Emscripten Malloc

The main files are as follows:

- entry.c contains the C code used to compiled with Emscripten and extract the code. Command used for compilation:

  ```
  emcc entry.c -s WASM=1 -o entry.js  -s ALLOW_MEMORY_GROWTH=1 -O2
  ```

- entry.wat contains the file generated by Emscripten using opt -O2
- entry.js contains the JS library support from Emscripten
- malloc_support_lib.js contains the extracted support for malloc from entry.js.
- malloc.wat contains the extracted support for malloc from entry.wat.
- The *wasm* files contain the actual wasm code for the .wat files.

To extract the corresponding files, we transformed the .wasm files generated into text format and extracted all the necessary function implementations. For more information contact: davidfherrerar@gmail.com

`_malloc` - Function 22

Lines: (Lines: 194 - 6234)

External Dependencies:

- Global 12 -> Stack Top
- Global 13 -> Stack Max
- Call 3 -> `abortStackOverflow` - Imported
- Call 71 -> `_sbrk` - Impletemented
- Call 28 ->`__errno_location` - Implemented

In terms of handling the stack, this function asks for 80 bytes from the stack. Before this it checks the global 13 for stack max to see whether there is space. If there is, it sets the STACKTOP to STACKTOP+80, and when the function returns it returns the stack to its original value. Moreover, it handles the stack address from then on via a local, which it sets at the beginning from the global. This saves loading of globals which is more expensive. In -03, the whole stack handling calls dissappears as is not necessary.

`_sbrk` - Function 71

Lines: (Lines: 18198 - 18249)

Grows memory when necessary, checks if there is enough memory, and if the parameter passed is appropriate if not, it throws an error. Returns the value of the `DYNAMIC_PTR` in memory **External Dependencies**:

- Global 9 - DYNAMICTOP_PTR

- Call 2 - `abortOnCannotGrowMemory` - Imported
- Call 7 - `__setErrNo` - Imported (Sets error in external emscripten state, CODE:12)
- Call 1 - `getTotalMemory` - Imported
- Call 0 - `enlargeMemory` - Imported **Return**: Returns ptr to start of payload

`__errono_location`

- Does not do anything, simply returns the address where the errors are saved, it seems like all the errors are thrown by the outside JavaScript environment, it is not clear at the time how this is used, as I see no calls being made to it. In zero opt flag is 1648, in -O1 ~1560, in -O2 is 1520

```
(func (;29;) (type 2) (result i32)
  (local i32 i32)
  get_global 12
  set_local 1
  i32.const 1520 // Address of errors
  return)
```

dependencies

- Global 12 -> `STACK_TOP`

Free

Frees chunks

- Global 12 ->`STACK_TOP`

Entry.js

All starts with a constant `GLOBAL_BASE` ptr. The memory is organized as follows:

- Above 1024 is static memory, starting with globals ptrs. (Example location of stack top, stack base, and start of dynamic memory).
- Then the stack.
- Then 'dynamic' memory for `sbrk`

```
GLOBALBASE = 1024;
```

Static memory

Meant for globals, and positions of ptrs in the program

```
// Line: 1730
STATICBASE = GLOBALBASE;
STATICTOP = STATICBASE + 5456;// Setting the top
STATICBUMP = 5456; //size of static memory
```

They weirdly then statically allocate for some other poiters. This functions are never called for this program. I got rid of them for my puposes

```
var tempDoublePtr = STATICTOP; STATICTOP += 16;
```

`DYNAMICTOP_PTR = staticAlloc(4)` // Sets the ptr where the location of dynamic memory starts.

After it sets the location where the `DYNAMICTOP_PTR` constant will be stored, it sets `STACK_BASE`, `STACKTOP`,`DYNAMIC_BASE`, finally it sets the dynamic ptr in the `DYNAMICTOP_PTR` location.

```
TOTAL_STACK=5242880 // Could be set via MODULE['TOTAL_STACK'];
STACK_BASE = STACKTOP = alignMemory(STATICTOP);
STACK_MAX = STACK_BASE + TOTAL_STACK;
DYNAMIC_BASE = alignMemory(STACK_MAX);
HEAP32[DYNAMICTOP_PTR>>2] = DYNAMIC_BASE;
```

`staticAlloc(size)`

- Uses STATICTOP allocate statically a given size, makes sure the result is 16 byte aligned.

```
function staticAlloc(size) {

  assert(!staticSealed);
  var ret = STATICTOP;
  STATICTOP = (STATICTOP + size + 15) & -16;
  return ret;
}
```

