Nikoloudakis Michael
michnik@csd.uoc.gr

Porting Maxine VM to MS Windows

Introduction

The goal of this project was the addition of Windows support to the Maxine VM without the use of any virtual environment or additional DLL dependencies (eg. Cygwin). Windows constitutes one of the most popular OSes for Java developers (regarding both Java VMs and Java aps in general) so, in my opinion, Maxine VM should support it along with the rest OSes. Additionally, Maxine VM, being a meta-circular Java VM, has the biggest part of it written in Java while a (relatively) small part of the VM is implemented in native C. Currently, Maxine VM fully supports Linux, Darwin and Solaris OS. As it is expected, native C code consists of many OS specific libraries and function calls ,thus, the main challenge was the porting of the native code making it eligible to run on a Windows environment, making use of Windows native shared libraries and functions. Some changes were also needed in some parts of Java code as well as in some Makefiles. At the current state, all the native source coude is sucessfulyy ported to Windows except code regarding Tele.dll, the library used in order to make the Maxine Inspector functional. The purpose of this report is to mention all the steps one needs to follow in order to reproduce the porting process on his local Windows machine, describe ,in some detail, the changes that were made and also state the problems and difficulties met in the process.

Initial Setting Up

1. First of all, we need a version of Git for Windows in order to be able to pull the Maxine-VM and mx repositories. It's advisable to **not** use the app of a specific Git platfrom (eg. Github, Gitlab etc) since it is not certain that they contain a git.exe executable which can be used from any command line terminal. You can download Git for Windows from https://git-scm.com/download/win If you choose the portable version, make sure to add the path to the bin folder it includes to the **PATH** Windows system Environment Variable. This will enable "git" command to work from Windows Command line as well. If you choose to install Git normally, make sure to check the respective option (to modify **PATH** system environment variable) during the installation (might be the default option).

2. Afterwards, we can use either the terminal included with Git or the common Windows Command Line (cmd.exe) or any other terminal you wish to clone the required repositories using:

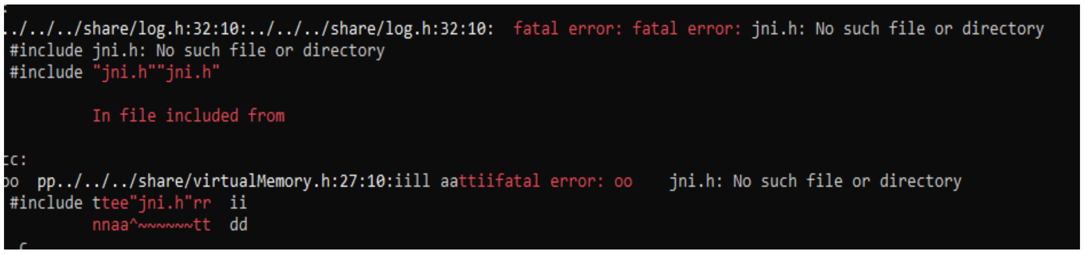
   git clone https://github.com/graalvm/mx
   git clone --recursive https://github.com/beehive-lab/Maxine-VM.git

   Instead of the official repository in the second command above, you can clone https://github.com/mihalis341/Maxine-VM which already contains all the changes made for Windows

Also, don't forget to add the cloned **mx** folder to the **PATH** system environment variable as well.

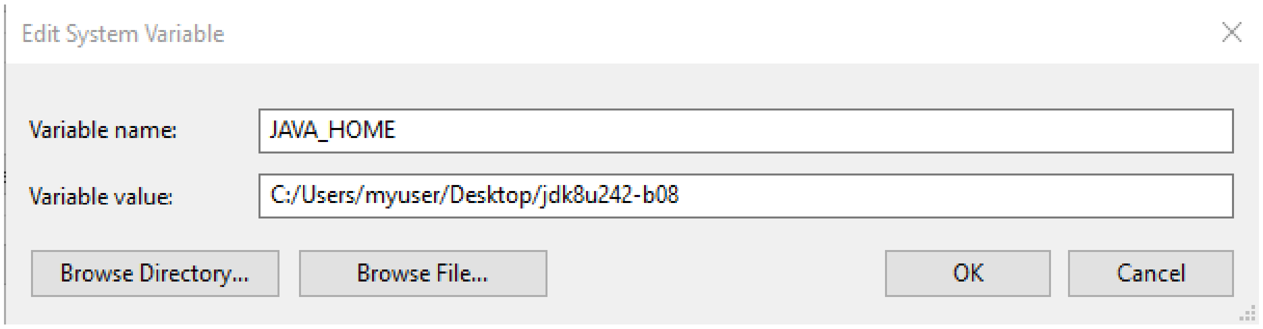
3. We need a supported version of OpenJDK 8 for Windows. You can download any **JDK** version >=2.2.2 from https://adoptopenjdk.net/archive.html?variant=openjdk8&jvmVariant=hotspot . It is recommended to choose Binary (which downloads a .tar.gz file) and not Installer in the above page. After the download, you must extract the tar.gz file (it contains just the JDK folder) to any location you want and also create a new Windows Environment Variable called **JAVA_HOME** which should point to the JDK folder.

   Caution: Make sure that the path you choose for the JDK folder does not contain any spaces. All the mx scripts make use of GNU executables (eg. make) which don't like spaces at all. Also, make sure that the path of the JAVA_HOME environment variable you created uses **Linux** path separator ('/') **instead of Windows** separator ('\'). Windows allows Linux separators too so there is no problem with that practice. If one used Windows separator, the makefiles are able to locate the necessary JDK files, however, GCC for Windows faces some problems with it as it handles Windows separators as escape characters resulting in peculiar errors like:



```
./../../share/log.h:32:10:../../../share/log.h:32:10:  fatal error: fatal error: jni.h: No such file or directory
#include jni.h: No such file or directory
#include "jni.h""jni.h"

         In file included from

c:
o  pp../../../share/virtualMemory.h:27:10:iill aattiifatal error: oo    jni.h: No such file or directory
#include ttee"jni.h"rr  ii
         nnaa^~~~~~~tt  dd
```

   Another Windows compiler may not have such a problem. Below, is an example of the path that must be used for JAVA_HOME environment variable



Edit System Variable

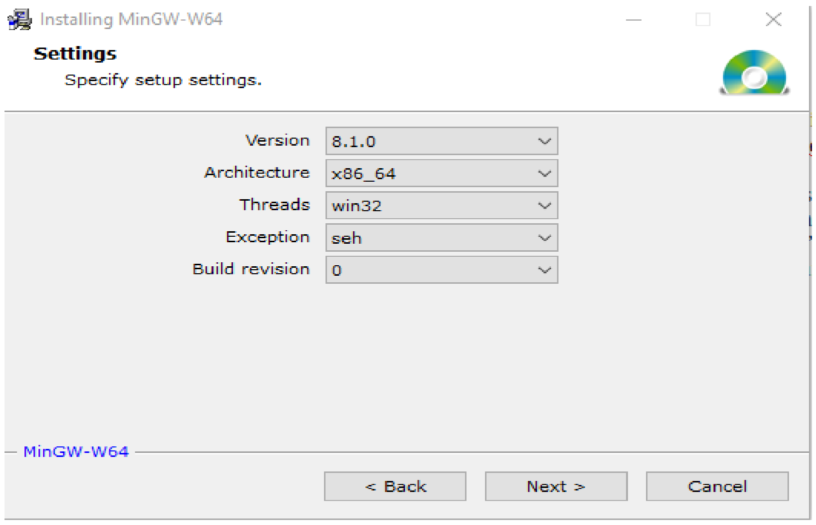
Variable name: JAVA_HOME

Variable value: C:/Users/myuser/Desktop/jdk8u242-b08

JAVA_HOME path containing no spaces and using Linux seperators

4. We also need a collection of Windows ports of common Linux commands (cp, mkdir, uname, make etc) used by mx scripts and Makefiles available for download here . Those exes rely on native Windows DLLs as well and they do not require the use of any virtual environment or extra dependency. We only need one folder from UnixUtils.zip ,we downloaded, called “wbin” which we must extract anywhere we wish and add its path to the **PATH** system environment variable as well. There are Windows commands equivalent to cp and mkdir we could use but the makefiles use “uname” command to determine the OS and ISA we are using and there is no other way other than “uname” to see if we are running on Windows from within a Makefile. Additionally, we need the make.exe executable from UnixUtils for which we have found no alternative until now. (**Update:** On a third PC with Windows 10, make.exe produces **The syntax of the command is incorrect** error on Windows Command Line for unknown reason but it works correctly with Git’s Bash instead so it is advisable to prefer it over cmd. This error does **not** appear on Windows Command Line of two other machines the port was tested.)

5. We also need a version of GCC for Windows. One can download it from here . The exact same GCC compiler may come with some Windows IDEs like CodeBlocks. During the installation make sure to choose correctly your ISA. In the “Threads” option choose “Win32” and no “posix” since we make no use of POSIX threads in our work. (Even if one chooses POSIX everything will work fine as well but you download an extra useless library)



Version: 8.1.0
Architecture: x86_64
Threads: win32
Exception: seh
Build revision: 0

After the installation is over, we must also add the folder that contains **gcc.exe** (typically something like this on 64bit windows: **C:\Program Files\mingw-w64\x86_64-8.1.0-win32-seh-rt_v6-rev0\mingw64\bin\** once again to the **PATH** environment variable.

Note: One can use any compiler they want. However, the makefiles are configured for GCC. Should anyone need to use a different C compiler, they must also edit the respective Makefiles

6. Python should be downloaded and installed from the official website. All python scripts used are Python 2.7 scripts so that Python version should be enough. However, we have noticed that at some cases Python 2.* cannot handle paths bigger than **MAX_PATH_LENGTH** defined on Windows. As a result, some maxine classes with a very long name failed to get found and an error mentioning that the file did not exist was arising. Python 3 offers the option to abolish that Windows limit during its installation so if ones has such a problem, they should use Python 3

instead. This Windows limit was supposed to disappear in a specific Windows 10 version, however, at one of the two machines that I tested the port on, the limit did continue to exist at the past. Today, though, I am unable to reproduce the error with Python 2.7 so maybe a Windows update has fixed it. In any case, should one face such an error and need to use Python 3 they should also use 2to3.py script included in Python 3 tools folder in order to convert mx.py to Python 3. One should also ensure that Python is added in **PATH** environment variable as well. (The installation should take care of it)

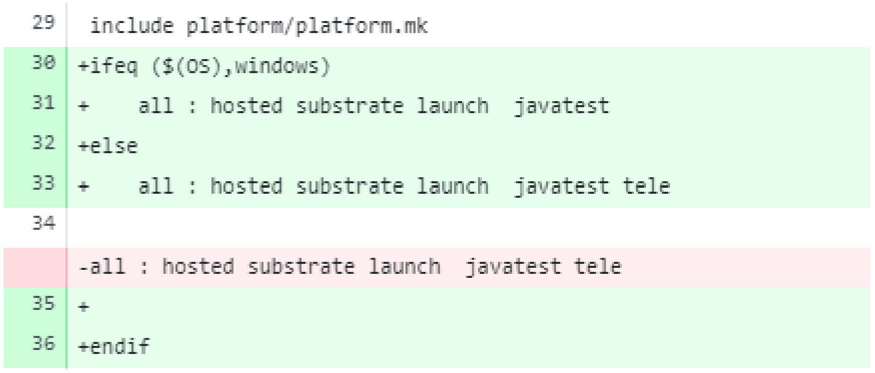
7. Finally, we can compile Maxine VM and build a boot image from Windows Command Prompt (cmd.exe) or any other terminal of your choice by executing

   mx build
   mx image

Summary of changes regarding Makefiles

First of all, there was no need for any change at any Python script (excluding the optional transformation of Python 2.7 to Python 3.8.2 described in step 6 of the previous section).

