Threading in the browser: Webassembly

Michaël Van Canneyt

April 21, 2025

Abstract

In this article, we explain how threading works in a webassembly program running in the browser, and what you need to do to be able to run a threaded FPC program in the browser.

1 Introduction

In a previous article we explained how to run multiple processes in the browser: the web workers. These are separate processes controlled by the browser which can communicate between each other. In this article, we'll explain how the web workers can be used to enable threading in webassembly.

We'll start with the good news: the webassembly specification has support for threading. In webassembly itself, the support for threading is mostly limited to some atomic instructions: there are the usual atomic increment, decrement, compare-and-exchange instructions, and a description of how to manage memory (the stack and linear memory - the heap memory) and some extra data that must be present in the webassembly module. It also describes a set of notify/wait instructions that can be used to implement thread synchronisation mechanisms.

But for the most part, the management of threading is left to the hosting environment used to run the webassembly code.

The WASI specification describes a standard for interacting with the host environment: functions to get time, read and write to files etc. The WASI specification also has a part concerned with threading: it describes a 'start thread' function. Any webassembly that is hosted in an environment that offers the WASI API will be able to run threads.

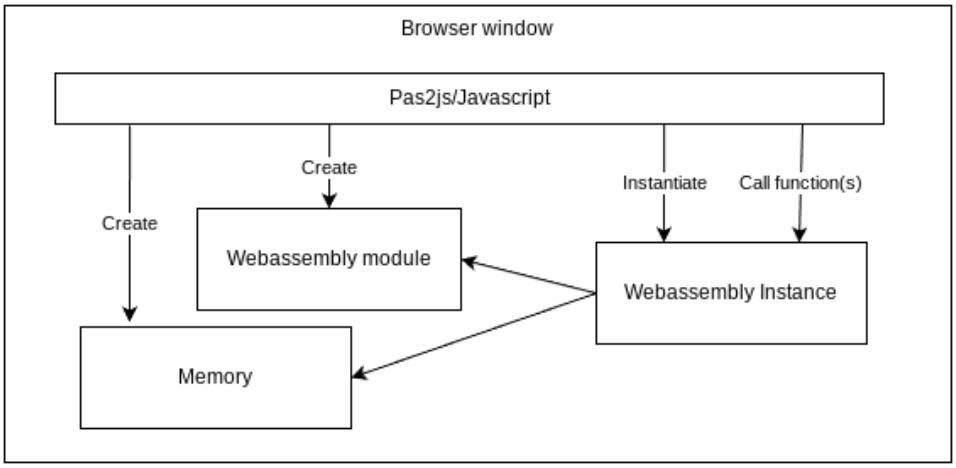
There is also some bad news: a webassembly module with support for threading is not compatible with a webassembly without support for threading. Practically, this means that if you want to compile your program with support for threading, you must recompile your webassembly module. The same module cannot be used to run both with or without threading support.

In Free Pascal, this means that if you want to use threading, you must recompile your code for another target. If you use a `TThread` descendent to run a task in a separate thread, and you compile without threading support, trying to execute the thread will raise an exception. If you compiled with threading support, then the thread will be executed.

For the atomic operations, these will work in threading and non-threading modes, except for the wait/notify instructions.

Practically, compiling without threading means you choose the `wasip1` target, and

Figure 1: Executing webassembly without threading



if you wish to support threading, you compile for the `wasip1threads` target. The `ppcrosswasm32` compiler supports both targets.

So why not simply always compile for the `wasip1threads` target ? Well, for starters, not all hosting environments support threading. It also imposes constraints on available memory: the maximum available memory must be specified in the webassembly module, and is therefore determined at compile time.

Fortunately, the browser does support threaded webassembly, but as well see shortly, the support comes with quite a price.

2 Threading support in the browser

Before explaining how threading for webassembly can be enabled in the browser, we'll review the 'regular' webassembly support of the browser.

Executing a non-threaded webassembly program is done in several steps:

1. Download the webassembly module.
2. Compile the webassembly module.
3. Create webassembly memory (actually, a memory descriptor is created, from which the browser will create the actual memory): A webassembly memory is basically an array of bytes, where a 'pointer' in the webassembly program is just an index in the array.
4. Create a webassembly instance: this is a combination of a webassembly module, memory and imported functions. (one could instantiate several instances of the same module, each time with a different memory and imported functions)
5. Execute the functions exported from the webassembly module: this can be a complete program (a function called start), or just an exported function from a library.

All this is schematically depicted in figure 1 on page 2. The browser has no di-

rect Javascript support for executing a function in a thread. Instead, support for threading can be emulated using web workers.

Remember that web workers are separate processes, and that we can send messages from one worker to another. We can also transfer objects from one worker to another. Also, the browser has support for shared memory and atomic operations on the shared memory. These 2 possibilities can be used to enable threading support for the webassembly module.

The first steps to run a threaded webassembly are the same:

1. Download the webassembly module.
2. Compile the webassembly module.
3. Create shared webassembly memory: This is the same as in the non-threaded case, but you specify that the memory must be shared.

The next steps differ:

4. Start several web workers: start as many workers as you want to allow simultaneous threads. These workers should be able to receive a webassembly module and memory, and should listen for a message to execute a thread function (see below).
5. Send the compiled webassembly module and the shared memory to these workers, using a message.

