SdP 2022 – OS161 Laboratoratory – 1

The SIS161 – OS161 operating system is installed in a VBOX OSE virtual machine, having Linux Ubuntu 20.04 as host OS. This kit does not exactly correspond to the pre-recorded demos, which refer to an Ubuntu 14 system: however, no substantial differences are present.

The (minor) differences w.r.t. the version used in videos (based on the Ubuntu14 kit) are:
- use of os161 version 2.0.3 instead of 2.0.2 (differences are negligible)
- Ubuntu user is now os161user, instead of pds
- running directory is now os161/root instead of pds-os161/root

The virtual machine (file ubuntu20-os161-2022) is available for download on the course web page (section materiale (Dropbox)). We suggest, for personal use, to install VirtualBox (https://www.virtualbox.org).

Oltre alla macchina virtuale Ubuntu pre-installata, è possibile

- utilizzare un proprio sistema (virtualizzato o no) Linux (Ubuntu o altro) su cui installare direttamente os161 utilizzando il kit kit-per-installazione-su-ubuntu20

- in alternativa utilizzare Docker come riportato in kit-per-docker (https://github.com/marcopalena/polito-os161-docker)

In addition to the pre-installed Ubuntu virtual machine, it is possible to

- use your own system (virtualized or not) Linux (Ubuntu or other) on which to install os161 directly using the kit-for-installation-on-ubuntu20

- alternatively use Docker as reported in kit-per-docker (https://github.com/marcopalena/polito-os161-docker): this choice is probably better on lower performance (CPU/RAM) platforms. The source directory is here **os161/src** instead of **os161/os161-base-2.0.3.**

MAC systems with M1 processor

For systems with M1 processor it is NOT possible to use VBOX, nor to import the pre-installed Ubuntu virtual machine. It is recommended to generate your own Ubuntu20 (or other Linux) system (with Parallel virtualization) and follow the instructions in **kit-for-installation-on-ubuntu20**, or to use the solution with Docker (**kit-for-docker**)

If the virtual machine does not start

First check that the downloaded .ova copy is not corrupt.

Eventually try to change the vbox virtualization options in settings - system

If necessary, try Vbox 6.0.

If you really can't (on some HW platforms in fact it is better to re-start from scratch) go to the same solutions indicated for MAC-M1.

In order to run the virtual machine, **CAREFULLY** follow the steps below

1) Download and install VirtualBox (https://www.virtualbox.org).

2) Download the virtual machine of OS161. Be careful to complete the download: a partially downloaded or corrupted file will likely prevent next steps.

3) Run Virtual Box.

4) Import the virtual machine, using the "*import appliance*" option from the menu.

5) Run the imported virtual machine.

6) To login, use the following account data:

    - USER: os161
    - PASSWORD: os161user

    We have noticed in the past a few cases of problems related to video setup, depending on the host machine and/or the vbox version. In cases of sudden logout (on Ubuntu 20) right after loggin in, you may wish to try an alternative user interface (Gnome) when loggin in. Vbox Guest Additions have already been installed. But if you feel your computer is not well synchronized (e.g. video does not re-shape correctly, problems with meyboard/mouse, etc,), you may wish to re-install the Vbox Guest Additions (the kit synchronizing the guest OS (Ubuntu) to the Host): in order to run guest additions once logged in Ubuntu, run (in priviledged mode) the CD-ROM file VBoxLinuxAdditions.run, or follow instructions available here: virtualbox.org/manual/ch04.html#additions-linux

7) The virtual machine can be exported, at the end of the task, with a dual procedure: option "export appliance" from

OS161 Environment

The os161 operating system is already pre-installed in the **os161/os161-base-2.0.3** directory, while the required SW packages (binutils, gcc compiler, gdb debugger, MIPS emulator, etc…) are installed in the **os161/tools** directory.
BE CAREFUL: ALL DIRECTORIES ARE WRITTEN STARTING FROM $HOME, which is located in /usr/os161user.

Remember that OS161 is executed by a MIPS processor emulator, SYS161.
In order to re-compile OS161, to debug it, or to perform other operations (such as executable file visualization), you must use software intended for the MIPS platform: these tools are re-named with the prefix "**mips-harvard-os161-**": for example, you must use **mips-harvard-os161-gcc**, instead of gcc; **mips-harvard-os161-gdb** instead of gdb, and so on. Executables of these tools are located in **os161/tools/bin**.