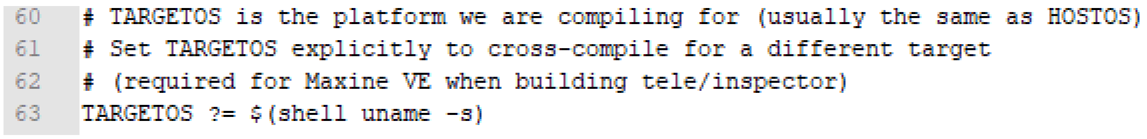
The central makefile (com.oracle.max.vm.native/makefile) which constitutes the starting point of the whole native building process needed to get changed in order to exclude Tele from the building process as it was not implemented as stated in the introduction. The reason for this is that Tele source files make extensive use of the ptrace syscall that has no equivalent for Windows. Also, complete UNIX emulating virtual environments like Cygwin do not support it. It’s also worth mentioning that the Maxine Inspector itself is an obsolete and abandoned part of Maxine VM receiving no active development. In order to make the makefile runnable on windows we should make the following change:



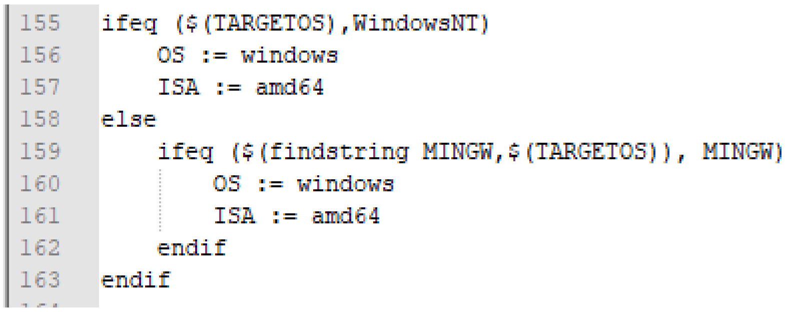
```
 include platform/platform.mk
+ifeq ($(OS),windows)
+    all : hosted substrate launch  javatest
+else
+    all : hosted substrate launch  javatest tele

-all : hosted substrate launch  javatest tele
+
+endif
```

The value of $OS is checked and if it equals windows, tele is not included in “all” target.

The variable OS is determined in platform.mk located at com.oracle.max.vm.native/platform.mk at the following lines:

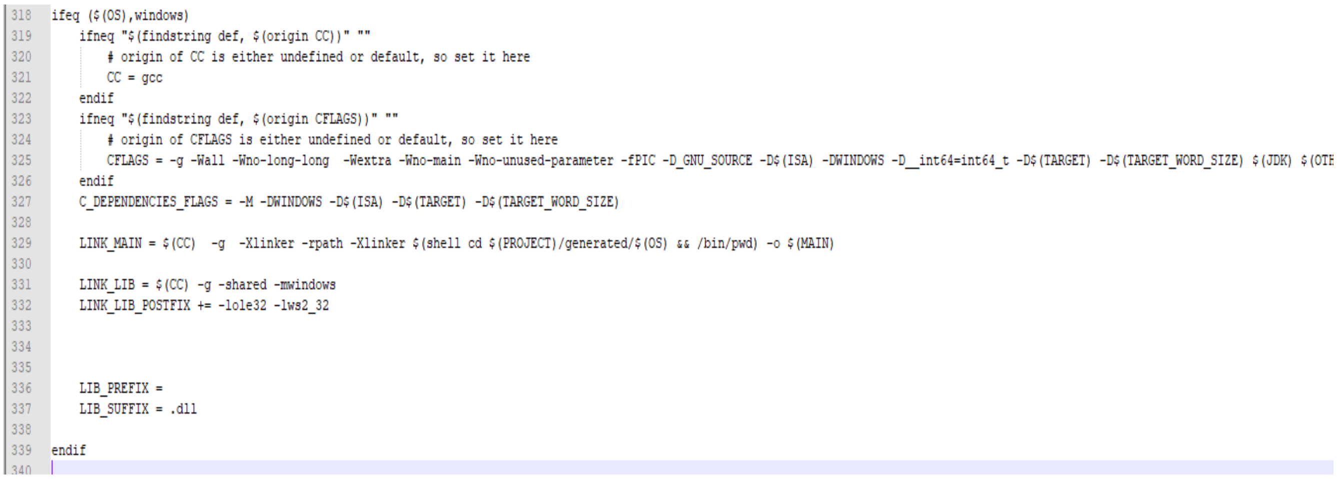


```
# TARGETOS is the platform we are compiling for (usually the same as HOSTOS)
# Set TARGETOS explicitly to cross-compile for a different target
# (required for Maxine VE when building tele/inspector)
TARGETOS ?= $(shell uname -s)
```



```
ifeq ($(TARGETOS),WindowsNT)
    OS := windows
    ISA := amd64
else
    ifeq ($(findstring MINGW,$(TARGETOS)), MINGW)
        OS := windows
        ISA := amd64
    endif
endif
```

Initially, the platform.mk makefile invokes umame -s command which returns a string containing the name of the OS. If that value equals WindowsNT" or xontains "MINGW" (the provider of GCC for windows) then the OS is determined as "windows" and ISA as "amd64" as it is the only architecture we tested our port on". Afterwards the value of OS is once again used in platform.mk in order to define specific compile and link flags as well as suffixes (eg."dll") for Windows shared libraries.

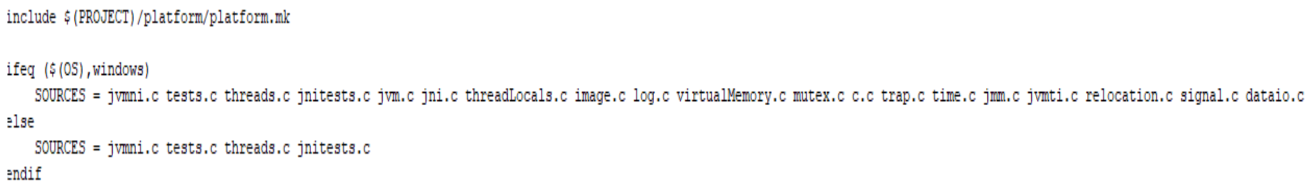


```
318 ifeq ($(OS),windows)
319     ifneq "$(findstring def, $(origin CC))" ""
320         # origin of CC is either undefined or default, so set it here
321         CC = gcc
322     endif
323     ifneq "$(findstring def, $(origin CFLAGS))" ""
324         # origin of CFLAGS is either undefined or default, so set it here
325         CFLAGS = -g -Wall -Wno-long-long  -Wextra -Wno-main -Wno-unused-parameter -fPIC -D_GNU_SOURCE -D$(ISA) -DWINDOWS -D__int64=int64_t -D$(TARGET) -D$(TARGET_WORD_SIZE) $(JDK) $(OT
326     endif
327     C_DEPENDENCIES_FLAGS = -M -DWINDOWS -D$(ISA) -D$(TARGET) -D$(TARGET_WORD_SIZE)
328 
329     LINK_MAIN = $(CC)  -g  -Xlinker -rpath -Xlinker $(shell cd $(PROJECT)/generated/$(OS) && /bin/pwd) -o $(MAIN)
330 
331     LINK_LIB = $(CC) -g -shared -mwindows
332     LINK_LIB_POSTFIX += -lole32 -lws2_32
333 
334 
335 
336     LIB_PREFIX =
337     LIB_SUFFIX = .dll
338 
339 endif
340 
```

As seen in the above image, at Line 325 the compile flags are defined. In general, those flags resemble the ones used in the Linux case, however, we need to include -DWINDOWS. Defining this macro helps the compiler determine the OS we are using and decide which segments of code to include in the compilation process. (There is much code in the source files which is inside #ifdef sentences). In addition, we need the flag -D__int64 = int64_t because GCC is not aware of the daatype __int64 used in some JDK's header files. Thus, we replace it with the equivalent int64_t. At line 332 one can also notice two libraries (lole32 and lws2_32) included in the link flags that are put at the end of the linking command (LINK_LIB_POSTFIX). Those two libraries are native in Windows and we mus link them at runtime if we want to use functions like CoTaskMemFree or socket functions that are described further in the report.

Also, at line 337 we define the suffix a shared library must have on Windows which is .dll
Furthermore, a small change was needed at com.oracle.max.vm.native/launch.mk which regards the executable maxvm which is the basic launcher of MaxineVM. Specifically, the suffix ".exe" has been added to target "MAIN" which gets automatically added by GCC in the executable's name and then "cp" command must be able to detect it.

Finally, com.oracle.max.vm.native/javatest.mk is a Makefile responsible for the building of javatest.dll which, as the name suggests, contains some tests for the JVM. Inside, the original javatest.mk are used sources which invoke functions that are implemented in other sources not included in javatest.mk makefile. Generally, this is allowed on Unix OSes where you can have as many undefined references (ie. function calls without implementation) as you wish inside a shared library and hope that another library loaded at runtime will provide them. However, on Windows DLLs needsto have all external dependencies resolved during linking and no undefined references are allowed. That's why we added more source files in that makefile if the detected OS is Windows. In that way, the DLL contains implementations for any function called within it. The rest of shared libraries of Maxine include by default all sources that are needed to ensure the absence of unresolved symbols so no change was needed.



```
include $(PROJECT)/platform/platform.mk

ifeq ($(OS),windows)
    SOURCES = jvmni.c tests.c threads.c jnitests.c jvm.c jni.c threadLocals.c image.c log.c virtualMemory.c mutex.c c.c trap.c time.c jmm.c jvmti.c relocation.c signal.c dataio.c
else
    SOURCES = jvmni.c tests.c threads.c jnitests.c
endif
```

Summary of some of the changes that got applied in native Code

The native code of Maxine VM can be segmented to 5 parts by taking into account the final shared libraries or executable it produces. More specifically, the final products of the native code are

hosted.dll

A library containing functions that reveal characteristics of the host system (eg. endianess, page size, path to jni.h etc). It is used by Boot Image Generator in order to produce a working boot image for a specific target.

javatest.dll

A library containing some Java tests in order to test the integrity of Maxine VM (eg. Print “Hello World” or return and print the name of a Java Class in C code etc). All of those functions are exported using JNI and can be called by a Java application that loads that DLL.

jvm.dll

The DLL constituting the implementation of the so called substrate. It makes extensive use of POSIX methods (eg.threads, mutexes, semaphores(not available on Windows), memory mapping methods, Signal handling, socket reading, shared Library loading etc. It is the actually the tool that JVM uses to interact with the host system (eg for allocating heap, loading the boot image file etc). It is also the “bootloader” of the whole VM itself.

maxvm.exe

It is the only executable that emerges from the building process and its main purpose is to load jvm.dll (substrate) and run its main function (maxine())

tele.dll   **UNIMPLEMENTED**

It is the main part (non- visual) part of the Maxine Inspector. It can debug Java programs, provide info about running threads (eg. how much memory is allocated, the status of its registers etc.) Its code is highly ISA specific and it relies almost exclusively to the UNIX **ptrace**() function whose equivalent most likely does not exist for Windows.