When that is done, we can start the main thread:

1. Create a webassembly instance. In the list of imported functions for the module, the WASI start thread function must be present.
2. Execute the functions exported from the webassembly module: this can be a complete program (a function called start), or just an exported function from a library.
3. whenever the webassembly module calls the WASI start thread function, a message must be sent to one of the workers, telling it to execute the thread function.

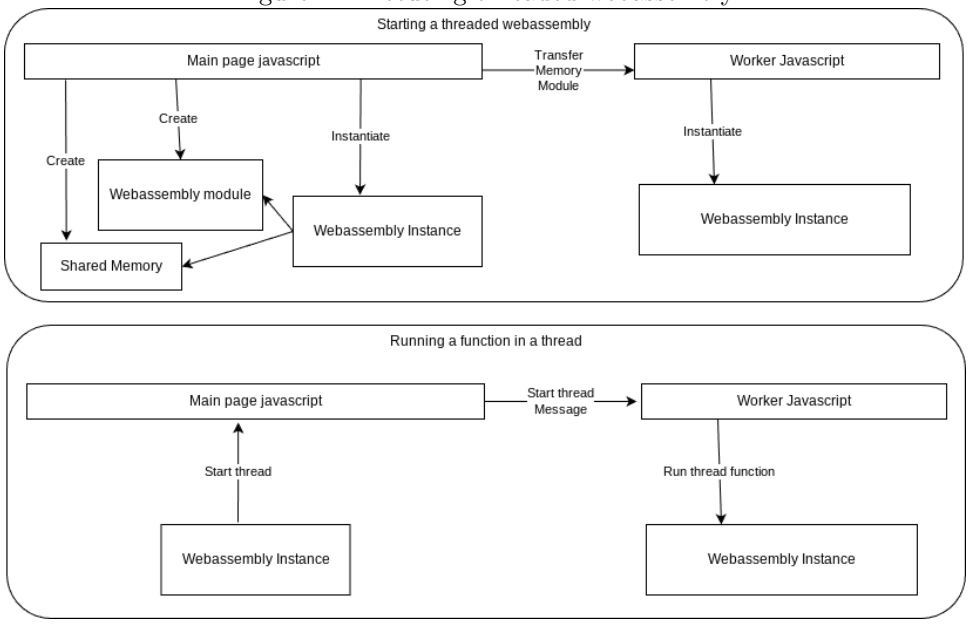
The workers do the following:

1. Create a webassembly instance with the received webassembly module and shared memory. The same import functions must be made available
2. They run an infinite loop, in the loop the following happens:
   (a) wait for a message to execute the 'run thread function' exported by the webassembly.
   (b) When the 'run thread function' returns, the webassembly thread has finished executing, and the result is reported to the main javascript through a message.

This process is depicted in figure 2 on page 4.

All this requires quite some bookkeeping in the main thread: it needs to keep track of which workers are currently busy, so when the 'start thread' function is called, it

Figure 2: Executing threaded webassembly



can decide which web worker will need to handle the thread. It must also pass on the necessary info for the web worker to know which function should be executed.

Also, if a thread (so a worker) wants to start another thread, this action must be passed on to the main page, as that acts as the orchestrator for all threads.

Basically, the thread handling is completely implemented in Javascript, and requires quite a lot of message handling and web worker management.

The constraints in this scheme are maybe not so obvious, so we'll make them explicit:

1. Shared memory cannot be resized at will: a maximum size must be specified, and it must fit within the limits specified in the webassembly module. The browser will complain if the memory does not fit. That makes it difficult to make a general hosting environment, capable of executing any threaded webassembly module.

2. The number of threads that can run simultaneously is limited by the number of workers that is started. One could imagine a dynamic scheme where more workers are started on an as-needed basis, but a worker process is in essence a separate browser process that takes up more memory than a mere thread.

3 Webassembly threading in FPC

So how can you create a threaded webassembly application in FPC ?

In code, you can use the usual `TThread`, `TCriticalSection` classes, as well as the `InterlockedIncrement` and other atomic instructions: there is no difference with native code.

However, when compiling your code, you will need the Free Pascal compiler compiled from the git "main" branch. When compiling, you must change the target OS from `wasip1` (which is the new default target for webassembly) to `wasip1threads`.

The `wasip1` and `wasip1threads` are 2 different operating systems as far as the compiler is concerned. You must therefore have compiled all of the RTL and packages for this target.

If you already have a webassembly cross-compiler, then you can execute the following commands in the directory with the FPC sources:

```
cd rtl
make clean all OS_TARGET=wasip1threads CPU_TARGET=wasm32 PP=ppcrosswasm32
make install OS_TARGET=wasip1threads CPU_TARGET=wasm32 PP=ppcrosswasm32
cd ../packages
make clean all OS_TARGET=wasip1threads CPU_TARGET=wasm32 PP=ppcrosswasm32
make install OS_TARGET=wasip1threads CPU_TARGET=wasm32 PP=ppcrosswasm32
```

If you do not yet have a webassembly cross-compiler, then you can execute the following command in the directory with the FPC sources:

```
make clean all OS_TARGET=wasip1 CPU_TARGET=wasm32 BINUTILSPREFIX= OPT="-O-" PP=fpc
make install OS_TARGET=wasip1 CPU_TARGET=wasm32 BINUTILSPREFIX= OPT="-O-" PP=fpc
```

This will create the cross compiler and the rtl for threaded applications.