`alignMemory()`

- Aligns 16 byte align of memory, by taking ceiling, so in both cases above it increases the pointer by 16

```
const STACK_ALIGN = 16;
function alignMemory(size, factor) {
  if (!factor) factor = STACK_ALIGN; // stack alignment (16-byte) by default
  var ret = size = Math.ceil(size / factor) * factor;
  return ret;
}
```

In terms of Emscripten, there are three functions to allocate memory, they all come into play depending on the Runtime initialization. This all materializes in a function called `getMemory()`

```
function getMemory(size) {
  if (!staticSealed) return staticAlloc(size);
  if (!runtimeInitialized) return dynamicAlloc(size);
  return _malloc(size);
}
```

If the pointers for the stack and dynamic ptr is not ready, then staticAlloc, if the `runtimeInitialized` is false, which means the `_malloc` function from emscripten is not ready, then use `dynamicAlloc(size)`

`abortStackOverflow(allocSize)`

Function simple throws error when there is no more space in the stack. Thrown by every function if its the case. Note that Emscripten is smart enough to eliminate this call when it can statically determined this is not true.

```
function abortStackOverflow(allocSize) {
  abort('Stack overflow! Attempted to allocate ' + allocSize + ' bytes on
the stack, but stack has only ' + (STACK_MAX - stackSave() + allocSize) +
' bytes available!');
}
```

`__setErrNo(num)`

```
function ___setErrNo(value) {
      if (Module['___errno_location'])
HEAP32[((Module['___errno_location']())>>2)]=value;
      else Module.printErr('failed to set errno from JS');
      return value;
 }
```

This function simply sets the place for the error location inside the Emscripten WebAssembly memory. i.e. Each C function has a set of string errors it throws under particular circumstances, this functions just sets the location of those functions in memory using the `__errno_location` base address.

`getTotalMemory()`

- Simply returns the total_memory for the program, the emscripten program gives the user the option to either set this total memory, allow for the growth of it, etc. **dependencies**
- `TOTAL_MEMORY`

`abortOnCannotGrowMemory`

dependencies

- abort()
- `TOTAL_MEMORY`

```
function abortOnCannotGrowMemory() {
  abort('Cannot enlarge memory arrays. Either (1) compile with  -s
TOTAL_MEMORY=X  with X higher than the current value ' + TOTAL_MEMORY + ',
(2) compile with  -s ALLOW_MEMORY_GROWTH=1  which allows increasing the
size at runtime, or (3) if you want malloc to return NULL (0) instead of
this abort, compile with  -s ABORTING_MALLOC=0 ');
}
```

`abortOnStackOverflow()`

- Quits application when a stackoverflow happens **dependencies**:
- `abort()`

```
function abortStackOverflow(allocSize) {
  abort('Stack overflow! Attempted to allocate ' + allocSize + ' bytes on
the stack, but stack has only ' + (STACK_MAX - stackSave() + allocSize) +
' bytes available!');
}
```

`enlargeMemory()`

- Grows the Emscripten memory, now this is the most complicated functions, this functions has many dependencies and roles. **depedencies**
- `HEAP32[DYNAMIC\_PTR>>2]`
- `TOTAL_MEMORY`
- `Module["usingWasm"]`
- `WASM_PAGE_SIZE` and `ASMJS_PAGE_SIZE`
- `Module.printErr()`
- `alignUp()`
- `Module.reallocBuffer(TOTAL_MEMORY)` Reallocates all memory, returns old if not possible.
- `updateGlobalBuffer()`
- `updateGlobalBufferViews()`
- `Module.usingWasm`

1. Checks the `DYNAMIC_PTR`, it should have been increased by now, sets the limit to one page short of 2gb
2. Checks to see if the `HEAP[DYNAMIC_PTR>>2] > LIMIT`, if it is, throws error.
3. Then proceeds to increase `TOTAL_MEMORY` towards 2GBs, if possible.
4. Calls `Module.reallocBuffer(TOTAL_MEMORY)` to reallocate the buffer, and copy the data from the old to the new buffer.
5. If error, prints error using `Module.printErr`
6. Otherwise it calls `updateGlobalBuffer()` and `updateGlobalBufferViews()`
7. Prints new memory, warning if is asmjs and finally returns.