Below there is a table showing the source files needed for each executable/ shared library (**except** for tele.dll)

hosted.dll	c.c log.c platform.c relocation.c dataio.c mutex.c
javatest.dll	jvmni.c tests.c threads.c jnitests.c jvm.c jni.c threadLocals.c image.c log.c virtualMemory.c mutex.c c.c trap.c time.c jmm.c jvmti.c relocation.c signal.c dataio.c
jvm.dll	c.c condition.c log.c image.c $(ISA).c jni.c jvm.c maxine.c memory.c mutex.c relocation.c dataio.c snippet.c threads.c threadLocals.c time.c trap.c virtualMemory.c jnitests.c sync.c signal.c jmm.c jvmti.c barrier.c
maxvm.exe	maxvm.c

It is apparent that some source files are present in more than one product of the building process (eg. relocation.c , c.c, mutex.c etc) those belong to a folder called “**share**” and we will refer to them just once (the first time they appear) while describing the changes that took place durig the porting process. If a source file is not mentioned it means that it did not include any OS specific code thus it can be used without changes on any OS or that the changes it includes are already described in other files or are not that important. So the choice of which changes will be included in the report is somewhat subjective. The full repository with all the changes can be found here https://github.com/mihalis341/Maxine-VM/

Changes regarding the creation of hosted.dll

platform.c

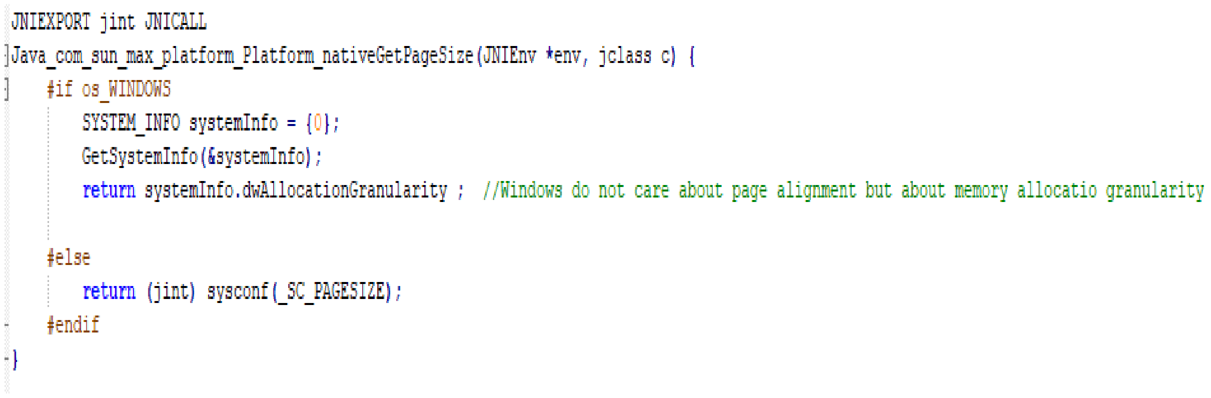
This source file provides functions that reveal information about the Host OS. For example Java_com_sun_max_platform_Platform_nativeGetOS(JNIEnv *env, jclass c);

or

Java_com_sun_max_platform_Platform_nativeGetPageSize(JNIEnv *env, jclass c); etc.

From those functions, the only OS specific one is that regarding the memory page size. In UNIX s ytems the system call `sysconf(_SC_PAGESIZE);` is available. On the other hand, on Windows, we have to use a struct of type SYSTEM_INFO which includes many field describing the page size, number of processors etc. We fill that struct using `GetSystemInfo` win32 call. The obvious field describing page size is `DWORD dwPageSize;`. However, Maxine VM needs this number in order to allocate addresses nd offsets that are multiples of it. (many functions such as mmap accept offsets and addresses that are multiples of page size) This works for UNIX,however, on Windows all functions that require aligned addresses or offsets (eg. MapViewOfFileEx) as arguments want this alignment to be done **not** in accordance to the page size but according to `AllocationGranularity`. That's why we return `DWORD dwAllocationGranularity;` instead of page size

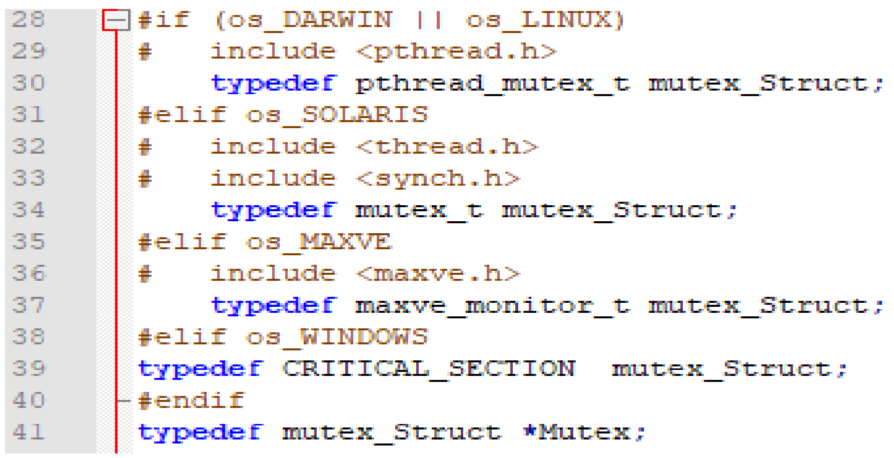
(why things are different on windows: https://devblogs.microsoft.com/oldnewthing/20031008-00/?p=42223)



```
JNIEXPORT jint JNICALL
Java_com_sun_max_platform_Platform_nativeGetPageSize(JNIEnv *env, jclass c) {
    #if os_WINDOWS
        SYSTEM_INFO systemInfo = {0};
        GetSystemInfo(&systemInfo);
        return systemInfo.dwAllocationGranularity ;  //Windows do not care about page alignment but about memory allocatio granularity

    #else
        return (jint) sysconf(_SC_PAGESIZE);
    #endif
}
```

mutex.c

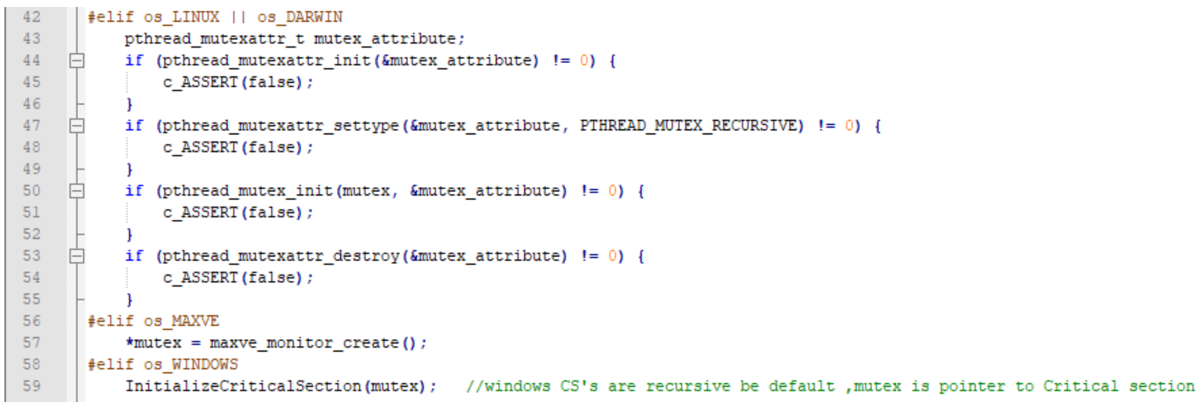
On Windows, there are objects that are also called Mutexes. However, for various reasons, it turns out that the real equivalent of POSIX mutexes are win32 Critical Sections .Windows Mutexes are shared between processes, not threads. That’s why they are quite different from POSIX ones. Also, win32 Critical Sections can be used as arguments to Condition Variables just like POSIX mutexes are used as arguments in POSIX conditions. Initially, we should define the following in mutex.h



```
28  #if (os_DARWIN || os_LINUX)
29  #    include <pthread.h>
30       typedef pthread_mutex_t mutex_Struct;
31  #elif os_SOLARIS
32  #    include <thread.h>
33  #    include <synch.h>
34       typedef mutex_t mutex_Struct;
35  #elif os_MAXVE
36  #    include <maxve.h>
37       typedef maxve_monitor_t mutex_Struct;
38  #elif os_WINDOWS
39  typedef CRITICAL_SECTION  mutex_Struct;
40  #endif
41  typedef mutex_Struct *Mutex;
```

At line 39 we define mutex_Struct as CRITICAL_SECTION while at line 41 Mutex is defined as a pointer to mutex_Struct (the latter happens for all OSes). In mutex.c
we make this change regarding function

void mutex_initialize(Mutex mutex);



```
#elif os_LINUX || os_DARWIN
    pthread_mutexattr_t mutex_attribute;
    if (pthread_mutexattr_init(&mutex_attribute) != 0) {
        c_ASSERT(false);
    }
    if (pthread_mutexattr_settype(&mutex_attribute, PTHREAD_MUTEX_RECURSIVE) != 0) {
        c_ASSERT(false);
    }
    if (pthread_mutex_init(mutex, &mutex_attribute) != 0) {
        c_ASSERT(false);
    }
    if (pthread_mutexattr_destroy(&mutex_attribute) != 0) {
        c_ASSERT(false);
    }
#elif os_MAXVE
    *mutex = maxve_monitor_create();
#elif os_WINDOWS
    InitializeCriticalSection(mutex);   //windows CS's are recursive be default ,mutex is pointer to Critical section
```

We see that in Linux, it is needed to create a mutex_atttribute object first in order to specify that the mutex will be recrusive (ie. the thread that has locked can relock it without udefined behavior). On windows, critical sections are by default recursive so a call to InitializeCriticalSection is enough.