If you use the latest Lazarus IDE, you can then set the target to 'wasip1threads' in the IDE, and then recompile your application.

We mentioned that WebAssembly thread support requires the program to specify the minimum and maximum amounts of memory it is going to use. The FPC compiler specifies 256Mb as the maximum amount of memory for multithreaded applications and 8Mb for the default stack size.

But you can change these values using the `$M` directive, which you must insert in your program file:

```
{$M stacksize,initialmem,maxmem}
```

For example, to specify 1 Mb stack size, 256Mb initial memory, and 512Mb max memory, you can write:

```
{$M 1048576,268435456,536870912}
```

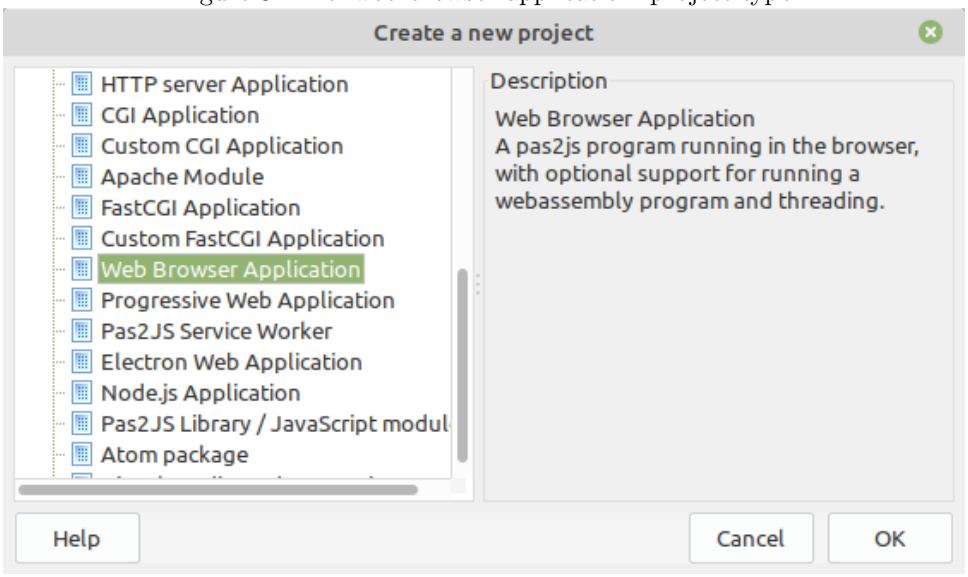
Beware: if you specify a really high value for the maximum amount of memory, this may result in wasted memory, (mobile browsers may not have so much memory). Some WebAssembly hosts (notably browsers) may reject to load the module completely if the amount of memory is too high. Setting the value too low can result in out of memory errors in the Free Pascal program. Using the `-gh` compiler switch offers some statistics when the program exits, it may be helpful to determine the limits.

This constraint (the need to specify the amount of memory in advance) may be lifted in a future specification of Webassembly, but for the moment the amount of memory must be specified at compile-time if the defaults do not suit your particular situation.

4 Webassembly threading in Pas2JS

Since the appearance of the FPC webassembly compiler, Pas2js provides a hosting environment for webassembly: you must create a browser application that provides

Figure 3: The 'web browser application' project type



the necessary APIs to the webassembly program. If you use the `TBrowserWASIHostApplication` class as the parent class of your application class, then the WASI version 1 API is made available to your webassembly.

In order to run various threads, the WASI version 1 threading API must be provided. Pas2js provides this API as well, but it is not enabled by default.

If you have an existing application which you want to convert to a threading-enabled application, you must make 2 changes:

1. change the `WasiHostApp` unit in your program's uses clause to `WASIThreadedApp`.
2. change the parent class of your application class (`TWASIHostApplication`) in your program's from uses clause to `TBrowserWASIThreadedHostApplication`.

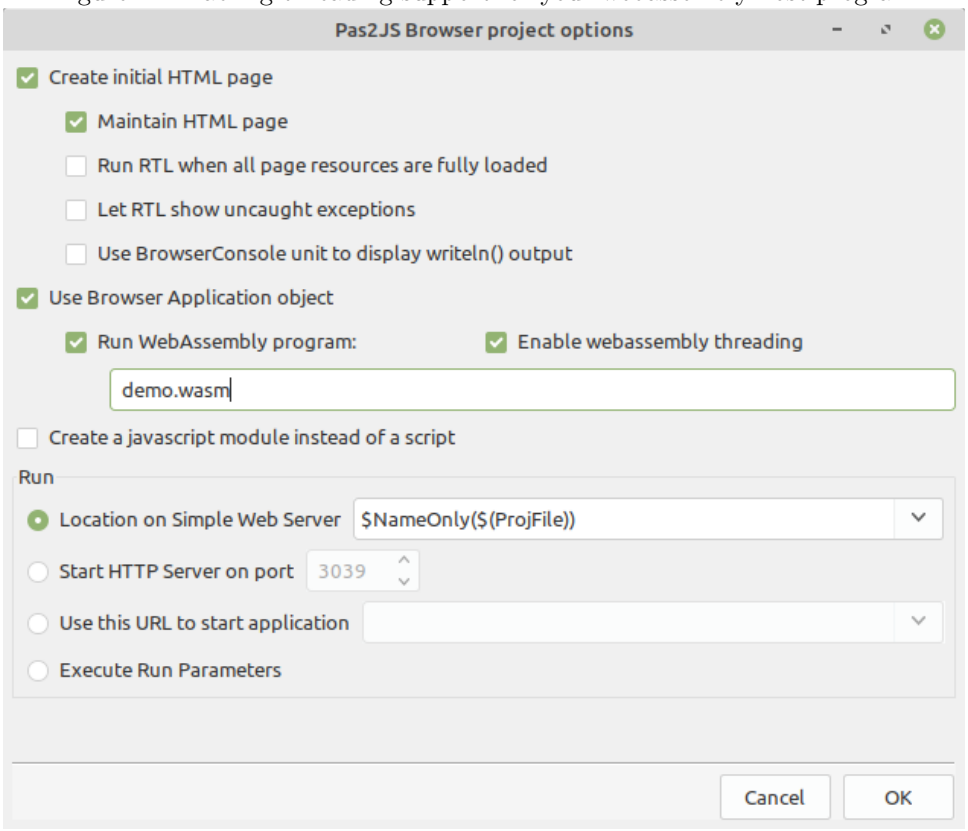
The reason for this change is that in order limit the size of the hosting app, the threading support is not built in into the standard `TWASIHostApplication`. Instead, a separate application class exists which has the threading support built-in.

