as required, signal errors directly via return values, and, as shown in the first example, don't even require a running event loop.

libuv's thread API is also very limited since the semantics and syntax of threads are different on all platforms, with different levels of completeness.

This chapter makes the following assumption: There is only one event loop, running in one thread (the main thread). No other thread interacts with the event loop (except using uv_async_send).

Core thread operations

There isn't much here, you just start a thread using uv_thread_create() and wait for it to close using uv_thread_join().

thread-create/main.c

```
1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
int main() {
```

```
    int tracklen = 10;
    uv_thread_t hare_id;
    uv_thread_t tortoise_id;
    uv_thread_create(&hare_id, hare, &tracklen);
    uv_thread_create(&tortoise_id, tortoise, &tracklen);



uv_thread_join(&hare_id);
    uv_thread_join(&tortoise_id);
    return 0;
}
```

Tip

uv_thread_t is just an alias for pthread_t on Unix, but this is an implementation detail, avoid depending on it to always be true.

The second parameter is the function which will serve as the entry point for the thread, the last parameter is a void * argument which can be used to pass custom parameters to the thread. The function hare will now run in a separate thread, scheduled pre-emptively by the operating system:

thread-create/main.c

```
1
2
3
4
5
6
7
8
9
void hare(void *arg) {
int tracklen = *((int *) arg);
while (tracklen) {
tracklen--;
sleep(1);
fprintf(stderr, "Hare ran another step\n");
}
fprintf(stderr, "Hare done running!\n");
}
```

Unlike pthread_join() which allows the target thread to pass back a value to the calling thread using a second parameter, uv_thread_join() does not. To send values use Inter-thread communication.

Synchronization Primitives

This section is purposely spartan. This book is not about threads, so I only catalogue any surprises in the libuv APIs here. For the rest you can look at the pthreads man pages.

Mutexes

The mutex functions are a direct map to the pthread equivalents.

libuv mutex functions

```
UV_EXTERN int uv_mutex_init(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_destroy(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_lock(uv_mutex_t* handle);
UV_EXTERN int uv_mutex_trylock(uv_mutex_t* handle);
UV_EXTERN void uv_mutex_unlock(uv_mutex_t* handle);
```

The uv_mutex_init() and uv_mutex_trylock() functions will return 0 on success, and an error code otherwise.

If libuv has been compiled with debugging enabled, uv_mutex_destroy(), uv_mutex_lock() and uv_mutex_unlock() will abort() on error. Similarly uv_mutex_trylock() will abort if the error is anything other than EAGAIN or EBUSY.

Recursive mutexes are supported by some platforms, but you should not rely on them. The BSD mutex implementation will raise an error if a thread which has locked a mutex attempts to lock it again. For example, a construct like:

```
uv_mutex_lock(a_mutex);
uv_thread_create(thread_id, entry, (void *)a_mutex);
uv_mutex_lock(a_mutex);
// more things here
```

can be used to wait until another thread initializes some stuff and then unlocks a_mutex but will lead to your program crashing if in debug mode, or return an error in the second call to uv_mutex_lock().

Note

Mutexes on linux support attributes for a recursive mutex, but the API is not exposed via libuv.

Locks

Read-write locks are a more granular access mechanism. Two readers can access shared memory at the same time. A writer may not acquire the lock when it is held by a reader. A reader or writer may not acquire a lock when a writer is holding it. Read-write locks are frequently used in databases. Here is a toy example.

locks/main.c - simple rwlocks

```
1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
#include &lt;stdio.h&gt;
#include &lt;uv.h&gt;



uv_barrier_t blocker;
uv_rwlock_t numlock;
int shared_num;

void reader(void *n) {
```

```
int num = *(int *)n;
int i;
for (i = 0; i < 20; i++) {
uv_rwlock_rdlock(&numlock);
printf("Reader %d: acquired lock\n", num);
printf("Reader %d: shared num = %d\n", num, shared_num);
uv_rwlock_rdunlock(&numlock);
printf("Reader %d: released lock\n", num);
}
uv_barrier_wait(&blocker);
}

void writer(void *n) {
int num = *(int *)n;
int i;
for (i = 0; i < 20; i++) {
uv_rwlock_wrlock(&numlock);
printf("Writer %d: acquired lock\n", num);
shared_num++;
printf("Writer %d: incremented shared num = %d\n", num, shared_num);
uv_rwlock_wrunlock(&numlock);
printf("Writer %d: released lock\n", num);
}
uv_barrier_wait(&blocker);
}

int main() {
uv_barrier_init(&blocker, 4);

shared_num = 0;
    uv_rwlock_init(&numlock);



uv_thread_t threads[3];



int thread_nums[] = {1, 2, 1};
    uv_thread_create(&threads[0], reader, &thread_nums[0]);
    uv_thread_create(&threads[1], reader, &thread_nums[1]);



uv_thread_create(&threads[2], writer, &thread_nums[2]);



uv_barrier_wait(&blocker);
    uv_barrier_destroy(&blocker);



uv_rwlock_destroy(&numlock);
    return 0;
}
```

Run this and observe how the readers will sometimes overlap. In case of multiple writers, schedulers will usually give them higher priority, so if you add two writers, you'll see that both writers tend to finish first before the readers get a chance again.