Similarly, regarding the next function **int mutex_enter_nolog(Mutex mutex)** the Linux statement **return pthread_mutex_lock(mutex);** replaced by

EnterCriticalSection(mutex);
return 0;

on Windows. (**EnterCriticalSection** returns nothing on Windows, so we return 0 which means success in Linux and is what ou program would expect.

Similarly, in function **int mutex_try_enter(Mutex mutex)** the Linux statement
return pthread_mutex_trylock(mutex); gets replaced by
return !TryEnterCriticalSection(mutex); Both of these calls try to lock mutex/cs without blocking. On Windows we use '!' before returning the result because Windows return non-zero on success while the Linux equivalent returns 0 so out program expects 0 for success (since it was built for Unix orginally).

With the same thinking, the rest of changes were applied in the rest of the mutex functions (Linux left, Windows right)

→

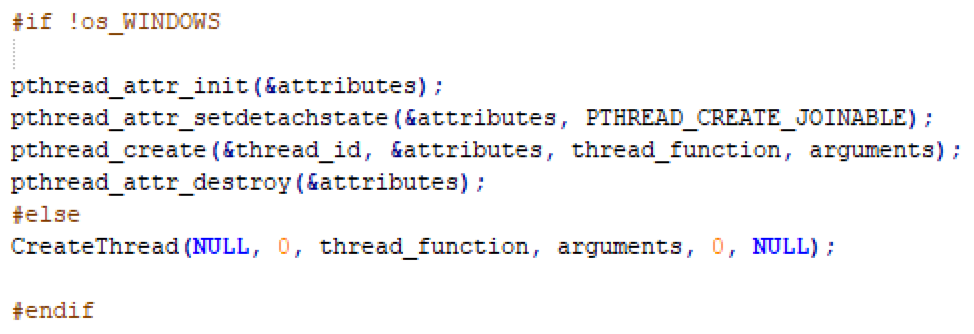
return pthread_mutex_unlock(mutex);	**LeaveCriticalSection(mutex);**
	
	return 0; //because leavecriticalsection
	
	does not return anything

pthread_mutex_destroy(mutex);	**DeleteCriticalSection(mutex);**

Changes regarding the creation of jvm.dll and javatest.dll (they have many sources in common on Windows)

tests.c

Inside this source file the only OS specifc action taking place is the creation of a Thread.



```
#if !os_WINDOWS

pthread_attr_init(&attributes);
pthread_attr_setdetachstate(&attributes, PTHREAD_CREATE_JOINABLE);
pthread_create(&thread_id, &attributes, thread_function, arguments);
pthread_attr_destroy(&attributes);
#else
CreateThread(NULL, 0, thread_function, arguments, 0, NULL);

#endif
```

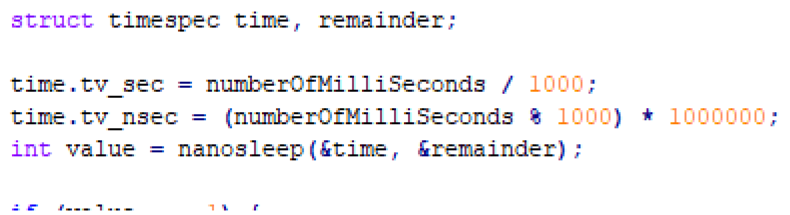
On Linux, the developers first create an attributes variable (creation not shown in image) and set the thread's detached state as JOINABLE (ie. main() can wait for it using pthread_join). The strange part is that this process is also redundant on Linux since threads are by default joinable so the setting of attributes just for that has no meaning. After that the thread is created using **pthread_create** on Linux and **CreateThread** on Windows. Windows Threads are also by default Joinable. CreateThread returns a HANDLE (ie void *) to the thread however, we do not need to return it. Also, on Linux, thread_id is created inside the function and never returned. The signature of thread_functions is also changed on

Windows form **void * thread_function(void * args)** to **DWORD thread_function(void * args)** in order to avoid compiler warnings since this is the required signature for Windows (DWORD is int)

threads.c

Apart from the previous changes described in the previous file regarding thread creation, this file provides also some other implementations like **JNIEXPORT void JNICALL Java_com_sun_max_vm_thread_VmThread_nativeYield(JNIEnv *env, jclass c)** which makes the current thread yield. So the Linux **pthread_yield();** is replaced by the Windows **SwitchToThread();** which yields the CPU if there is another thread asking for CPU time.

Additionally, an interesting function contained in that file is also
jboolean thread_sleep(jlong numberOfMilliSeconds)
which on Linux calls the following

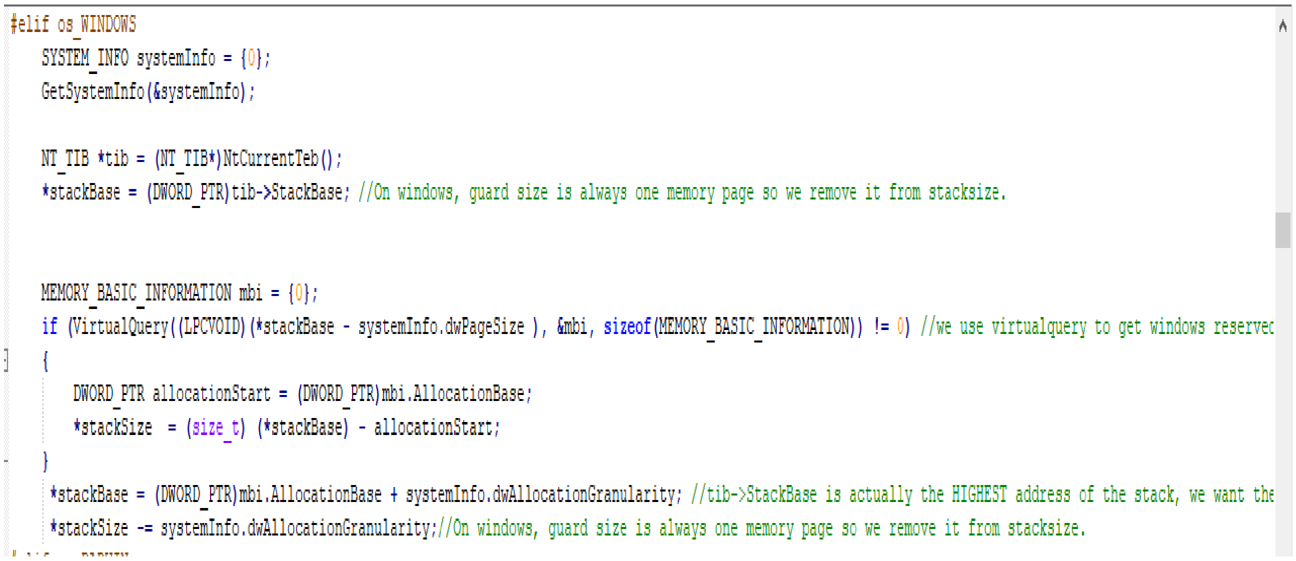


```
struct timespec time, remainder;

time.tv_sec = numberOfMilliSeconds / 1000;
time.tv_nsec = (numberOfMilliSeconds % 1000) * 1000000;
int value = nanosleep(&time, &remainder);
```

While on Windows it is simply implemented as **Sleep(numberOfMilliSeconds);** since win32 Sleep() accept milliseconds as argument.

A more interesting function is **void thread_getStackInfo(Address *stackBase, Size* stackSize)** which return the base of the stack (ie. lowest address) and the stack size available for use be the VM while on Linux there are **pthread_attr_getstack(&attr, (void**) stackBase, (size_t *) stackSize)** which can directly return the requested values.

Unfortunately, on Windows there is no direct way to find either of these.



```
#elif os_WINDOWS
    SYSTEM_INFO systemInfo = {0};
    GetSystemInfo(&systemInfo);

    NT_TIB *tib = (NT_TIB*)NtCurrentTeb();
    *stackBase = (DWORD_PTR)tib->StackBase; //On windows, guard size is always one memory page so we remove it from stacksize.


    MEMORY_BASIC_INFORMATION mbi = {0};
    if (VirtualQuery((LPCVOID)(*stackBase - systemInfo.dwPageSize ), &mbi, sizeof(MEMORY_BASIC_INFORMATION)) != 0) //we use virtualquery to get windows reservec
    {
        DWORD_PTR allocationStart = (DWORD_PTR)mbi.AllocationBase;
        *stackSize  = (size_t) (*stackBase) - allocationStart;
    }
     *stackBase = (DWORD_PTR)mbi.AllocationBase + systemInfo.dwAllocationGranularity; //tib->StackBase is actually the HIGHEST address of the stack, we want the
     *stackSize -= systemInfo.dwAllocationGranularity;//On windows, guard size is always one memory page so we remove it from stacksize.
```

First, we use **NtCurrentTeb();** call which returns info about the current thread. Inside, the returned struct there is a field called **StackBase** **BUT** unfortunately, windows define as StackBase the **highest** address of the stack instead of the lowest. This part caused many errors and bugs that got fixed eventually. In order to find the real StackBase (ie the lowest stack address) we can call virtual query anywhere inside the stack. This takes as argument MEMORY_BASIC_INFORMATION struct and upon return it contains a field called allocation base. This is the real stack base. So stacksize = StackBase – allocation base. Since the last page of the stack is always a guard page, we want to move the pointer to stackbase a bit upwards so we add dwAllocationGranularity value and we substract it from stack_size. We could have used dwPageSize instead but as explained earlier Windows want addresses to be aligned according to dwAllocationGranularity and not page size so we sacrifice some Kbs of available memory. (The returned stackbase is later used bu windows functions that require such kind of allocation) (page size is tpyically 4kb while allocation granulrity is 64kb)

jvm.c

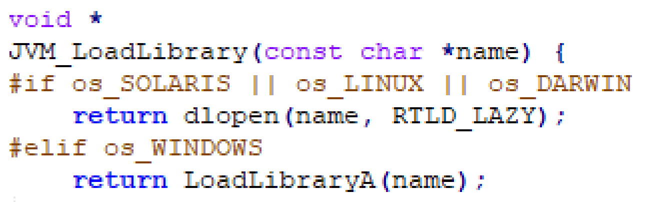
In this file there are native functions supposed to be used by Java program. Those functions concern many different things. For example,

jint JVM_ActiveProcessorCount(void) which returns the number of processors.