If you create a new application in the Lazarus IDE, then the 'Web Browser application' project type in the 'New project' dialog (figure 3 on page 6) now has support for creating a webassembly host program that supports threaded webassembly: you can enable support for threading simply by checking a checkbox (see figure 4 on page 7). The lazarus IDE will handle the above changes for you.

As discussed above, the threads are executed in a web worker. The `TBrowserWASIThreadedHostApplication` will start a standard worker script, which will execute the thread run function when a new thread is started. This worker script is the `pas2jsthreadworker.js` script. It is a pas2js program which you can compile using pas2js called`pas2jsthreadworker.pas`, it is located in the `packages/wasi/worker` directory of the Pas2js sources.

The reason that you must compile the web worker yourself is simply the way webassembly works: When instantiating a webassembly module, the host environment must provide all the 'imported APIs' of the webassembly module. If your webassembly module requires a set of functions to handle HTTP requests, then the hosting environment must provide these functions, with the correct names and signatures.

Figure 4: Enabling threading support for your webassembly host program



It follows that when a worker script instantiates a webassembly module in order to execute thread functions, the worker script must provide the same APIs as the main program provides.

So if your main program provides an API for executing HTTP requests, then your web worker program will also need to provide these same functions that handle HTTP requests.

The default worker script only provides the WASI API and the threading API. If you need additional APIs (such as an API to execute HTTP requests) you must add the implementation of these APIs to the webworker program, and recompile it.

To demonstrate all this, we'll create a small example: we'll create a library that calculates a fibonacci number in a thread. The following routine calculates the fibonacci number N:

```
Function Fibonacci(N : Integer) : Int64;

Var
  Next,Last : Int64;
  I : Integer;

begin
  if N=0 then
    exit(0);
  Result:=1;
  Last:=0;
  for I:=1 to N-1 do
    begin
    Next:=Result+last;
    Last:=Result;
    Result:=Next;
    end;
end;
```

This is not something you would normally need to calculate in a thread, but it serves just to demonstrate the use of threads. So, the next thing we need is a `TThread` class that calls this function:

```
Type
  { TCalcThread }
  TCalcThread = Class(TThread)
  Private
    FN : Integer;
  Public
    constructor create(N : Integer);
    Procedure Execute; override;
  end;

{ TCalcThread }
constructor TCalcThread.Create(N: Integer);
begin
  FN:=N;
  Inherited Create(False);
end;
```

```
procedure TCalcThread.Execute;
begin
  FreeOnTerminate:=True;
  Writeln('Fibonacci(',FN,') = ',(Fibonacci(FN));
end;
```

The thread will destroy itself when done. We put all this in a unit called 'CalcFib'. Lastly, we create a library with a single exported function, which looks like this:

```
library threaddemo;

uses CalcFib;

procedure CalcFibonacci(N : Integer);

begin
  Writeln('Starting thread');
  With TCalcThread.Create(N) do
    begin
    Writeln('Thread created');
    end;
end;

exports CalcFibonacci;

begin
end.
```

You can compile the library in lazarus, or on the command-line:

```
ppcrosswasm32 -Twasip1threads threaddemo.lpr -othreaddemo.wasm
```

The webpage hosting part for this webassembly module we can create using the 'Web Browser Application' wizard: we ask for a HTML page, console output, and of course the loading of a webassembly In the created HTML, we add a reference to Bulma CSS and insert the following HTML to show an edit and a button, a click on the button should execute the fibonacci calculation in a thread:

```
  <div class="section pb-4">
    <h1 class="title is-4">Threaded Fibonacci:</h1>
  </div>
  <div class="section pb-4">
    <div class="columns">
      <div class="column is-one-quarter">
        <div class="field has-addons">
          <div class="control">
            <input class="input"
                   id="edtFib"
                   type="number"
                   placeholder="N"
                   value="10">
          </div>
          <div class="control">
            <button id="btnStart"
```

```
                    class="button is-primary"
                    disabled>Calculate fibonacci</button>
          </div>
        </div>
      </div>
    </div>
  </div>
```