We also use barriers in the above example so that the main thread can wait for all readers and writers to indicate they have ended.

Others

libuv also supports semaphores, condition variables and barriers with APIs very similar to their pthread counterparts.

In addition, libuv provides a convenience function uv_once(). Multiple threads can attempt to call uv_once() with a given guard and a function pointer, only the first one will win, the function will be called once and only once:

```
/* Initialize guard */
static uv_once_t once_only = UV_ONCE_INIT;

int i = 0;
```

```
void increment() {
i++;
}

void thread1() {
/* ... work */
uv_once(once_only, increment);
}

void thread2() {
/* ... work */
uv_once(once_only, increment);
}

int main() {
/* ... spawn threads */
}
```
After all threads are done, i == 1.

libuv v0.11.11 onwards also added a uv_key_t struct and api for thread-local storage.

libuv work queue

uv_queue_work() is a convenience function that allows an application to run a task in a separate thread, and have a callback that is triggered when the task is done. A seemingly simple function, what makes uv_queue_work() tempting is that it allows potentially any third-party libraries to be used with the event-loop paradigm. When you use event loops, it is imperative to make sure that no function which runs periodically in the loop thread blocks when performing I/O or is a serious CPU hog, because this means that the loop slows down and events are not being handled at full capacity.

However, a lot of existing code out there features blocking functions (for example a routine which performs I/O under the hood) to be used with threads if you want responsiveness (the classic 'one thread per client' server model), and getting them to play with an event loop library generally involves rolling your own system of running the task in a separate thread. libuv just provides a convenient abstraction for this.

Here is a simple example inspired by node.js is cancer. We are going to calculate fibonacci numbers, sleeping a bit along the way, but run it in a separate thread so that the blocking and CPU bound task does not prevent the event loop from performing other activities.

queue-work/main.c - lazy fibonacci

```
1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
void fib(uv_work_t *req) {
    int n = *(int *) req->data;
    if (random() % 2)
        sleep(1);
    else
        sleep(3);
    long fib = fib_(n);
    fprintf(stderr, "%dth fibonacci is %lu\n", n, fib);
}



void after_fib(uv_work_t req, int status) {
fprintf(stderr, "Done calculating %dth fibonacci\n", *(int *) req->data);
}
```
The actual task function is simple, nothing to show that it is going to be run in a separate thread. The uv_work_t structure is the clue. You can pass arbitrary data through it using the void data field and use it to communicate to and from the thread. But be sure you are using proper locks if you are changing things while both threads may be running.

The trigger is uv_queue_work:

queue-work/main.c

```
1
 2
```

```
int main() {
    loop = uv_default_loop();
```

```
int data[FIB_UNTIL];
    uv_work_t req[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        req[i].data = (void *) &data[i];
        uv_queue_work(loop, &req[i], fib, after_fib);
    }
```

```
return uv_run(loop, UV_RUN_DEFAULT);
}
```

The thread function will be launched in a separate thread, passed the uv_work_t structure and once the function returns, the after function will be called on the thread the event loop is running in. It will be passed the same structure.

For writing wrappers to blocking libraries, a common pattern is to use a baton to exchange data.

Since libuv version 0.9.4 an additional function, uv_cancel(), is available. This allows you to cancel tasks on the libuv work queue. Only tasks that are yet to be started can be cancelled. If a task has already started executing, or it has finished executing, uv_cancel() will fail.

uv_cancel() is useful to cleanup pending tasks if the user requests termination. For example, a music player may queue up multiple directories to be scanned for audio files. If the user terminates the program, it should quit quickly and not wait until all pending requests are run.

Let's modify the fibonacci example to demonstrate uv_cancel(). We first set up a signal handler for termination.

queue-cancel/main.c

```
int main() {
    loop = uv_default_loop();
```

```
int data[FIB_UNTIL];
    int i;
    for (i = 0; i < FIB_UNTIL; i++) {
        data[i] = i;
        fib_reqs[i].data = (void *) &data[i];
        uv_queue_work(loop, &fib_reqs[i], fib, after_fib);
```

```
  }


uv_signal_t sig;
    uv_signal_init(loop, &sig);
    uv_signal_start(&sig, signal_handler, SIGINT);



return uv_run(loop, UV_RUN_DEFAULT);
}
```

When the user triggers the signal by pressing Ctrl+C we send uv_cancel() to all the workers. uv_cancel() will return 0 for those that are already executing or finished.

queue-cancel/main.c

```
1
2
3
4
5
6
7
8
9
void signal_handler(uv_signal_t *req, int signum) {
printf("Signal received!\n");
int i;
for (i = 0; i < FIB_UNTIL; i++) {
uv_cancel((uv_req_t*) &fib_reqs[i]);
}
uv_signal_stop(req);
}
```

For tasks that do get cancelled successfully, the after function is called with status set to UV_ECANCELED.

queue-cancel/main.c

```
1
2
3
4
void after_fib(uv_work_t *req, int status) {
if (status == UV_ECANCELED)
fprintf(stderr, "Calculation of %d cancelled.\n", *(int *) req->data);
}
```

uv_cancel() can also be used with uv_fs_t and uv_getaddrinfo_t requests. For the filesystem family of functions, uv_fs_t.errorno will be set to UV_ECANCELED.