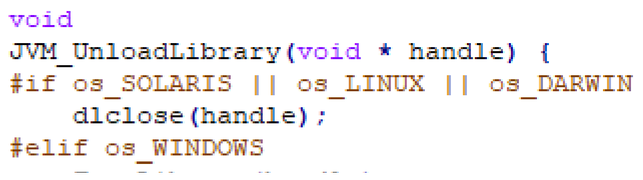
On Linux and Darwin a call to **sysconf(_SC_NPROCESSORS_ONLN)** is utiized which in Windows is replaced with

SYSTEM_INFO systemInfo = {0};
GetSystemInfo(&systemInfo);
return systemInfo.dwNumberOfProcessors;

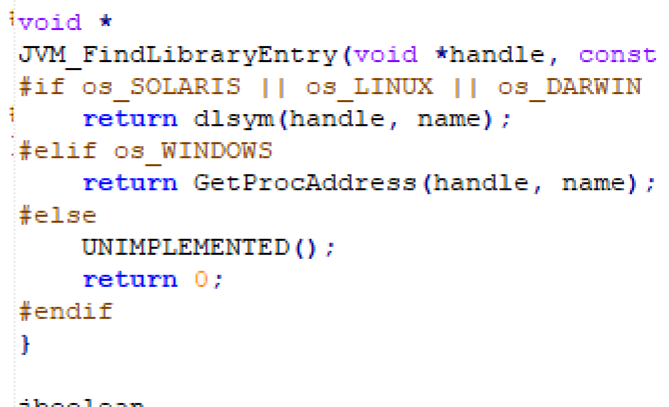
Also there are functions used to load shared libraries like



```
void *
JVM_LoadLibrary(const char *name) {
#if os_SOLARIS || os_LINUX || os_DARWIN
    return dlopen(name, RTLD_LAZY);
#elif os_WINDOWS
    return LoadLibraryA(name);
```



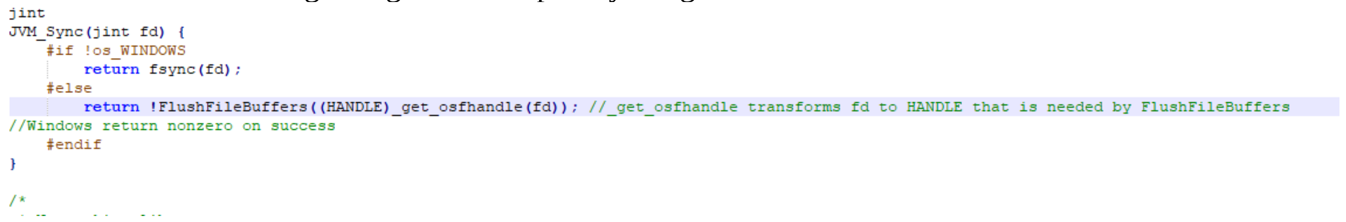
```
void
JVM_UnloadLibrary(void * handle) {
#if os_SOLARIS || os_LINUX || os_DARWIN
    dlclose(handle);
#elif os_WINDOWS
```



```
void *
JVM_FindLibraryEntry(void *handle, const char *name) {
#if os_SOLARIS || os_LINUX || os_DARWIN
    return dlsym(handle, name);
#elif os_WINDOWS
    return GetProcAddress(handle, name);
#else
    UNIMPLEMENTED();
    return 0;
#endif
}
```

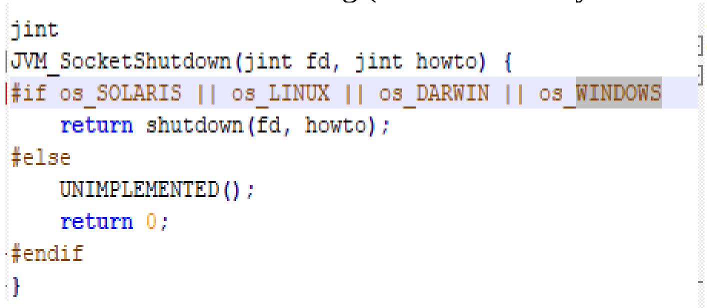
Other functions include signal sending which unfortunately is not supported on Windows. Windows signals can only get from the same process/ thread that sent (ie. raised) them (just like exceptions). In order to achieve this the win32 Raise() function was used.

There also functions regarding file descriptor syncing

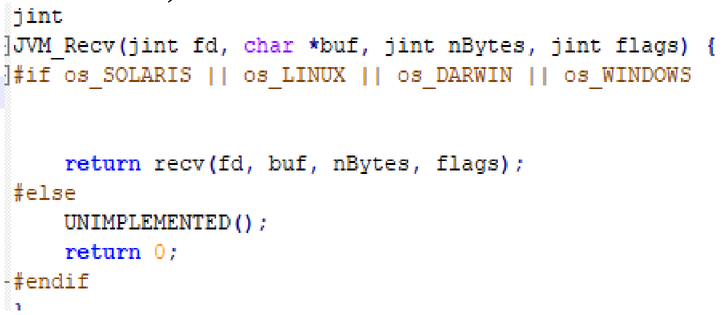


```
jint
JVM_Sync(jint fd) {
    #if !os_WINDOWS
        return fsync(fd);
    #else
        return !FlushFileBuffers((HANDLE)_get_osfhandle(fd)); //_get_osfhandle transforms fd to HANDLE that is needed by FlushFileBuffers
//Windows return nonzero on success
    #endif
}

/*
```

As well as socket handling (those are exactly the same in all OSes)



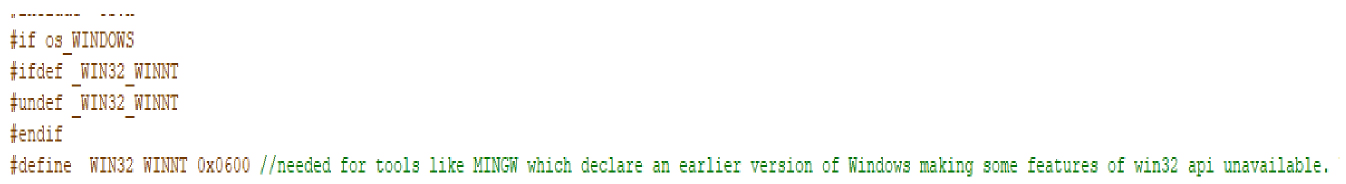
```
jint
JVM_SocketShutdown(jint fd, jint howto) {
#if os_SOLARIS || os_LINUX || os_DARWIN || os_WINDOWS
    return shutdown(fd, howto);
#else
    UNIMPLEMENTED();
    return 0;
#endif
}
```



```
jint
JVM_Recv(jint fd, char *buf, jint nBytes, jint flags) {
#if os_SOLARIS || os_LINUX || os_DARWIN || os_WINDOWS


    return recv(fd, buf, nBytes, flags);
#else
    UNIMPLEMENTED();
    return 0;
#endif
```

On windows we must also link with Library lws2_32 at run time (remember the changes at Makefiles where we included 2 libraries at Linking Flags)

One can also notice this macro on the top of that file



```
#if os_WINDOWS
#ifdef _WIN32_WINNT
#undef _WIN32_WINNT
#endif
#define  WIN32 WINNT 0x0600 //needed for tools like MINGW which declare an earlier version of Windows making some features of win32 api unavailable.
```

Here, 0x0600 means Windows Vista and we define it in order to be able to use attributes that are available from that OS and on. Windows Header Files put some of their datatypes and structs inside
#if WIN32_WINNT >= something
….
#endif

in order to prevent users running Windows Versions older than the oldest supported one from using unsupported attributes so we need to define this macro in order to “tell” the compiler that we can/want to use them.

threadLocals.c

This source file regards Thread Local Storage which is in simpler words the ability to use local variables without them constituting arguments to the thread function or being created inside the function. However, at the same time, they are not global variables either. Only the thread that creates them can see them.

Below follows a table showing win32 functions equivalent to TLS Linux functions used in the source file

pthread_key_create	TlsAlloc()
pthread_getspecific(theThreadLocalsKey);	TlsGetValue(theThreadLocalsKey);
pthread_setspecific(theThreadLocalsKey, (void *) tlBlock);	TlsSetValue(theThreadLocalsKey, (LPVOID) tlBlock);

TheThreadLocalKey is defined as DWORD (int) on Windows and pthread_key_t on Linux

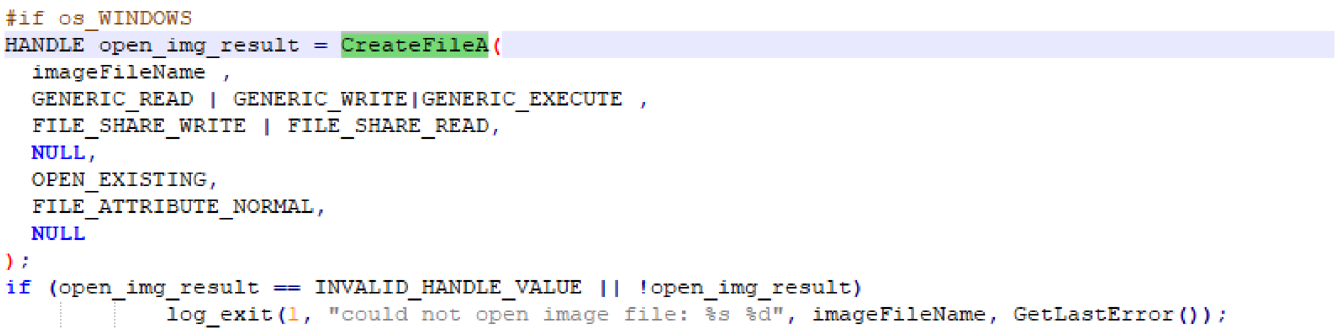
image.c

This source file is responsible for opening the boot image file, reading its info and map into memory.

An interesting thing to mention is that on Linux
fd = open(imageFileName, _O_RDWR);

is used to open the file. The same exact call exists on Windows. However, on Windows it opens the file in text mode. What's interesting with Windows and text mode is that Ctrl-Z character is interpreted as the EOF character. Our boot image file happened to contain such characters so it led the read() function to return before reading all the available bytes. In order to resolve the issue we must open the image using
fd = open(imageFileName, _O_RDWR|_O_BINARY); on Windows

We also need to open once more the image file using CreateFileA function because that's the only way to allow execution rights to the image. On linux it is enough to open with open() while on Windows one must do something like it in order to be allowed to map the image to a page with execution rights



```
#if os_WINDOWS
HANDLE open_img_result = CreateFileA(
  imageFileName ,
  GENERIC_READ | GENERIC_WRITE|GENERIC_EXECUTE ,
  FILE_SHARE_WRITE | FILE_SHARE_READ,
  NULL,
  OPEN_EXISTING,
  FILE_ATTRIBUTE_NORMAL,
  NULL
);
if (open_img_result == INVALID_HANDLE_VALUE || !open_img_result)
            log_exit(1, "could not open image file: %s %d", imageFileName, GetLastError());
```

virtualmemory.c

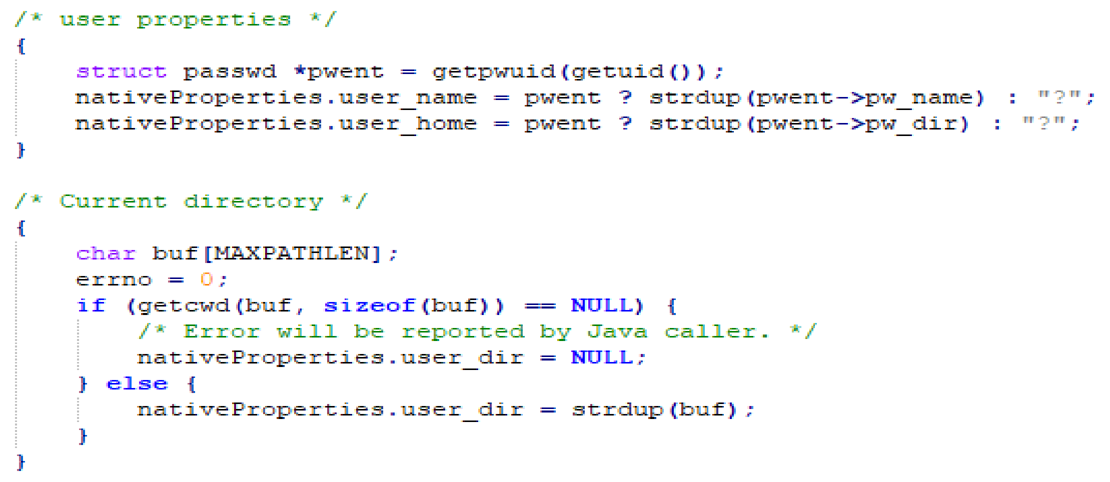
This where all the function responsible for file mapping reside. Linux uses mmap() and mprotect() for mapping and protecting pages while on Windows we need VirtualAlloc when mapping memory without using any file and a combination of CreateFileMapping and MapViewOfFile while mapping actual files. For protecting pages we use VirtualProtect.

