Nvim core

Module-specific details are documented at the top of each module (`terminal.c` , `screen.c` , ...).

See `:help dev` for guidelines.

Filename conventions

The source files use extensions to hint about their purpose.

- `*.c` , `*.generated.c` - full C files, with all includes, etc.
- `*.c.h` - parametrized C files, contain all necessary includes, but require defining macros before actually using. Example: `typval_encode.c.h`
- `*.h` - full headers, with all includes. Does *not* apply to `*.generated.h` .
- `*.h.generated.h` - exported functions' declarations.
- `*.c.generated.h` - static functions' declarations.

Logs

Low-level log messages sink to `$NVIM_LOG_FILE` .

UI events are logged at DEBUG level (`DEBUG_LOG_LEVEL`).

```
rm -rf build/
make CMAKE_EXTRA_FLAGS="-DMIN_LOG_LEVEL=0"
```

Use `LOG_CALLSTACK()` (Linux only) to log the current stacktrace. To log to an alternate file (e.g. stderr) use `LOG_CALLSTACK_TO_FILE(FILE*)` . Requires `-no-pie` (ref):

```
rm -rf build/
make CMAKE_EXTRA_FLAGS="-DMIN_LOG_LEVEL=0 -DCMAKE_C_FLAGS=-no-pie"
```

Many log messages have a shared prefix, such as "UI" or "RPC". Use the shell to filter the log, e.g. at DEBUG level you might want to exclude UI messages:

```
tail -F ~/.cache/nvim/log | cat -v | stdbuf -o0 grep -v UI | stdbuf -o0 tee -a log
```

Build with ASAN

Building Nvim with Clang sanitizers (Address Sanitizer: ASan, Undefined Behavior Sanitizer: UBSan, Memory Sanitizer: MSan, Thread Sanitizer: TSan) is a good way to catch undefined behavior, leaks and other errors as soon as they happen. It's significantly faster than Valgrind.

Requires clang 3.4 or later, and `llvm-symbolizer` must be in `$PATH` :

```
clang --version
```

Build Nvim with sanitizer instrumentation (choose one):

```
CC=clang make CMAKE_EXTRA_FLAGS="-DCLANG_ASAN_UBSAN=ON"
CC=clang make CMAKE_EXTRA_FLAGS="-DCLANG_MSAN=ON"
```

```
CC=clang make CMAKE_EXTRA_FLAGS="-DCLANG_TSAN=ON"
```

Create a directory to store logs:

```
mkdir -p "$HOME/logs"
```

Configure the sanitizer(s) via these environment variables:

```
# Change to detect_leaks=1 to detect memory leaks (slower).
export ASAN_OPTIONS="detect_leaks=0:log_path=$HOME/logs/asan"
# Show backtraces in the logs.
export UBSAN_OPTIONS=print_stacktrace=1
export MSAN_OPTIONS="log_path=${HOME}/logs/tsan"
export TSAN_OPTIONS="log_path=${HOME}/logs/tsan"
```

Logs will be written to `${HOME}/logs/*san.PID` then.

For more information: https://github.com/google/sanitizers/wiki/SanitizerCommonFlags

Debug: Performance

Profiling (easy)

For debugging performance bottlenecks in any code, there is a simple (and very effective) approach:

1. Run the slow code in a loop.
2. Break execution ~5 times and save the stacktrace.
3. The cause of the bottleneck will (almost always) appear in most of the stacktraces.

Profiling (fancy)

For more advanced profiling, consider `perf` + `flamegraph`.

USDT profiling (powerful)

Or you can use USDT probes via `NVIM_PROBE` (#12036).

> *USDT is basically a way to define stable probe points in userland binaries. The benefit of bcc is the ability to define logic to go along with the probe points.*

Tools:

- bpftrace provides an awk-like language to the kernel bytecode, BPF.
- BCC provides a subset of C. Provides more complex logic than bpftrace, but takes a bit more effort.