`updateGlobalBuffer` & `updateGlobalBufferViews()`

```
function updateGlobalBuffer(buf) {
  Module['buffer'] = buffer = buf;
}

function updateGlobalBufferViews() {
  Module['HEAP8'] = HEAP8 = new Int8Array(buffer);
  Module['HEAP16'] = HEAP16 = new Int16Array(buffer);
  Module['HEAP32'] = HEAP32 = new Int32Array(buffer);
  Module['HEAPU8'] = HEAPU8 = new Uint8Array(buffer);
  Module['HEAPU16'] = HEAPU16 = new Uint16Array(buffer);
  Module['HEAPU32'] = HEAPU32 = new Uint32Array(buffer);
  Module['HEAPF32'] = HEAPF32 = new Float32Array(buffer);
  Module['HEAPF64'] = HEAPF64 = new Float64Array(buffer);
}
```

`reallocBuffer`

```
  var wasmReallocBuffer = function(size) {
    var PAGE_MULTIPLE = Module["usingWasm"] ? WASM_PAGE_SIZE : ASMJS_PAGE_SIZE; // In wasm, heap size must be a multiple of 64KB. In asm.js, they need to be multiples of 16MB.
    size = alignUp(size, PAGE_MULTIPLE); // round up to wasm page size
    var old = Module['buffer'];
    var oldSize = old.byteLength;
    if (Module["usingWasm"]) {
      // native wasm support
      try {
        var result = Module['wasmMemory'].grow((size - oldSize) / wasmPageSize); // .grow() takes a delta compared to the previous size
        if (result !== (-1 | 0)) {
          // success in native wasm memory growth, get the buffer from the memory
          return Module['buffer'] = Module['wasmMemory'].buffer;
        } else {
          return null;
        }
      } catch(e) {
        console.error('Module.reallocBuffer: Attempted to grow from ' + oldSize  + ' bytes to ' + size + ' bytes, but got error: ' + e);
        return null;
      }
    }
  };
```

MEMORY SET UP

```
var byteLength;
try {
  byteLength = Function.prototype.call.bind(Object.getOwnPropertyDescriptor(ArrayBuffer.prototype, 'byteLength').get);
  byteLength(new ArrayBuffer(4)); // can fail on older ie
} catch(e) { // can fail on older node/v8
  byteLength = function(buffer) { return buffer.byteLength; };
}

var TOTAL_STACK = Module['TOTAL_STACK'] || 5242880;
var TOTAL_MEMORY = Module['TOTAL_MEMORY'] || 16777216;
if (TOTAL_MEMORY < TOTAL_STACK) Module.printErr('TOTAL_MEMORY should be larger than TOTAL_STACK, was ' + TOTAL_MEMORY + '! (TOTAL_STACK=' + TOTAL_STACK + ')');

// Initialize the runtime's memory
// check for full engine support (use string 'subarray' to avoid closure compiler confusion)
assert(typeof Int32Array !== 'undefined' && typeof Float64Array !== 'undefined' && Int32Array.prototype.subarray !== undefined && Int32Array.prototype.set !== undefined,
       'JS engine does not provide full typed array support');



// Use a provided buffer, if there is one, or else allocate a new one
if (Module['buffer']) {
  buffer = Module['buffer'];
  assert(buffer.byteLength === TOTAL_MEMORY, 'provided buffer should be ' + TOTAL_MEMORY + ' bytes, but it is ' + buffer.byteLength);
} else {
  // Use a WebAssembly memory where available
  if (typeof WebAssembly === 'object' && typeof WebAssembly.Memory === 'function') {
    assert(TOTAL_MEMORY % WASM_PAGE_SIZE === 0);
    Module['wasmMemory'] = new WebAssembly.Memory({ 'initial': TOTAL_MEMORY / WASM_PAGE_SIZE });
    buffer = Module['wasmMemory'].buffer;
  } else
```

```
      {
        buffer = new ArrayBuffer(TOTAL_MEMORY);
      }
      assert(buffer.byteLength === TOTAL_MEMORY);
      Module['buffer'] = buffer;
    }
    updateGlobalBufferViews();
```

Error Messages in Static Memory

Another consideration when dealing with WebAssembly is the error strings that are written down at start-up in static memory via the `data` WebAssembly construct. When dealing with your own library use the static offset to set the limit of your constant messages. Also, to the top of your static offset, make sure to set the `__errno_location()` function, (which could be set statically in JavaScript, if you only have one version of `wasm` This way it does not have to be imported from the WebAssembly instantiated module.).