The code of this file is really lenthy and complex so you can see it here :https://github.com/mihalis341/Maxine-VM/blob/develop/com.oracle.max.vm.native/share/virtualMemory.c with all the necessary comments.

maxine.c

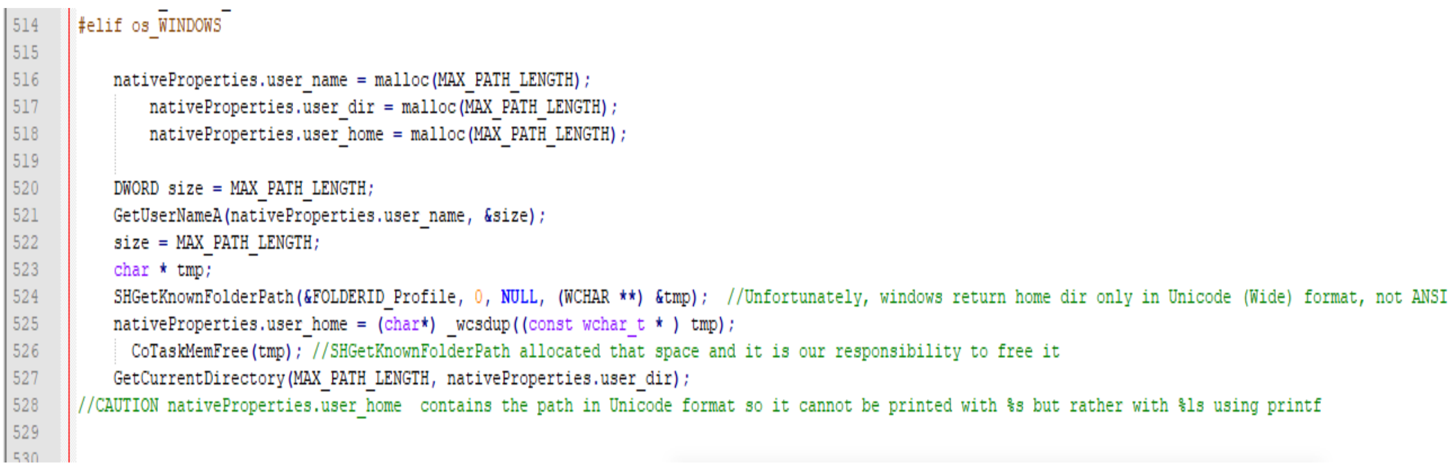
Contains the maxine() function loaded by maxvm.c using windows loadlibrary functions shown in jvm.c

It also provides a function for getting user’s home dir, current working directory and user’s name which is implemented using getpwuid() on Linux



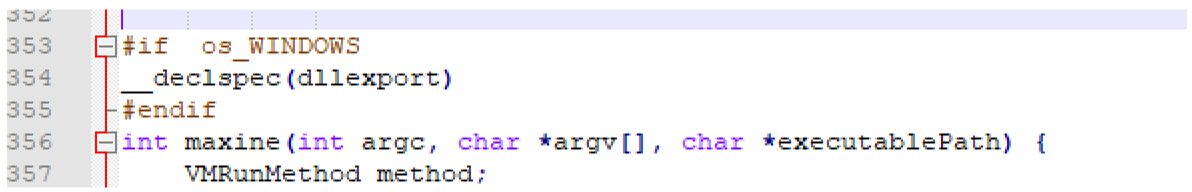
```c
/* user properties */
{
    struct passwd *pwent = getpwuid(getuid());
    nativeProperties.user_name = pwent ? strdup(pwent->pw_name) : "?";
    nativeProperties.user_home = pwent ? strdup(pwent->pw_dir) : "?";
}

/* Current directory */
{
    char buf[MAXPATHLEN];
    errno = 0;
    if (getcwd(buf, sizeof(buf)) == NULL) {
        /* Error will be reported by Java caller. */
        nativeProperties.user_dir = NULL;
    } else {
        nativeProperties.user_dir = strdup(buf);
    }
}
```

and a combination of GetUserNameA(), SHGetKnownFolderPath() and GetCurrentDirectory() on Windows



```c
#elif os_WINDOWS

    nativeProperties.user_name = malloc(MAX_PATH_LENGTH);
        nativeProperties.user_dir = malloc(MAX_PATH_LENGTH);
        nativeProperties.user_home = malloc(MAX_PATH_LENGTH);

    DWORD size = MAX_PATH_LENGTH;
    GetUserNameA(nativeProperties.user_name, &size);
    size = MAX_PATH_LENGTH;
    char * tmp;
    SHGetKnownFolderPath(&FOLDERID_Profile, 0, NULL, (WCHAR **) &tmp);  //Unfortunately, windows return home dir only in Unicode (Wide) format, not ANSI
    nativeProperties.user_home = (char*) _wcsdup((const wchar_t * ) tmp);
      CoTaskMemFree(tmp); //SHGetKnownFolderPath allocated that space and it is our responsibility to free it
    GetCurrentDirectory(MAX_PATH_LENGTH, nativeProperties.user_dir);
//CAUTION nativeProperties.user_home  contains the path in Unicode format so it cannot be printed with %s but rather with %ls using printf
```

It’s also worth noting that the “maxine” function must be “decorated” using this annotation __declspec(dllexport) right before its signature.



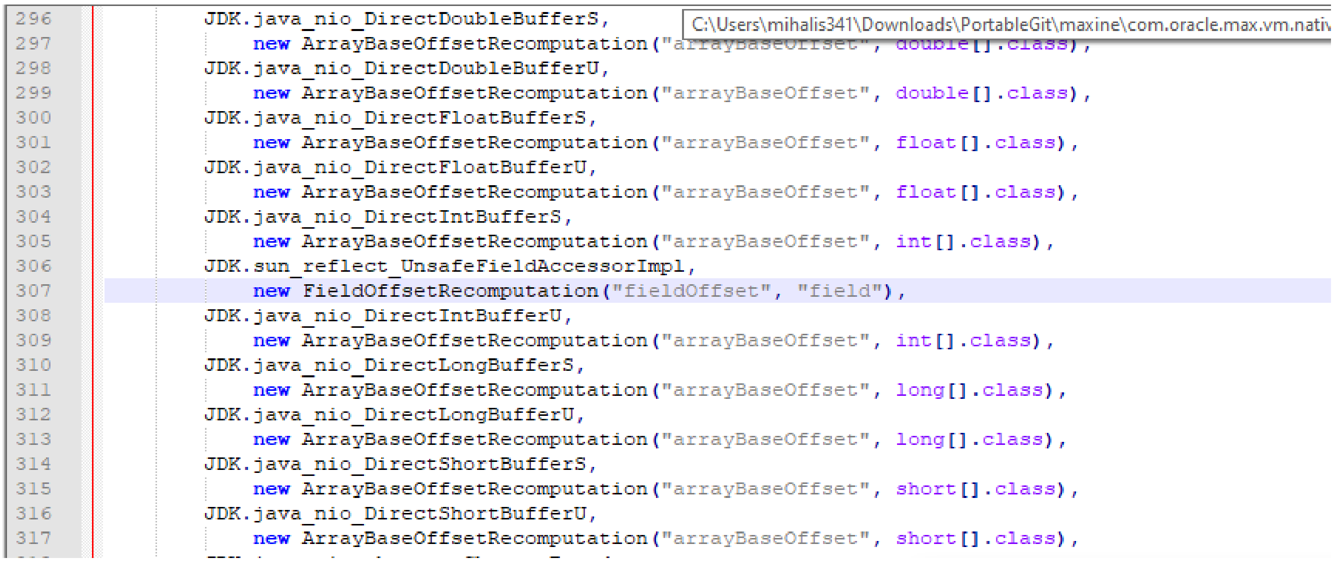
```c
#if  os_WINDOWS
__declspec(dllexport)
#endif
int maxine(int argc, char *argv[], char *executablePath) {
    VMRunMethod method;
```

This makes the symbol visible inside the DLL and accessible by others who load jvm.dll (in that case, maxvm.exe)

These were some of the source files changed. There others containing interesting changes like conditions.c
Also the full native code can be found here
https://github.com/mihalis341/Maxine-VM/tree/develop/com.oracle.max.vm.native

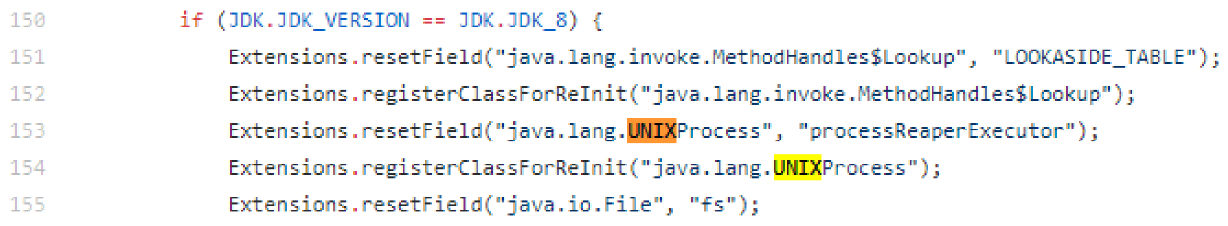
Boot Image Creation

In order to get able to create the image file we had to go through a lot. Initially, we faced many unexplainable errors during the compilation of some classes with no sufficient info about it. During this process it was needed some interceptions inside the class JDKInterceptor.Java. This class is responsible for resetting the values of some fields of specific classes that for some reason cause crashes while running the boot image file. An example of some interceptions is here



