Virginia Tech ❖ Bradley Department of Electrical and Computer Engineering

ECE 5984 Virtualization Technologies

Project 1: Unikernel File-System

Helper guide

This document is a technical guide offered to help you familiarize yourself with HermitCore. It gives also a few technical starting points for the project.

Requirements

The Linux distribution that should be used to develop with HermitCore is **Ubuntu 16.04 64 bits**. While other distribution may work, the students are expected to use this specific distribution and version to avoid tools and library versions compatibility issues.

Most of the development can be realized in a virtual machine (ex: VirtualBox): indeed, HermitCore supports running on top of the Qemu emulator that does not require hardware virtualization support. However, emulation being particularly slow, note that the performance evaluation should not be realized within an emulator: for that evaluation HermitCore should run on the two hypervisors it supports that make use of hardware virtualization acceleration: Qemu/KVM or Uhyve. Thus, it is highly recommended to install Ubuntu natively from the start.

The size of the partition Ubuntu is installed should be at least 30 GB, to be on the safe side.

Installing, building HermitCore, and launching an example unikernel

HermitCore sources are divided into two main software components:

1. The sources of the unikernel itself;
2. The toolchain sources.

The toolchain comprises a set of tools: gcc, *binutils* (ld, ar, etc.), a C library (*newlib*), and a pthread library. These tools has been ported to generate HermitCore unikernel binaries, by compiling the unikernel sources into a static library and linking the compiled sources of an application with this library.

HermitCore is available here: `https://github.com/RWTH-OS/HermitCore`.

Dependencies

Ubuntu packages

The following packages are needed to start developing with HermitCore:

```
$ sudo apt update && sudo apt install -y git build-essential nasm flex bison
    texinfo libmpfr-dev libmpc-dev libgmp-dev
```

Toolchain

The toolchain can be compiled from sources (it is available here with installation instruction: `https://github.com/RWTH-OS/hermit-toolchain`) but pre-compiled Debian packages are available, making the toolchain installation very simple:

```
$ echo "deb [trusted=yes] https://dl.bintray.com/rwth-os/hermitcore vivid main" |
   sudo tee -a /etc/apt/sources.list
$ sudo apt update
$ sudo apt install binutils-hermit newlib-hermit pthread-embedded-hermit gcc-hermit
    libhermit qemu-system-x86
```

The toolchain is then installed in `/opt/hermit/`.

Cmake

HermitCore uses a recent version of Cmake to generate the kernel build infrastructure. The version present in the Ubuntu repositories being too old, we have to install it from source:

```
# First uninstall any existing cmake installation
$ sudo apt remove cmake
# Download and install the version 3.10.2 of cmake
$ cd /tmp
$ wget https://cmake.org/files/v3.10/cmake-3.10.2.tar.gz
$ tar xf cmake-3.10.2.tar.gz
$ cd cmake-3.10.2/
$ ./configure && make && sudo make install
```

Downloading and building the kernel

Note that here we assume that the toolchain is installed in `/opt/hermit`. Once compiled the kernel also needs to be installed in that same location (this is the default install path).

```
$ git clone https://github.com/RWTH-OS/HermitCore.git
$ cd HermitCore

# Don't forget to fetch the submodules before any further step!
$ git submodule init
$ git submodule update

# Create a build directory and compile the kernel
$ mkdir build && cd build/
$ cmake ..
$ make -j`nproc` && sudo make install

# Install the kernel
$ sudo make install
```

Launching an example unikernel

A few example applications are available in HermitCore sources, for example you can have a look in `HermitCore/usr/tests` which contains the sources of a few simple test applications. Once the kernel compile steps presented above are done, the unikernel binaries corresponding to these test applications are in /opt/hermit /x86_64-hermit/extra/tests/.