Example using bpftrace to track slow vim functions, and print out any files that were opened during the trace. At the end, it prints a histogram of function timing:

```
#!/usr/bin/env bpftrace

BEGIN {
  @depth = -1;
}

tracepoint:sched:sched_process_fork /@pidmap[args->parent_pid]/ {
```

```
  @pidmap[args->child_pid] = 1;
}

tracepoint:sched:sched_process_exit /@pidmap[args->pid]/ {
  delete(@pidmap[args->pid]);
}

usdt:build/bin/nvim:neovim:eval__call_func__entry {
    @pidmap[pid] = 1;
    @depth++;
    @funcentry[@depth] = nsecs;
}

usdt:build/bin/nvim:neovim:eval__call_func__return {
    $func = str(arg0);
    $msecs = (nsecs - @funcentry[@depth]) / 1000000;

    @time_histo = hist($msecs);

    if ($msecs >= 1000) {
      printf("%u ms for %s\n", $msecs, $func);
      print(@files);
    }

    clear(@files);
    delete(@funcentry[@depth]);
    @depth--;
}

tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat {
  if (@pidmap[pid] == 1 && @depth >= 0) {
    @files[str(args->filename)] = count();
  }
}

END {
  clear(@depth);
}

$ sudo bpftrace funcslower.bt
1527 ms for Slower
@files[/usr/lib/libstdc++.so.6]: 2
@files[/etc/fish/config.fish]: 2
<snip>

^C
@time_histo:
[0]                71430 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[1]                  346 |                                                    |
[2, 4)               208 |                                                    |
[4, 8)                91 |                                                    |
```

```
[8, 16)                22 |                                                  |
[16, 32)               85 |                                                  |
[32, 64)                7 |                                                  |
[64, 128)               0 |                                                  |
[128, 256)              0 |                                                  |
[256, 512)              6 |                                                  |
[512, 1K)               1 |                                                  |
[1K, 2K)                5 |                                                  |
```

Debug: TUI

TUI troubleshoot

Nvim logs its internal terminfo state at 'verbose' level 3. This makes it possible to see exactly what terminfo values Nvim is using on any system.

```
nvim -V3log
```

TUI trace

The ancient `script` command is still the "state of the art" for tracing terminal behavior. The libvterm `vterm-dump` utility formats the result for human-readability.

Record a Nvim terminal session and format it with `vterm-dump`:

```
script foo
./build/bin/nvim -u NONE
# Exit the script session with CTRL-d

# Use `vterm-dump` utility to format the result.
./.deps/usr/bin/vterm-dump foo > bar
```

Then you can compare `bar` with another session, to debug TUI behavior.

TUI redraw

Set the 'writedelay' option to see where and when the UI is painted.

```
:set writedelay=1
```

Terminal reference

- `man terminfo`
- http://bazaar.launchpad.net/~libvterm/libvterm/trunk/view/head:/doc/seqs.txt
- http://invisible-island.net/xterm/ctlseqs/ctlseqs.html

Nvim lifecycle

Following describes how Nvim processes input.

Consider a typical Vim-like editing session:

1. Vim displays the welcome screen

2. User types: `:`
3. Vim enters command-line mode
4. User types: `edit README.txt<CR>`
5. Vim opens the file and returns to normal mode
6. User types: `G`
7. Vim navigates to the end of the file
8. User types: `5`
9. Vim enters count-pending mode
10. User types: `d`
11. Vim enters operator-pending mode
12. User types: `w`
13. Vim deletes 5 words
14. User types: `g`
15. Vim enters the "g command mode"
16. User types: `g`
17. Vim goes to the beginning of the file
18. User types: `i`
19. Vim enters insert mode
20. User types: `word<ESC>`
21. Vim inserts "word" at the beginning and returns to normal mode

Note that we split user actions into sequences of inputs that change the state of the editor. While there's no documentation about a "g command mode" (step 16), internally it is implemented similarly to "operator-pending mode".