The main application class is changed so its definition is similar to the following:

```
TMyApplication = class(TBrowserWASIThreadedHostApplication)
Private
  BtnStart : TJSHTMLButtonElement;
  edtFib : TJSHTMLInputElement;
  procedure DoBeforeWasmInstantiate(Sender: TObject);
  function DoStartClick(aEvent: TJSMouseEvent): boolean;
  procedure DoWasmLoaded(Sender: TObject);
  procedure DoWrite(Sender: TObject; aOutput: String);
Public
  procedure doRun; override;
end;
```

The `DoRun` is where the ball starts rolling. It begins by gettig references to the button and edit in the HTML, and attaching a `click` handler to the button. After that it creates a large memory block descriptor (4096 pages of 64Kb) for the webassembly instance and sets some callbacks to be notified when the wasm module is about to be instantiated, and a callback to handling the `Writeln` output of our pascal program.

```
procedure TMyApplication.doRun;

var
  Mem : TJSWebAssemblyMemoryDescriptor;

begin
  Terminate;
  btnStart:=TJSHTMLButtonElement(GetHTMLElement('btnStart'));
  btnStart.onclick:=@DoStartClick;
  BtnStart.Disabled:=True;
  edtFib:=TJSHTMLInputElement(GetHTMLElement('edtFib'));
  Mem.initial:=4096;
  Mem.Maximum:=4096;
  Mem.Shared:=true;
  Host.MemoryDescriptor:=Mem;
  Host.OnConsoleWrite:=@DoWrite;
  Host.AfterInstantation:=@DoWasmLoaded;
  Host.BeforeInstantation:=@DoBeforeWasmInstantiate;
  Host.RunEntryFunction:='_initialize';
  Writeln('Host: Loading wasm...');
  StartWebAssembly('threaddemo.wasm',True);
end;
```

Finally we set the `RunEntryFunction` to `_initialize`, because we are loading a library and not a program. Finally we call `StartAssembly` to load the webassembly module and initialize it. The three callbacks are easy enough:

```
procedure TMyApplication.DoWrite(Sender: TObject; aOutput: String);
begin
  Writeln('Wasm: '+aOutput);
end;

procedure TMyApplication.DoBeforeWasmInstantiate(Sender: TObject);
begin
  Writeln('Host: Webassembly downloaded, instantiating VM');
end;

procedure TMyApplication.DoWasmLoaded(Sender: TObject);
begin
  Writeln('Host: wasm loaded, ready to run');
  BtnStart.Disabled:=False;
end;
```

Only when the webassembly is loaded, do we enable the button to start the calculation.

The `click` handler of the `btnStart` button calls the actual function:

```
function TMyApplication.DoStartClick(aEvent: TJSMouseEvent): boolean;

type
  TCalcProcedure = procedure(N : Integer);

var
  N : Integer;
begin
  Result:=false;
  N:=StrToInt(Trim(edtFib.Value));
  Writeln('Host: Calculating fibonacci(',N,')');
  TCalcProcedure(Host.Exported['CalcFibonacci'])(N);
end;
```

Note how the function is called: the `Exported` property of the `TWasiHost` class contains all exported functions. By typecasting them to a procedure with the correct signature, we can call the webassembly function as a normal function.

If all went well, running the program and hitting the calculate button results in something like figure 5 on page 12.

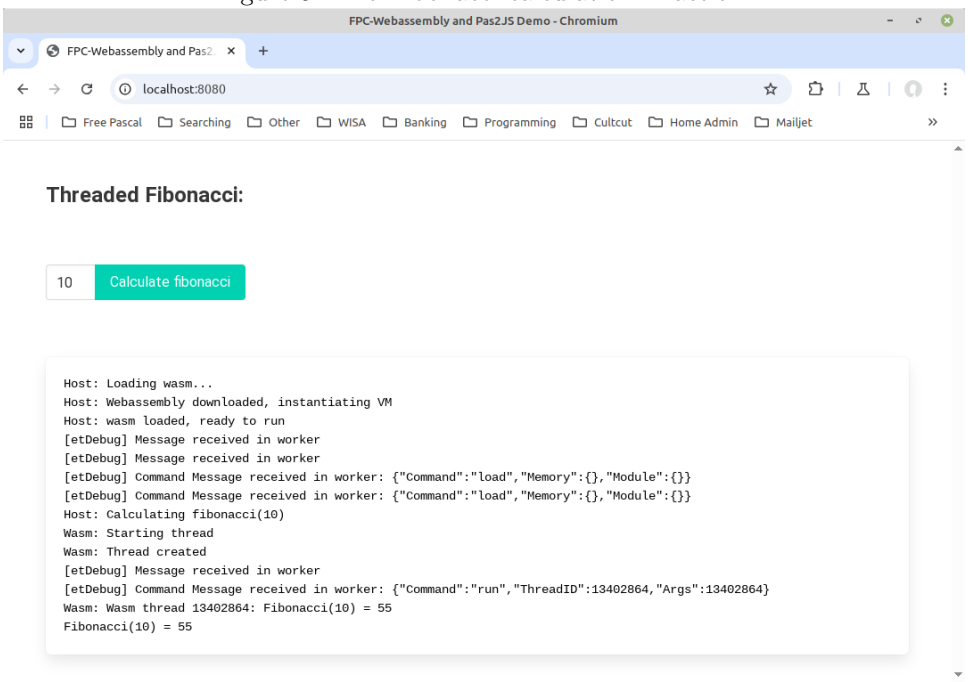
The `TBrowserWASIThreadedHostApplication` has a property called `ThreadSupport`. This property is an instance of the `TThreadSupport` class that is used to manage threads. It does all the work of loading workers, sending messages to start threads etc.