To launch an example unikernel using the Qemu/KVM hypervisor with 1GB of RAM:

```
$ HERMIT_ISLE=qemu HERMIT_KVM=0 HERMIT_MEM=1G /opt/hermit/bin/proxy /opt/hermit/
    x86_64-hermit/extra/tests/hello
```

You should see a warning related to Qemu (it can be ignored), and a few lines outputted by the unikernel. A few notable things about this command:

- HERMIT_ISLE defines the hypervisor to use. In the example above, qemu in emulation mode is used (KVM is disabled through the use of HERMIT_KVM=0. For Uhyve, another hypervisor supported by HermitCore, use HERMIT_ISLE=uhyve.
- HERMIT_MEM obviously defines the amount of memory given to the unikernel
- proxy is a regular Linux program that:
  - Makes the connection with qemu if that hypervisor is used;
  - Contains the Uhyve hypervisor if Uhyve is used instead. The sources of proxy are available in HermitCore/tools/proxy.c
- hello is the unikernel binary.

Other interesting environment variables related to HermitCore (HERMIT_*) are described in HermitCore's README.md file.

Writing your own unikernel

In order to write your own unikernel, there are two possibilities:

- Write your application code and place it in HermitCore/usr/tests, then edit the Cmake metadata files (HermitCore/usr/tests/CMakeLists.txt, or
- Write your application in an arbitrary folder and use a Makefile to call the toolchain directly.

Here we take the second approach through the use of this simple Makefile:

```
CC=/opt/hermit/bin/x86_64-hermit-gcc
SRC=prog.c
PROG=prog
CFLAGS= -g
LDFLAGS=
PROXY=/opt/hermit/bin/proxy
MEM=4G

all: $(PROG)
	$(CC) $(CFLAGS) $(SRC) -o $(PROG) $(LDFLAGS)

test: all
	HERMIT_MEM=$(MEM) HERMIT_ISLE=qemu HERMIT_KVM=0 $(PROXY) $(PROG)
```

We can use it to compile this simple application (prog.c, needs to be in the same folder as the Makefile) :

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("hello world!\n");
  printf("argv[] = %s\n", argv[0]);
  return 0;
}
```

The program can be compiled using `make`, and a `make test` target is available for easy execution with Qemu.

HermitCore kernel sources organization

A few interesting folders are:

- `arch/x86/`: contains the architecture-specific code of the kernel. Around 80-90% of HermitCore code is architecture specific, so most of the code is located there;
- `cmake`: contains part of the Cmake files used to generate HermitCore's Makefiles. Also, have a look at the files named `CMakeLists.txt` spread all over the HermitCore repository;
- `include/`: Kernel include files, in particular `include/hermit`;
- `kernel/`: non-architecture-specific kernel files: in particular system calls implementation and late kernel initialization functions;
- `mm/`: non-architecture-specific memory management
- `tools/`: various tools, in particular `proxy` and Uhyve
- `usr/`: some libraries and example programs.

Modifying a system call

During the project, some HermitCore system calls will need to be modified, for example the ones that are related to file-system operations and that are currently forwarded to the host.

The system calls implementations are present in `HermitCore/kernel/syscall.c`. Similarly to the Linux kernel, each the function implementing each system call is named `sys_<syscall name>`, for example `sys_read`.

You'll notice that for the host-forwarded system calls, the implementation is different according to the hypervisor used (Qemu vs Uhyve). The Uhyve implementation are way simpler as with that hypervisor, it is simple to share memory for communication between the host and the unikernel. For Qemu, a simple TCP/IP client/server is used, the client being the unikernel, and the server being `proxy`.

The project aims to make all file-system operations internal to the unikernel, so calls to the host in that context should not be performed anymore, and file-system operations should probably be independent from the hypervisor used.

Adding a system call

Adding a system call necessitates (1) implementing the system call in the HermitCore kernel, and (2) adding support for that system call in the C library (HermitCore uses `newlib`). Indeed, as for regular, non-unikernel applications, the C library is an intermediate layer between the application and the kernel. The C library offers high-level functions to the application, and perform the system calls itself.

In the following paragraphs we present how to add 2 new system calls in HermitCore: the first one is a brand new, dummy system call named `my_syscall`. The second one is a dummy implementation of an existing Linux file-system related system call that is not supported by HermitCore: `rename`.

Kernel level implementations

Edit the following file: `HermitCore/kernel/syscall.c` and add the two system calls implementations:

```
int sys_my_syscall(int x)
{
  return x + 42;
}

int sys_rename(const char *oldpath, const char *newpath)
{
  /* you will do the implementation later, for now just return success :) */
  return 0;
}
```

Then add the prototypes in `HermitCore/include/hermit/syscall.h`:

```
int sys_my_syscall(int x);
int sys_rename(const char *oldpath, const char *newpath);
```

Recompile and re-install the kernel:

```
make -j`nproc` -C HermitCore/build && sudo make install -C HermitCore/build
```

C library level support

Automake

Implementing new system calls includes modifying the C library, *newlib*. Adding new source files to the library makes that we will need to re-generate newlib's `configure` script infrastructure using `automake`. We will need to download and compile from source a specific version of this tool, along with a specific version of *autoconf*: automake v1.11.6:

```
# Autoconf:
$ wget https://ftp.gnu.org/gnu/autoconf/autoconf-2.64.tar.bz2
$ tar xf autoconf-2.64.tar.bz2
$ cd autoconf-2.64/
$ ./configure
$ make
$ sudo make install
# Automake:
$ wget https://ftp.gnu.org/gnu/automake/automake-1.11.6.tar.gz
$ tar xf automake-1.11.6.tar.gz
$ cd automake-1.11.6/
$ ./configure
$ make
$ sudo make install
```

Automake is then located in `/usr/local/bin/automake`. When invoking it make sure to use the full path, otherwise the system might default on a potential newer version already installed.

Adding new system calls support

As we need to compile newlib for HermitCore, it is a good idea to put the toolchain binary folder in the path, by running that command in a terminal or adding it to the `.bashrc` file:

```
export PATH=$PATH:/opt/hermit/bin
```

We will use this specific version of newlib that is compatible with HermitCore and with the previously downloaded version of automake. it can be downloaded and installed as follows:

```
$ git clone https://github.com/olivierpierre/newlib.git
$ cd newlib
$ ./configure --target=x86_64-hermit --prefix=/opt/hermit --disable-shared --
    disable-multilib --enable-lto --enable-newlib-hw-fp --enable-newlib-io-c99-
    formats --enable-newlib-multithreaded
$ make -j`nproc`
$ sudo PATH=$PATH:/opt/hermit/bin make install
# add the path when compile the newlib
```

Hermit system call implementations are in `newlib/newlib/libc/sys/hermit`. Let's add our 2 new system calls implementations there. First, `newlib/newlib/libc/sys/hermit/my_syscall.c`:

```
#include "config.h"
#include <_ansi.h>
#include <_syslist.h>
#include "syscall.h"

int
_DEFUN (my_syscall, (x),
  int x)
{
  /* call the kernel implementation */
  return sys_my_syscall(x);
}
```

Second, `newlib/newlib/libc/sys/hermit/rename.c`:

```
#include "config.h"
#include <reent.h>
#include <_ansi.h>
#include <_syslist.h>
#include <errno.h>
#include "syscall.h"
#include "warning.h"

int
_DEFUN (rename, (oldpath, newpath),
  const char *oldpath  _AND
  const char *newpath)
{
  return _rename_r(_REENT, oldpath, newpath);
}

int
_DEFUN (_rename_r, (p, oldpath, newpath),
  struct _reent *p _AND
        const char *oldpath  _AND
        const char *newpath)
{
  int ret;

  /* call HermitCore implementation */
  ret = sys_rename(oldpath, newpath);
  if (ret < 0) {
    p->_errno = -ret;
    ret = -1;
  }

  return ret;
}
```

Note how `errno` is managed in a per-thread way by calling `_rename_r` which first parameter is a per-thread data structure, more precisely the data structure corresponding to the calling thread. `errno` being a per-thread value, it is stored in that data structure.

Then, we need to create a new header file for `my_syscall` so that an application wanting to call that function from the C library can include that header. Note that `rename` is already in the standard headers of newlib so there is no need to add anything for that syscall. Edit `newlib/newlib/libc/include/my_syscall.h`:

```
#ifndef MY_SYSCALL_H
#define MY_SYSCALL_H

int my_syscall(int x);

#endif /* MY_SYSCALL_H */
```

Next, we need to edit the file used by the `configure` script to generate newlib makefiles, in order to include the two source files we added. Edit the following line in `newlib/newlib/libc/sys/hermit/Makefile.am`:

```
lib_a_SOURCES = errno.c chown.c environ.c execve.c fork.c getpid.c kill.c lseek.c
    readlink.c stat.c times.c wait.c close.c _exit.c fstat.c gettod.c isatty.c link.
    c open.c read.c sbrk.c symlink.c unlink.c write.c signal.c my_syscall.c rename.c
```

Finally, we need to regenerate the configure script:

```
# in case it's not done already:
$ export PATH=$PATH:/opt/hermit/bin
$ cd newlib/
$ make distclean
$ cd newlib/libc/sys/hermit
$ /usr/local/bin/automake # Make sure to call automake with the full path
# Automake warnings can be ignored
$ cd ../../../../ # Back the the root of newlib sources
$ ./configure --target=x86_64-hermit --prefix=/opt/hermit --disable-shared --
    disable-multilib --enable-lto --enable-newlib-hw-fp --enable-newlib-io-c99-
    formats --enable-newlib-multithreaded
$ make -j`nproc`
$ sudo PATH=$PATH:/opt/hermit/bin make install
```

That's it! The new system calls can now be tested using an example application:

```
#include <stdio.h>
#include <my_syscall.h>

int main(int argc, char** argv)
{
  int res = my_syscall(12);
  /* Should return 54 */
  printf("my_syscall returns: %d\n", res);

  res = rename("./x", "./y");
  /* Should return 0 */
  printf("rename returns %d\n", res);

  return 0;
}
```

Allocating memory for a ramdisk in HermitCore

One possible way to design a solution for the project is to allocate a large buffer of memory and use that buffer as the backing store for the file-system. We can call that buffer a 'ramdisk'.

Inside the HermitCore kernel, `kmalloc` can be used to dynamically allocate some memory. However, it is not functional for large amounts of memory such as the ones needed to allocate a large buffer for the ramdisk.

A solution to that problem is to request a virtual memory area (VMA) from the kernel, a set of physical memory pages, and map the virtual pages in the VMA to the physical pages we obtained. Here is an example on how to do it, by editing `HermitCore/kernel/main.c`:

```
/* ... */

/* size is the requested ramdisk size in bytes */
size_t init_ramdisk(unsigned long size) {
  size_t viraddr, phyaddr, bits;
  uint32_t npages = PAGE_CEIL(size) >> PAGE_BITS;
  int err;

  /* Request a VMA in the address space with the correspondign size */
  viraddr = vma_alloc((npages + 2)*PAGE_SIZE, VMA_READ|VMA_WRITE|VMA_CACHEABLE);
  if(BUILTIN_EXPECT(!viraddr, 0))
    return (size_t)NULL;

  /* Request physical pages */
  phyaddr = get_pages(npages);
  if(BUILTIN_EXPECT(!phyaddr, 0)) {
    vma_free(viraddr, viraddr + (npages+2)*PAGE_SIZE);
    return (size_t)NULL;
  }

  bits = PG_RW | PG_GLOBAL | PG_NX;

  /* Perform the virtual to physical mapping */
  err = page_map(viraddr + PAGE_SIZE, phyaddr, npages, bits);
  if(BUILTIN_EXPECT(err, 0)) {
    vma_free(viraddr, viraddr + (npages+2)*PAGE_SIZE);
    put_pages(phyaddr, npages);
    return (size_t)NULL;
  }

  /* Return a pointer to the beginning of the allocated area */
  return (size_t) (viraddr + PAGE_SIZE);
}
```

```
/* ... */

/* initd is one of the last kernel initialization function, executed right
   before jumping to the application code */
static int initd(void* arg)
{
/* ... */

  // initialize network
  err = init_netifs();

