printf() and stdio in the Julia runtime

Libuv wrappers for stdio

`julia.h` defines libuv wrappers for the `stdio.h` streams:

```
uv_stream_t *JL_STDIN;
uv_stream_t *JL_STDOUT;
uv_stream_t *JL_STDERR;
```

... and corresponding output functions:

```
int jl_printf(uv_stream_t *s, const char *format, ...);
int jl_vprintf(uv_stream_t *s, const char *format, va_list args);
```

These `printf` functions are used by the `.c` files in the `src/` and `ui/` directories wherever stdio is needed to ensure that output buffering is handled in a unified way.

In special cases, like signal handlers, where the full libuv infrastructure is too heavy, `jl_safe_printf()` can be used to `write(2)` directly to `STDERR_FILENO`:

```
void jl_safe_printf(const char *str, ...);
```

Interface between JL_STD* and Julia code

`Base.stdin`, `Base.stdout` and `Base.stderr` are bound to the `JL_STD*` libuv streams defined in the runtime.

Julia's `__init__()` function (in `base/sysimg.jl`) calls `reinit_stdio()` (in `base/stream.jl`) to create Julia objects for `Base.stdin`, `Base.stdout` and `Base.stderr`.

`reinit_stdio()` uses `ccall` to retrieve pointers to `JL_STD*` and calls `jl_uv_handle_type()` to inspect the type of each stream. It then creates a Julia `Base.IOStream`, `Base.TTY` or `Base.PipeEndpoint` object to represent each stream, e.g.:

```
$ julia -e 'println(typeof((stdin, stdout, stderr)))'
```

```
Tuple{Base.TTY,Base.TTY,Base.TTY}

$ julia -e 'println(typeof((stdin, stdout, stderr)))' < /dev/null 2>/dev/null
Tuple{IOStream,Base.TTY,IOStream}

$ echo hello | julia -e 'println(typeof((stdin, stdout, stderr)))' | cat
Tuple{Base.PipeEndpoint,Base.PipeEndpoint,Base.TTY}
```

The `Base.read` and `Base.write` methods for these streams use `ccall` to call libuv wrappers in `src/jl_uv.c`, e.g.:

```
stream.jl: function write(s::IO, p::Ptr, nb::Integer)
               -> ccall(:jl_uv_write, ...)
  jl_uv.c:          -> int jl_uv_write(uv_stream_t *stream, ...)
                        -> uv_write(uvw, stream, buf, ...)
```

printf() during initialization

The libuv streams relied upon by `jl_printf()` etc., are not available until midway through initialization of the runtime (see `init.c`, `init_stdio()`). Error messages or warnings that need to be printed before this are routed to the standard C library `fwrite()` function by the following mechanism:

In `sys.c`, the `JL_STD*` stream pointers are statically initialized to integer constants: `STD*_FILENO (0, 1 and 2)`. In `jl_uv.c` the `jl_uv_puts()` function checks its `uv_stream_t* stream` argument and calls `fwrite()` if stream is set to `STDOUT_FILENO` or `STDERR_FILENO`.

This allows for uniform use of `jl_printf()` throughout the runtime regardless of whether or not any particular piece of code is reachable before initialization is complete.

Legacy `ios.c` library

The `src/support/ios.c` library is inherited from femtolisp. It provides cross-platform buffered file IO and in-memory temporary buffers.

`ios.c` is still used by:

- `src/flisp/*.c`
- `src/dump.c` – for serialization file IO and for memory buffers.
- `src/staticdata.c` – for serialization file IO and for memory buffers.
- `base/iostream.jl` – for file IO (see `base/fs.jl` for libuv equivalent).

Use of `ios.c` in these modules is mostly self-contained and separated from the libuv I/O system. However, there is one place where femtolisp calls through to `jl_printf()` with a legacy `ios_t` stream.

There is a hack in `ios.h` that makes the `ios_t.bm` field line up with the `uv_stream_t.type` and ensures that the values used for `ios_t.bm` to not overlap with valid `UV_HANDLE_TYPE` values. This allows `uv_stream_t` pointers to point to `ios_t` streams.

This is needed because `jl_printf()` caller `jl_static_show()` is passed an `ios_t` stream by femtolisp's `fl_print()` function. Julia's `jl_uv_puts()` function has special handling for this:

```
if (stream->type > UV_HANDLE_TYPE_MAX) {
    return ios_write((ios_t*)stream, str, n);
}
```

« Working with LLVM Bounds checking »