You can control it with the following properties:

WorkerScript This is a string value with the name of worker script that must be started to run threads in. The default value is `pas2jsthreadworker.js`, but if you create your own APIs you'll need to create another worker with the same APIs, and you can specify its name here.

InitialWorkerCount Initial number of thread runner web workers to start. It is set by the constructor of the `TThreadSupport` class.

Figure 5: The Fibonacci calculation in action.



MaxWorkerCount is the maximum number of thread runner web workers to start. The script will start additional threads up to this number of workers.

OnUnknownMessage an event handler that can be used to handle unknown messages that are received by the thread controller. You can use this to add and handle extra messages.

OnAllocateWorker an event handler that is called when a new thread worker is created.

You can send custom messages to all workers or to a single worker with the following methods:

SendCommandToAllWorkers can be used to send a command to all workers:

```
procedure SendCommandToAllWorkers(aCommand : TWorkerCommand);
```

SendCommandToThread can be used to send a command to a single worker:

```
procedure SendCommandToThread(aCommand : TWorkerCommand);
```

The `TWorkerCommand` class contains the ID of the thread to which the message must be sent.

The worker command class is simply a leightweight javascript object:

```
TWorkerCommand = Class external name 'Object' (TJSObject)
  Command : String;
  ThreadID : Integer; // Meaning depends on actual command.
  TargetID : Integer; // Forward to thread ID
```

```
end;
TCommandNotifyEvent = Procedure (Sender : TObject;
                                 aCommand : TWorkerCommand) of object;
```

The `Command` field has the actual command that needs to be executed. For some commands there can be a descendant of this class which contains the extra fields needed to execute the command. You can find several `TWorkerCommand` descendant implementations in the `Rtl.WebThreads` unit, together with the names of all the commands used by the `TThreadSupport` class.

5 Constraint: Accessing browser APIs

When you develop a Delphi or Lazarus application, it is well known that you should not access the UI layer from within a thread. Instead, a TThread.Synchronize must be used to execute code that needs to update the UI. Failing to do so will result in strange behaviour.

In the browser, something similar holds true: Because threads are emulated in the browser using web workers, and web workers have no access to the DOM, it is simply impossible for a thread to manipulate the UI directly. Any update must happen on the main page itself: the necessary objects and APIs simply do not exist in the web worker.

The constraint goes even further than that: since Javascript objects only exist in the worker in which they are created, they cannot be manipulated in another webworker.

For this reason, in pas2js the various JavaScript classes that make up the browser APIs are divided over 3 units:

weborworker this unit contains the Javascript classes that are available in web workers as well as in the main page. Examples are Blob, ImageBitmap Request and response of the Fetch API, and many more.

web contains aliases for most of the `weborworker` classes for backwards compatibility, and all the classes related to the DOM.

worker this unit contains APIs that are specific to web workers. This is usually the `DedicatedGlobalScope` class, the global scope of a worker.

This means that if you want to be sure that a webworker does not contain code that can access the DOM, you should only use the `weborworker` unit and `webworker` unit.

If a program needs to know in what environment it is executing, the `weborworker` contains the following functions:

```
type
  TJSScriptContext = (jscUnknown,jscMainBrowserThread,jscWebWorker,jscServiceWorker);

function isMainBrowserThread: boolean;
function isWebWorker : boolean;
function IsServiceWorker :boolean;
function GetScriptContext : TJSScriptContext;
```

They allow you to detect the environment and take corresponding action if need be.

Some javascript objects can be transferred to another webworker using `PostMessage`, but then they are no longer available in the original webworker.

So, what browser APIs can be used in a thread ? HTTP and websockets are available. The multimedia APIS (audio, video, drawing) are also available, just as the crypto APIs. Low-level Javascript such as promises, arrays etc. are of course available, they are part of the Javascript language and can be accessed if need be from a thread.

The use of webworkers also means that Javascript objects created in one thread are only available in that thread: If you create a HTTP request in a thread, you can only handle the response of that HTTP request in the same thread. A websocket created in one thread can only be used in that thread. Trying to write something to the websocket from another thread will fail.

All Objects created by the JOB framework (Javascript Object Bindings), presented in an earlier article, are also subject to this limitation: a JOB object can only be manipulated by the thread that created it: the object exists in the Javascript engine of the webworker that hosts the thread, but it does not exist in any other webworker.

And obviously, the JOB objects that reference any DOM-related classes (`HTMLELement`, `InputElement` and so on) will not work in a thread.