Tip
A well designed program would have a way to terminate long running workers that have already started executing. Such a worker could periodically check for a variable that only the main process sets to signal termination.

Inter-thread communication

Sometimes you want various threads to actually send each other messages while they are running. For example you might be running some long duration task in a separate thread (perhaps using uv_queue_work) but want to notify progress to the main thread. This is a simple example of having a download manager informing the user of the status of running downloads.

progress/main.c

```
1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
```

```
14
15
uv_loop_t *loop;
uv_async_t async;



int main() {
loop = uv_default_loop();

uv_work_t req;
    int size = 10240;
    req.data = (void*) &size;



uv_async_init(loop, &async, print_progress);
    uv_queue_work(loop, &req, fake_download, after);



return uv_run(loop, UV_RUN_DEFAULT);
}
```

The async thread communication works on loops so although any thread can be the message sender, only threads with libuv loops can be receivers (or rather the loop is the receiver). libuv will invoke the callback (print_progress) with the async watcher whenever it receives a message.

Warning

It is important to realize that since the message send is async, the callback may be invoked immediately after uv_async_send is called in another thread, or it may be invoked after some time. libuv may also combine multiple calls to uv_async_send and invoke your callback only once. The only guarantee that libuv makes is – The callback function is called at least once after the call to uv_async_send. If you have no pending calls to uv_async_send, the callback won't be called. If you make two or more calls, and libuv hasn't had a chance to run the callback yet, it may invoke your callback only once for the multiple invocations of uv_async_send. Your callback will never be called twice for just one event.

progress/main.c

```
1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
void fake_download(uv_work_t req) {
    int size = *((int) req->data);
    int downloaded = 0;
    double percentage;
    while (downloaded < size) {
        percentage = downloaded*100.0/size;
        async.data = (void*) &percentage;
        uv_async_send(&async);



sleep(1);
        downloaded += (200+random())%1000; // can only download max 1000bytes/sec,
                                           // but at least a 200;
    }
}
```

In the download function, we modify the progress indicator and queue the message for delivery with uv_async_send. Remember: uv_async_send is also non-blocking and will return immediately.

progress/main.c

```
1
2
```

```
3
4
void print_progress(uv_async_t handle) {
double percentage = *((double) handle->data);
fprintf(stderr, "Downloaded .2f%\n", percentage);
}
```

The callback is a standard libuv pattern, extracting the data from the watcher.

Finally it is important to remember to clean up the watcher.

progress/main.c

```
1
2
3
4
void after(uv_work_t req, int status) {
fprintf(stderr, "Download complete\n");
uv_close((uv_handle_t) &async, NULL);
}
```

After this example, which showed the abuse of the data field, bnoordhuis pointed out that using the data field is not thread safe, and uv_async_send() is actually only meant to wake up the event loop. Use a mutex or rwlock to ensure accesses are performed in the right order.

Note
mutexes and rwlocks DO NOT work inside a signal handler, whereas uv_async_send does.
One use case where uv_async_send is required is when interoperating with libraries that require thread affinity for their functionality. For example in node.js, a v8 engine instance, contexts and its objects are bound to the thread that the v8 instance was started in. Interacting with v8 data structures from another thread can lead to undefined results. Now consider some node.js module which binds a third party library. It may go something like this:

In node, the third party library is set up with a JavaScript callback to be invoked for more information:

```
var lib = require('lib');
lib.on_progress(function() {
console.log("Progress");
});

lib.do();

// do other stuff
```

lib.do is supposed to be non-blocking but the third party lib is blocking, so the binding uses uv_queue_work.

The actual work being done in a separate thread wants to invoke the progress callback, but cannot directly call into v8 to interact with JavaScript. So it uses uv_async_send.

The async callback, invoked in the main loop thread, which is the v8 thread, then interacts with v8 to invoke the JavaScript callback.

« Networking :: Contents :: Processes »

#3 - 04/04/2016 03:29 PM - Madars

https://nodejs.org/api/addons.html - How to write addons...

#4 - 04/04/2016 03:48 PM - Madars

http://stackoverflow.com/questions/15685793/callback-from-different-thread-in-nodejs-native-extension

#5 - 10/13/2016 07:19 AM - Madars

https://github.com/laverdet/node-fibers - some binding stuff.

#6 - 01/23/2018 03:58 PM - Madars

https://gist.github.com/matzoe/11082417

#7 - 05/10/2018 01:53 PM - Madars

https://stackoverflow.com/questions/25223294/calling-a-js-function-by-event-emitting

#8 - 05/10/2018 02:01 PM - Madars

https://scottfrees.com/ebooks/nodecpp/

#9 - 09/06/2018 02:27 PM - Madars

we could use napi: https://nodejs.org/api/n-api.html

#10 - 09/08/2018 05:47 AM - Madars

https://github.com/nodejs/node-addon-api/issues/110