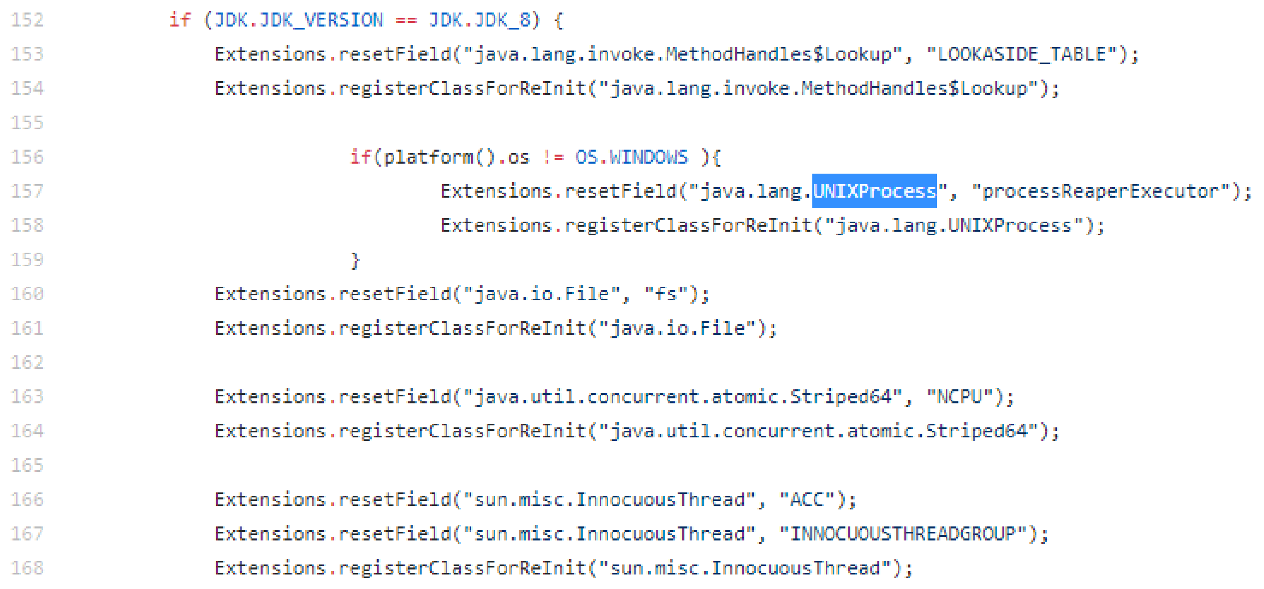
```
C:\Users\mihalis341\Downloads\PortableGit\maxine\com.oracle.max.vm.nativ
JDK.java_nio_DirectDoubleBufferS,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", double[].class),
JDK.java_nio_DirectDoubleBufferU,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", double[].class),
JDK.java_nio_DirectFloatBufferS,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", float[].class),
JDK.java_nio_DirectFloatBufferU,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", float[].class),
JDK.java_nio_DirectIntBufferS,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", int[].class),
JDK.sun_reflect_UnsafeFieldAccessorImpl,
    new FieldOffsetRecomputation("fieldOffset", "field"),
JDK.java_nio_DirectIntBufferU,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", int[].class),
JDK.java_nio_DirectLongBufferS,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", long[].class),
JDK.java_nio_DirectLongBufferU,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", long[].class),
JDK.java_nio_DirectShortBufferS,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", short[].class),
JDK.java_nio_DirectShortBufferU,
    new ArrayBaseOffsetRecomputation("arrayBaseOffset", short[].class),
```

Finally, I found out that the main reason behind the strange compilation errors of the boot image was the inability to load that class https://github.com/beehive-lab/Maxine-VM/blob/develop/com.sun.max/src/com/sun/max/config/jdk/Package.java . This class contains info about known problematic classes that break compilation and adds those to a blacklist. However, on Windows this class failed to load during image generation so nothing was added to the compilation blacklist. As a result, we faced many needless and most likely unsolvable errors for a pretty big period of time. That class attempts to get loaded here
https://github.com/beehive-lab/Maxine-VM/blob/develop/com.sun.max/src/com/sun/max/config/BootImagePackage.java#L246 however, all exceptions are handled silently so they are not visible on output. The reason that previous class failed to load, is its dependency on java.lang.UNIXprocess class which is visible on lines 153-154. Those classes do not exist on any windows JDK for obvious reasons so trying to load a class that depends on them throws NoClassDefFoundError

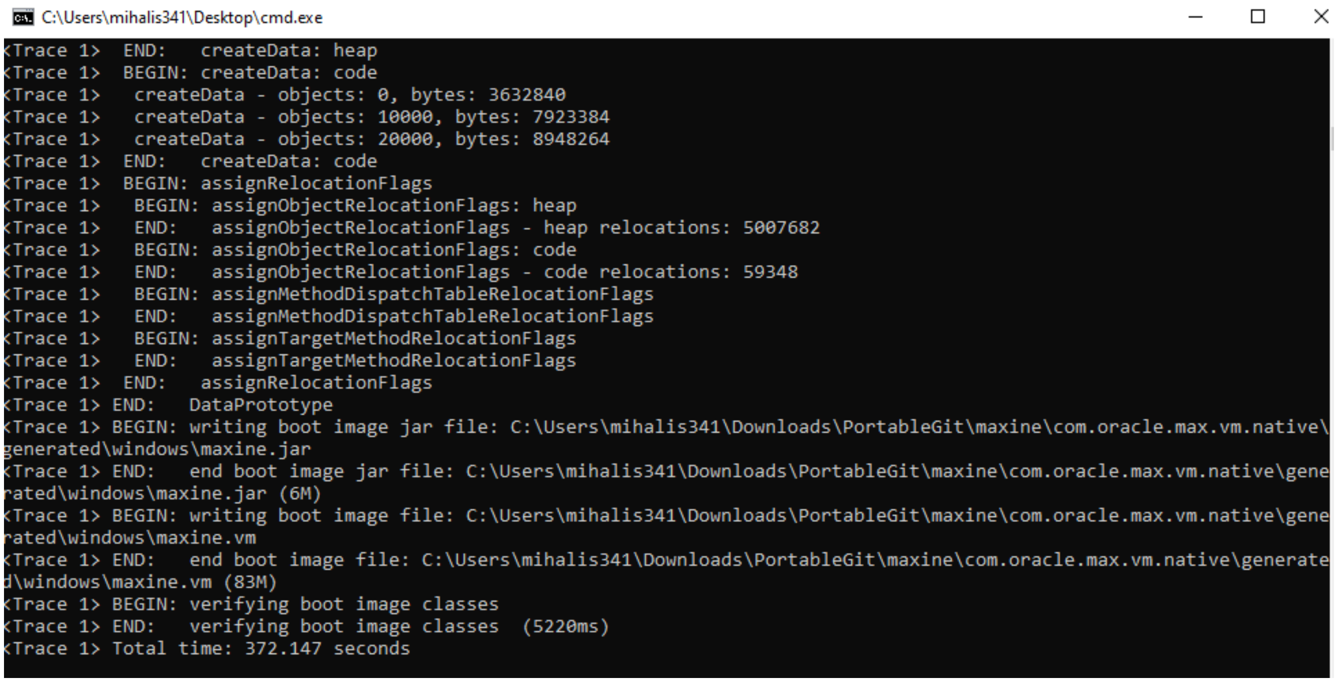


```
150         if (JDK.JDK_VERSION == JDK.JDK_8) {
151             Extensions.resetField("java.lang.invoke.MethodHandles$Lookup", "LOOKASIDE_TABLE");
152             Extensions.registerClassForReInit("java.lang.invoke.MethodHandles$Lookup");
153             Extensions.resetField("java.lang.UNIXProcess", "processReaperExecutor");
154             Extensions.registerClassForReInit("java.lang.UNIXProcess");
155             Extensions.resetField("java.io.File", "fs");
```

changing those lines like that did the trick



```
        if (JDK.JDK_VERSION == JDK.JDK_8) {
            Extensions.resetField("java.lang.invoke.MethodHandles$Lookup", "LOOKASIDE_TABLE");
            Extensions.registerClassForReInit("java.lang.invoke.MethodHandles$Lookup");

                    if(platform().os != OS.WINDOWS ){
                            Extensions.resetField("java.lang.UNIXProcess", "processReaperExecutor");
                            Extensions.registerClassForReInit("java.lang.UNIXProcess");
                    }
            Extensions.resetField("java.io.File", "fs");
            Extensions.registerClassForReInit("java.io.File");

            Extensions.resetField("java.util.concurrent.atomic.Striped64", "NCPU");
            Extensions.registerClassForReInit("java.util.concurrent.atomic.Striped64");

            Extensions.resetField("sun.misc.InnocuousThread", "ACC");
            Extensions.resetField("sun.misc.InnocuousThread", "INNOCUOUSTHREADGROUP");
            Extensions.registerClassForReInit("sun.misc.InnocuousThread");
```

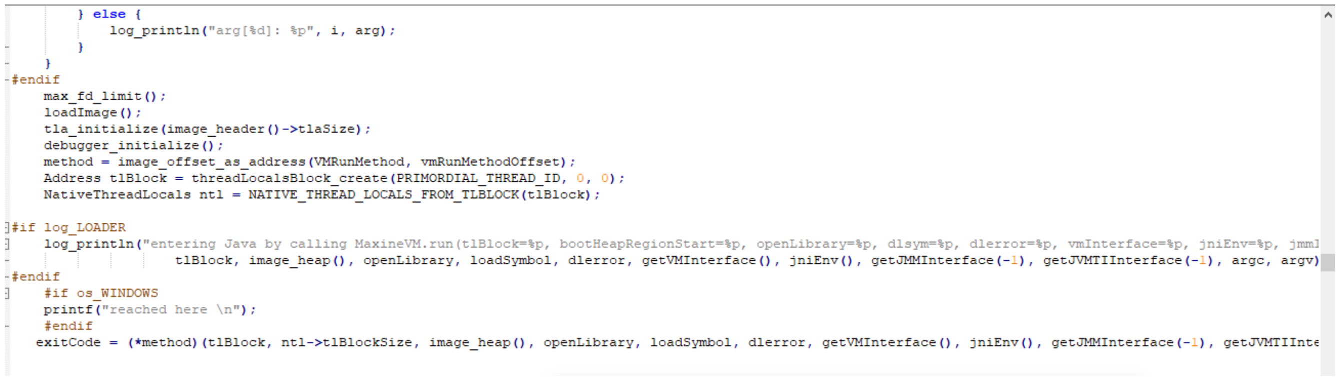


C:\Users\mihalis341\Desktop\cmd.exe

```
<Trace 1>  END:   createData: heap
<Trace 1>  BEGIN: createData: code
<Trace 1>   createData - objects: 0, bytes: 3632840
<Trace 1>   createData - objects: 10000, bytes: 7923384
<Trace 1>   createData - objects: 20000, bytes: 8948264
<Trace 1>  END:   createData: code
<Trace 1>  BEGIN: assignRelocationFlags
<Trace 1>   BEGIN: assignObjectRelocationFlags: heap
<Trace 1>   END:   assignObjectRelocationFlags - heap relocations: 5007682
<Trace 1>   BEGIN: assignObjectRelocationFlags: code
<Trace 1>   END:   assignObjectRelocationFlags - code relocations: 59348
<Trace 1>   BEGIN: assignMethodDispatchTableRelocationFlags
<Trace 1>   END:   assignMethodDispatchTableRelocationFlags
<Trace 1>   BEGIN: assignTargetMethodRelocationFlags
<Trace 1>   END:   assignTargetMethodRelocationFlags
<Trace 1>  END:   assignRelocationFlags
<Trace 1> END:   DataPrototype
<Trace 1> BEGIN: writing boot image jar file: C:\Users\mihalis341\Downloads\PortableGit\maxine\com.oracle.max.vm.native\
generated\windows\maxine.jar
<Trace 1> END:   end boot image jar file: C:\Users\mihalis341\Downloads\PortableGit\maxine\com.oracle.max.vm.native\gene
rated\windows\maxine.jar (6M)
<Trace 1> BEGIN: writing boot image file: C:\Users\mihalis341\Downloads\PortableGit\maxine\com.oracle.max.vm.native\gene
rated\windows\maxine.vm
<Trace 1> END:   end boot image file: C:\Users\mihalis341\Downloads\PortableGit\maxine\com.oracle.max.vm.native\generate
d\windows\maxine.vm (83M)
<Trace 1> BEGIN: verifying boot image classes
<Trace 1> END:   verifying boot image classes  (5220ms)
<Trace 1> Total time: 372.147 seconds
```

so now, the boot image gets compiled successfully.