6 Constraint: No locking in the main page

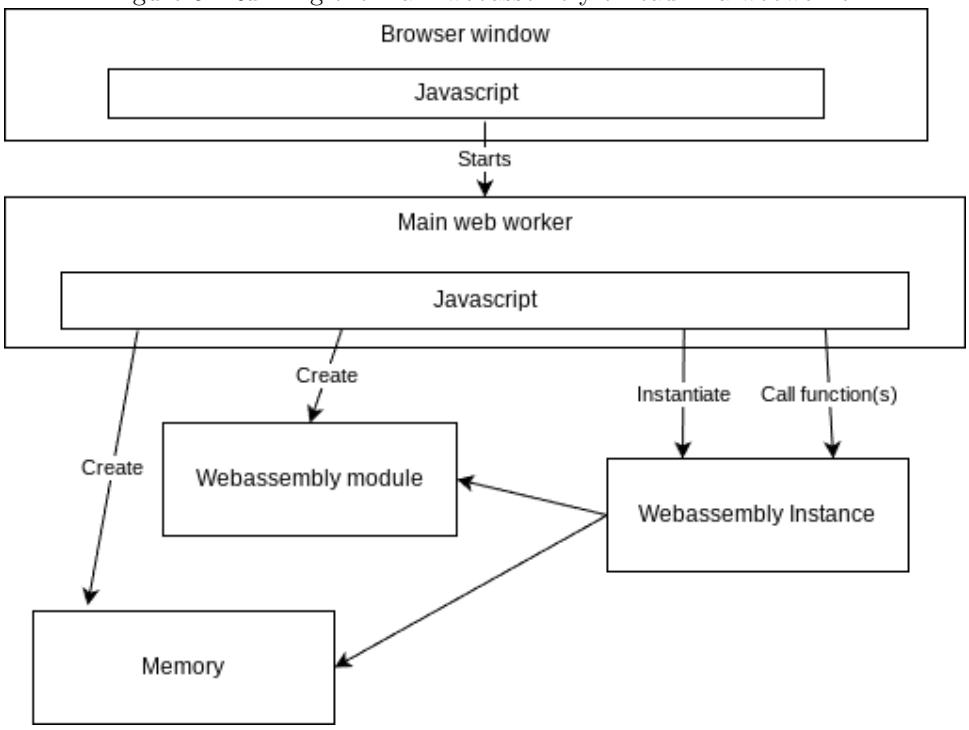
To ensure a smooth UI experience for the end-user, the browser prohibits blocking the main browser thread: you cannot perform any Javascript operations that would block the UI. If you write a javascript routine that takes too long, the browser will pop up a dialog to the user asking whether it should abort your code. Lengthy operations (HTTP requests, websocket handling, audio/video operations, encryption, database access and so on) are all performed in an asynchronous manner: These APIs use javascript Promises to ensure that their operation does not block the calling code. Blocking atomic operations such as `Atomics.wait` are not allowed in the main thread, and are replaced with asynchronous operations such as `Atomics.waitAsync`.

This has some consequences for threaded WebAssembly code: locking primitives (`TCriticalSection`, `TMonitor`) will not work in the main browser thread. If your code uses these primitives, you'll have no choice but to run it in a webworker.

The `TThread` class, and in particular `TThread.Synchronize` and `TThread.Queue` - and their `CheckSynchronize` pendant in the main thread - use locking primitives to do their work. The above means they cannot do their work if used in the main browser page. If your code uses threads, then indeed your program must run in a webworker.

For this reason, Pas2JS has the `TWorkerThreadControllerApplication` class in unit `wasiworkerthreadhost`: This is an application class which acts as the main thread for a webassembly program, but which runs in a webworker, instead of in the main browser window. At this point, the Lazarus 'Web Browser Application' wizard does not yet allow you to create such a project, so, the best start is to create a 'Node.js' project and adapt that to use a `TWorkerThreadControllerApplication` class. The reason for creating a 'Node.js' application is that it calls the `rtl.run()` function, which runs your pascal code: in a browser application, the `rtl.run()` function is called by the html page itself.

Figure 6: Running the main webassembly thread in a webworker



In that, starting a threaded webassembly program involves the following steps:

1. The main HTML page program starts a webworker.
2. The webworker uses a `TWorkerThreadControllerApplication`.
3. The `TWorkerThreadControllerApplication` class loads the webassembly module and instantiates it.
4. The threading code in Pas2js will take care of creating more web workers to run various threads.

The situation is depicted in figure 6 on page 15.

7 GUI programs and threading

Since we know that the DOM is not available in a webworker, does this mean that you cannot create a webassembly GUI application if it uses threads ? At first sight, it would seem so.

Luckily, the limitations can be worked around:

- It is possible to forward UI events from the main HTML page to a worker using `PostMessage`.
- Similarly, it is possible to send instructions from a worker to the main page. These instructions could include HTML snippets to insert at certain locations.

- Drawing operations on a canvas can be done on an `OffscreenCanvas` : in essence, the drawing operations are performed on a bitmap. This bitmap can then be sent to the main page, and drawn on an actual canvas. Sending the bitmap is a cheap operation, as only a handle to the bitmap needs to be transferred.

These workarounds need quite some setup. Lots of different messages are passed between the main page to the webworker with the thread controller.

The web assembly version of the Fresnel project can work in both cases: the webassembly can be executed directly in the main page if it is not threaded, or it can be executed in a web worker and will handle all the messages to draw the UI on a HTML canvas. This is completely transparant to the webassembly code: all the details are taken care of by the pas2js code that handles the Javascript side of the Fresnel webassembly framework.

As reported in an earlier article, Delphi's FMX framework can be compiled in Free Pascal and run in the browser using a webassembly program. Since it is based on Fresnel, it will also work in a web worker if you use a thread-enabled version of webassembly.

The following simple camera application (written in FMX) demonstrates this:

`https://www.freepascal.org/~michael/wasm-demos/webcamdemo/`

Pressing the 'EnumMedia' button will enumerate all media devices accessible by your browser in the memo at the left. Once done, you can select a camera using the dropdown and see the capture camera displayed: the camera's image is transferred to the webassembly, who then draws the image in a paintbox. The whole FMX application's form is drawn on a HTML canvas.