From this we can see that Vim has the behavior of an input-driven state machine (more specifically, a pushdown automaton since it requires a stack for transitioning back from states). Assuming each state has a callback responsible for handling keys, this pseudocode represents the main program loop:

```
def state_enter(state_callback, data):
  do
    key = readkey()                 # read a key from the user
  while state_callback(data, key)   # invoke the callback for the current state
```

That is, each state is entered by calling `state_enter` and passing a state-specific callback and data. Here is a high-level pseudocode for a program that implements something like the workflow described above:

```
def main()
  state_enter(normal_state, {}):

def normal_state(data, key):
  if key == ':':
    state_enter(command_line_state, {})
  elif key == 'i':
    state_enter(insert_state, {})
  elif key == 'd':
    state_enter(delete_operator_state, {})
  elif key == 'g':
    state_enter(g_command_state, {})
  elif is_number(key):
```

```
    state_enter(get_operator_count_state, {'count': key})
  elif key == 'G'
    jump_to_eof()
  return true

def command_line_state(data, key):
  if key == '<cr>':
    if data['input']:
      execute_ex_command(data['input'])
    return false
  elif key == '<esc>'
    return false

  if not data['input']:
    data['input'] = ''

  data['input'] += key
  return true

def delete_operator_state(data, key):
  count = data['count'] or 1
  if key == 'w':
    delete_word(count)
  elif key == '$':
    delete_to_eol(count)
  return false  # return to normal mode

def g_command_state(data, key):
  if key == 'g':
    go_top()
  elif key == 'v':
    reselect()
  return false  # return to normal mode

def get_operator_count_state(data, key):
  if is_number(key):
    data['count'] += key
    return true
  unshift_key(key)  # return key to the input buffer
  state_enter(delete_operator_state, data)
  return false

def insert_state(data, key):
  if key == '<esc>':
    return false  # exit insert mode
  self_insert(key)
  return true
```

The above gives an idea of how Nvim is organized internally. Some states like the `g_command_state` or `get_operator_count_state` do not have a dedicated `state_enter` callback, but are implicitly embedded

into other states (this will change later as we continue the refactoring effort). To start reading the actual code, here's the recommended order:

1. `state_enter()` function (state.c). This is the actual program loop, note that a `VimState` structure is used, which contains function pointers for the callback and state data.
2. `main()` function (main.c). After all startup, `normal_enter` is called at the end of function to enter normal mode.
3. `normal_enter()` function (normal.c) is a small wrapper for setting up the NormalState structure and calling `state_enter`.
4. `normal_check()` function (normal.c) is called before each iteration of normal mode.
5. `normal_execute()` function (normal.c) is called when a key is read in normal mode.

The basic structure described for normal mode in 3, 4 and 5 is used for other modes managed by the `state_enter` loop:

- command-line mode: `command_line_{enter,check,execute}()` (`ex_getln.c`)
- insert mode: `insert_{enter,check,execute}()` (`edit.c`)
- terminal mode: `terminal_{enter,execute}()` (`terminal.c`)

Async event support

One of the features Nvim added is the support for handling arbitrary asynchronous events, which can include:

- RPC requests
- job control callbacks
- timers

Nvim implements this functionality by entering another event loop while waiting for characters, so instead of:

```
def state_enter(state_callback, data):
  do
    key = readkey()                 # read a key from the user
  while state_callback(data, key)   # invoke the callback for the current state
```

Nvim program loop is more like:

```
def state_enter(state_callback, data):
  do
    event = read_next_event()       # read an event from the operating system
  while state_callback(data, event) # invoke the callback for the current state
```

where `event` is something the operating system delivers to us, including (but not limited to) user input. The `read_next_event()` part is internally implemented by libuv, the platform layer used by Nvim.

Since Nvim inherited its code from Vim, the states are not prepared to receive "arbitrary events", so we use a special key to represent those (When a state receives an "arbitrary event", it normally doesn't do anything other update the screen).

Main loop

The `Loop` structure (which describes `main_loop`) abstracts multiple queues into one loop:

```
uv_loop_t uv;
MultiQueue *events;
MultiQueue *thread_events;
MultiQueue *fast_events;
```

`loop_poll_events` checks `Loop.uv` and `Loop.fast_events` whenever Nvim is idle, and also at `os_breakcheck` intervals.

MultiQueue is cool because you can attach throw-away "child queues" trivially. For example `do_os_system()` does this (for every spawned process!) to automatically route events onto the `main_loop`:

```
Process *proc = &uvproc.process;
MultiQueue *events = multiqueue_new_child(main_loop.events);
proc->events = events;
```