Reading and loading the boot image from the native machine code works well. Image.c maxine.c and virtualmemory.c contain much defensive code (eg. asserts) to ensure that the file is correctly loaded and mapped into memory. However, we currently get a crash just after the “reached here” print inside the **maxine() function** in maxine.c just when the VMRun Java method gets invoked



```
            } else {
                log_println("arg[%d]: %p", i, arg);
            }
        }
#endif
    max_fd_limit();
    loadImage();
    tla_initialize(image_header()->tlaSize);
    debugger_initialize();
    method = image_offset_as_address(VMRunMethod, vmRunMethodOffset);
    Address tlBlock = threadLocalsBlock_create(PRIMORDIAL_THREAD_ID, 0, 0);
    NativeThreadLocals ntl = NATIVE_THREAD_LOCALS_FROM_TLBLOCK(tlBlock);

#if log_LOADER
    log_println("entering Java by calling MaxineVM.run(tlBlock=%p, bootHeapRegionStart=%p, openLibrary=%p, dlsym=%p, dlerror=%p, vmInterface=%p, jniEnv=%p, jmmI
                    tlBlock, image_heap(), openLibrary, loadSymbol, dlerror, getVMInterface(), jniEnv(), getJMMInterface(-1), getJVMTIInterface(-1), argc, argv)
#endif
    #if os_WINDOWS
    printf("reached here \n");
    #endif
   exitCode = (*method)(tlBlock, ntl->tlBlockSize, image_heap(), openLibrary, loadSymbol, dlerror, getVMInterface(), jniEnv(), getJMMInterface(-1), getJVMTIInte
```

Before that point, many checks have been performed inside the code that everything is correct. In addition, I have also used my code to execute a small memory mapped file with machine code and it appears to work correctly so I believe my code, regarding the loading and mapping of the image file in memory, is correct.
(test: https://github.com/mihalis341/Maxine-VM/blob/develop/com.oracle.max.vm.native/windows_tests/memory_map_test.c)

In addition, we further looked into **com.sun.max.vm.MaxineVM.run()** function to understand more about the crash.

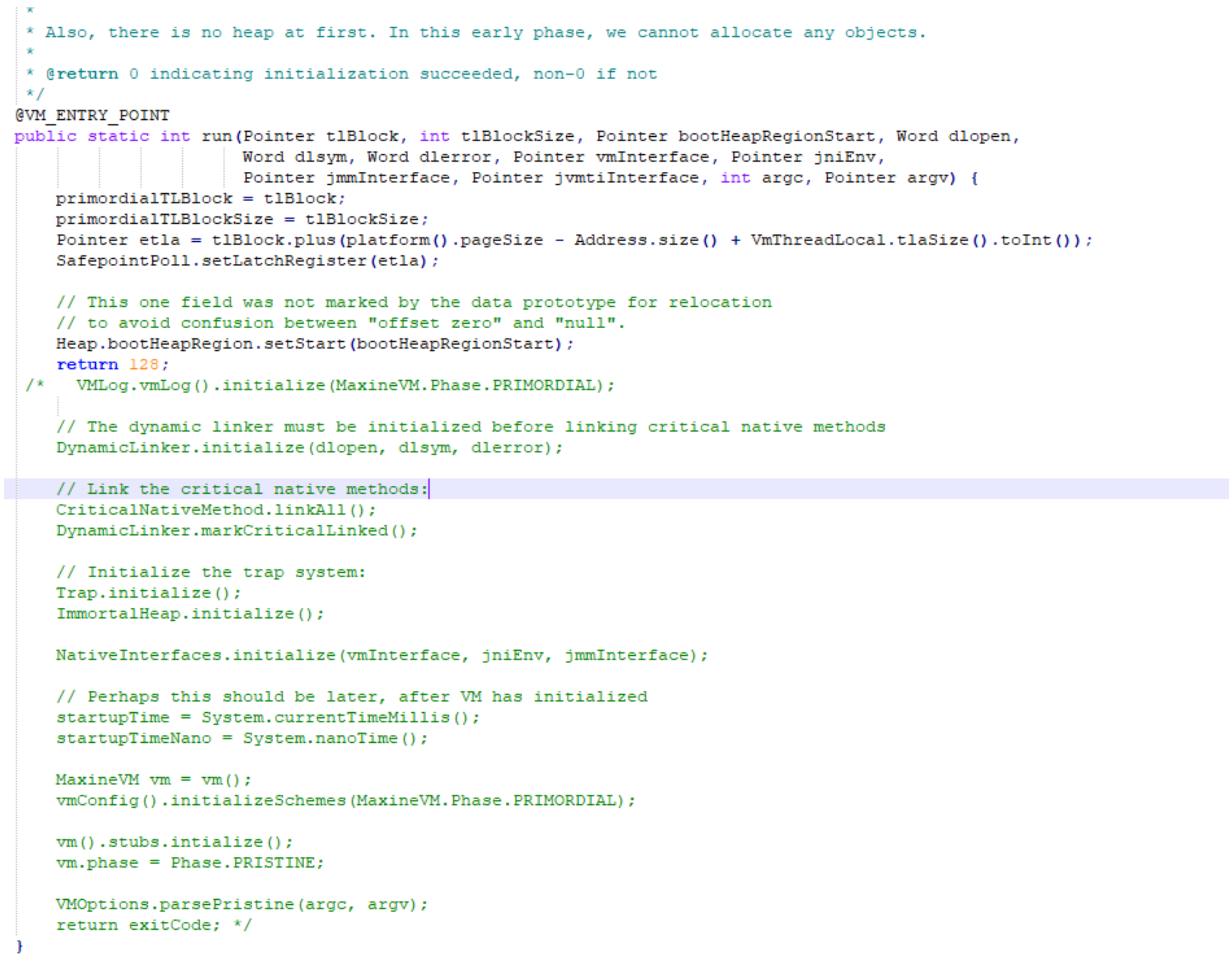
We tried commenting out some of the code in that function in order to figure out where it crashes and found out that the Java code is successfully executed until (including) the line where the bootHeapRegion gets set. After that, some of the next functions (eg. VMLog.vmLog().initialize()) cause the crash.
In my understanding, there is **not** some problematic code in those function that make the program crash but rather the "location" of those functions in memory is the problem. The same crash occurs even if I completely replace the body of VMLog.vmLog().initialize() with a simple "return;" and rebuild maxine and its image.

The reason behind the crash might be some needed interception for the new OS the VM is getting ported to.

As a test one can try the following:

Comment all the code after the setting of bootheap region start in **com.sun.max.vm.MaxineVM.run()**

and then return a random number (eg.128). (as depicted below)



```java
 *
 * Also, there is no heap at first. In this early phase, we cannot allocate any objects.
 *
 * @return 0 indicating initialization succeeded, non-0 if not
 */
@VM_ENTRY_POINT
public static int run(Pointer tlBlock, int tlBlockSize, Pointer bootHeapRegionStart, Word dlopen,
                      Word dlsym, Word dlerror, Pointer vmInterface, Pointer jniEnv,
                      Pointer jmmInterface, Pointer jvmtiInterface, int argc, Pointer argv) {
    primordialTLBlock = tlBlock;
    primordialTLBlockSize = tlBlockSize;
    Pointer etla = tlBlock.plus(platform().pageSize - Address.size() + VmThreadLocal.tlaSize().toInt());
    SafepointPoll.setLatchRegister(etla);

    // This one field was not marked by the data prototype for relocation
    // to avoid confusion between "offset zero" and "null".
    Heap.bootHeapRegion.setStart(bootHeapRegionStart);
    return 128;
 /*   VMLog.vmLog().initialize(MaxineVM.Phase.PRIMORDIAL);

    // The dynamic linker must be initialized before linking critical native methods
    DynamicLinker.initialize(dlopen, dlsym, dlerror);

    // Link the critical native methods:
    CriticalNativeMethod.linkAll();
    DynamicLinker.markCriticalLinked();

    // Initialize the trap system:
    Trap.initialize();
    ImmortalHeap.initialize();

    NativeInterfaces.initialize(vmInterface, jniEnv, jmmInterface);

    // Perhaps this should be later, after VM has initialized
    startupTime = System.currentTimeMillis();
    startupTimeNano = System.nanoTime();

    MaxineVM vm = vm();
    vmConfig().initializeSchemes(MaxineVM.Phase.PRIMORDIAL);

    vm().stubs.intialize();
    vm.phase = Phase.PRISTINE;

    VMOptions.parsePristine(argc, argv);
    return exitCode; */
}
```

Rebuild maxine and image.
Afterwards, go back to **maxine.c** and after the line that invokes the VMRunFunction

(*exitCode = (*method)(tlBlock, ntl->tlBlockSize, image_heap(), openLibrary, loadSymbol, dlerror, getVMInterface(), jniEnv(), getJMMInterface(-1), getJVMTIInterface(-1), argc, argv);*
)

add code to print exitCode value (eg.with printf). Recompile and run **maxvm.exe** in the native code directory (you can also try mx helloworld from base directory) You will see that 128 (or the number you used) gets printed which indicates that the Java Code of VMRun gets indeed **correctly** executed until the point we have left it uncommented. This proves that the image file is correctly created and mapped into memory, the offset of the VMRun function in it is correctly calculated in our C code etc. It remains unknown why the commented Java code causes a segmentation fault (when not commented out of course!) even if all the classes it uses are included in the image file.