The result can be seen in figure 7 on page 17. If you look at the browser javascript (and webassembly) debugger in figure 8 on page 17, you'll see 4 "threads" :

- The main page thread, which starts the `wasm_worker_loader` worker. After that it just forwards mouse and keyboard messages to the `wasm_worker_loader` thread and responds to drawing messages from that thread. It also forwards the video camera images to the webassembly.
- The `wasm_worker_loader` thread, which runs the webassembly module's main thread.
- Two `wasm_worker_runner` threads. They allow to run a thread started by the webassembly program.

The program's sources will be included in an upcoming update of the patches needed to run a FMX program in the browser.

8 Conclusion

Despite the limitations imposed by the browser, running threaded webassembly programs is definitely possible. This means that existing FPC (and Delphi) applications can be ported to the browser. As shown here, doing so takes some work to set up an executing environment on the Javascript side, but it is expected that this amount of work will be reduced by adding some more wizards to the Lazarus IDE. But for every application there may always be some extra work to include custom

Figure 7: FMX program with the main webassembly thread in a webworker

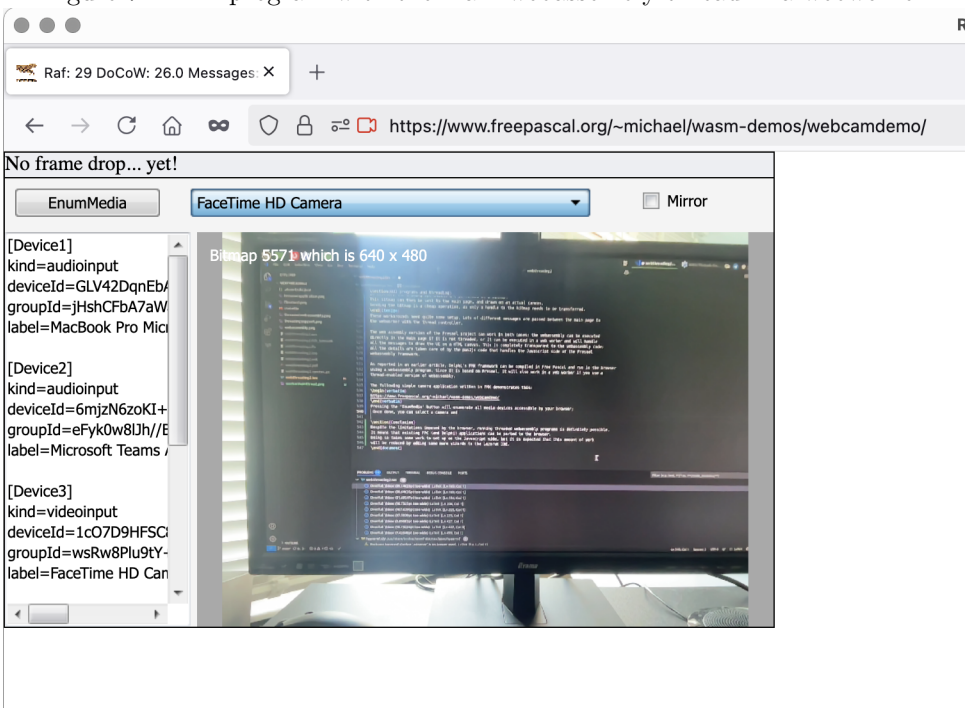
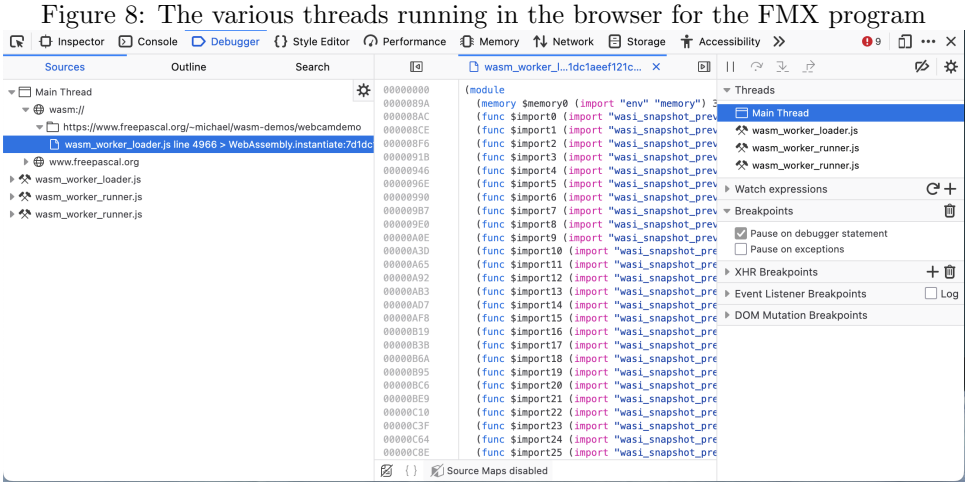


Figure 8: The various threads running in the browser for the FMX program



APIs into the hosting environment: The combination of FPC and pas2js allows you to use HTTP and websockets out of the box, but if you want to use other Javascript APIs you may need to make them availabe to the webassembly module yourself. How to do this will be reported in a next article.