  // initialize ramdisk
#define RAMDISK_SIZE  (1024*1024*1024*3)  /* 3 GB */
  size_t ramdisk = init_ramdisk(RAMDISK_SIZE);
  /* Let's access all memory in the ramdisk to see if all goes well */
  memset(ramdisk, 0x00, RAMDISK_SIZE);

/* ... */
}
/* ... */
```

Debugging a unikernel

Kernel log and verbose mode

It is possible to log some information from the kernel using logging functions such as `LOG_INFO()`,`LOG_ERROR()`, etc. They are used in a similar way as `printf()` and are defined in `HermitCore/include/hermit/logging.h`.

In order to display the kernel log, `proxy` should be invoked with the following environment variable set: `HERMIT_VERBOSE=1`. The log will be displayed once the application exists. Sometimes, a crash can make Qemu hang, in that case pressing 'ctrl+a' then 'x' exits qemu and print the log. Note that this technique works with both Qemu and Uhyve.

Instruction pointer and disassembling

A large portion of kernel errors generates an exception or a page fault. In that case, the kernel log will output a very valuable information: a dump of the registers content when the exception took place. In particular, the value of the instruction pointer (RIP) indicates the instruction that triggered the exception. In case of a page fault, the kernel log also contains the virtual address that caused the fault, as well as the reasons of the fault (page not present, wrong access permission, etc.).

Disassembling the unikernel binary and using the instruction pointer value allows to get a closer look at the code that caused the error:

```
objdump --disassemble <unikernel binary>
```

The virtual addresses are present on the left column. For binaries compiled the with debug symbols (-g compiler flag) source C code can be additionally displayed by objdump using:

```
objdump --source <unikernel binary>
```

Note that a unikernel binary is the result of the linkage of multiple entities: the kernel, the C library, the application code, etc. According to which entity's code needs to be disassembled, the corresponding entity needs to be compiled with debug symbols.

The register dump made on crash with both Qemu and Uhyve, so this debug technique works with both hypervisors.

Qemu debug mode

Setting the environment variable `HERMIT_DEBUG=1` when running `proxy` will have qemu spawns a GDB server which can be connected from a GDB client on the host. This can be used to debug the unikernel by setting up breakpoints, inspecting the values of variables, explore the execution path at runtime, etc.

A slight modification to `proxy` is needed in order to have Qemu pause the unikernel execution on the first guest instruction, giving the user the time to connect a GDB client: this is useful to debug a bug that would trigger early in the boot process. Modify `HermitCore/tools/proxy.c` as follows around line 392:

```
/* ... */
    // add flag to start gdbserver on TCP port 1234
    qemu_argv[i] = "-s";
    qemu_argv[++i] = "-S"; /* add this line */
  }
/* ... */
```

Recompile and reinstall HermitCore, then launch a unikernel with the environment variable set:

```
$ HERMIT_ISLE=qemu HERMIT_DEBUG=1 /opt/hermit/bin/proxy <path to unikernel>
```

Then, in another terminal, connect with a GDB client:

```
$ gdb <path to unikernel>
(gdb) target remote :1234
```

To be able to correctly use GDB, the entities composing the unikernel (application, kernel, C library) should be compiled with debug symbols. An interesting benefit of unikernels is that is is straightforward to debug the application alongside the operating system, as they both live in the same address space without privilege protection.

Note that this process relies on Qemu to create a GDB server, thus it will not work with Uhyve. Moreover, for the debug process to work correctly, it is needed to use Qemu in emulation mode (no KVM).

Using the default version of GDB from Ubuntu repositories, the debug session might fails with the following error: `Remote 'g' packet reply is too long`. In that case, the solution is to download GDB sources and apply a small patch:

```
$ wget https://ftp.gnu.org/gnu/gdb/gdb-8.0.tar.xz
$ tar xf gdb-8.0.tar.xz
$ cd gdb-8.0/
$ wget https://raw.githubusercontent.com/olivierpierre/gdb-remote-patch/master/gdb-
    remote.patch
$ patch -p1 < gdb-remote.patch
$ ./configure
$ make -j`nproc`
$ sudo make install
```