The reference web site is http://www.os161.org/.

Another really interesting website is: http://www.student.cs.uwaterloo.ca/~cs350/common/OS161main.html. This is the website of the "cs350" class at the Waterloo University (Canada).

On the desktop of the virtual machine, you can find a link to a browsable source code.
If you want to install OS161 on your PC, you can find proper instructions in the previously suggested websites. On request, we can provide installation scripts for a Linux-Ubuntu OS.

Detailed information on starting and running OS161 can be found, for example, at the following link: Working with OS161

The initial working directory is: **os161/root**.

BE CAREFUL: there are a few main work directories:

- ***os161/os161-base-2.0.3****: it holds source codes, configuration files, object and executable files, of OS161;* *this is the folder to edit and re-compile OS161*
- ***os161/tools****: this is the folder of all tools used for compilation/link, debug and other tasks (customized for the MIPS processor)*
- ***pds-os161/root****: it is the folder used to boot the operating system and to execute user processes;* *this is the folder to runt and test OS161*.

There are two main ways to bootstrap OS161 in the sys161 emulated environment (from a shell, using the "Terminal" application):

- Normal execution:

  ```
  cd $HOME/os161/root
  sys161 kernel
  ```

  The following lines will be printed:

  ```
  sys161: System/161 release 2.0.8, compiled Mar 30 2016 12:38:39

  OS/161 base system version 2.0.3

  Copyright (c) 2000, 2001-2005, 2008-2011, 2013, 2014

  President and Fellows of Harvard College.  All rights reserved.

  Put-your-group-name-here's system version 0 (DUMBVM #1)

  788k physical memory available

  Device probe...

  lamebus0 (system main bus)

  emu0 at lamebus0

  ltrace0 at lamebus0

  ltimer0 at lamebus0

  beep0 at ltimer0

  rtclock0 at ltimer0

  lrandom0 at lamebus0

  random0 at lrandom0
  ```

```
lhd0 at lamebus0

lhd1 at lamebus0

lser0 at lamebus0

con0 at lser0


cpu0: MIPS/161 (System/161 2.x) features 0x0

OS/161 kernel [? for menu]:
```

After these lines, you can execute commands (menu with ?). Some commands (for example those with option ?o) are NOT fully available, since OS161 is NOT a fully complete operating system (students will complete some of the missing parts).

- Execution using a ***debugger***.

WARNING: the program executed by Ubuntu is sys161. Sys161 is an executable in the host machine (for an Intel/AMD cpu, sys161 is already provided and there is NO NEED to debug it).
The command "sys161 kernel" runs "sys161", which is a program acting as a virtual machine with a MIPS processor. For this MIPS virtual machine, "kernel" is an executable file, loaded and executed.
The goal of all these laboratories is to interact with the "kernel" (executed in a MIPS machine, called sys161), and NOT TO interact with sys161 itself.

Avoid to (unsuccessfully try to) debug the sys161 emulator. The MIPS debugger must be executed **after** running sys161. For this purpose, you need two processes: one to execute sys161 (running with the proper option to debug the kernel) and one to execute mips-harvard-os161-gdb (the proper debugger for MIPS architectures), that communicate by means of a socket. It his highly recommended to use two terminal windows (shells).
In the first shell, from the directory *pso-os161/root*, run the command:

```
sys161 -w kernel
```

in the second, from the same directory, run the command:

```
mips-harvard-os161-gdb kernel
(gdb) dir ../../os161/os161-base-2.0.3/kern/compile/DUMBVM
(gdb) target remote unix:.sockets/gdb
```

Be careful to use the same kernel. DUMBVM represents a suitable version, possibly modified, of the OS161 operating system. More in detail, the previous three lines mean, respectively:

1. The MIPS executable to debug.

2. The (compiling) directory, needed to start locating source files: only if interested (but **highly recommended)** to a debug session showing the C source code under analysis. In this situation, it is enough to locate object files, since they hold a link to the corresponding source files.

3. The socket connection needed for the communication between sys161 and gdb.

The creation of a new kernel version will be described in the next section "Modify the kernel". If, after recompiling other kernels (for example kernel-GENERIC, kernel-ASST1, kernel-HELLO, etc…), you want to use a different kernel, be careful to use the same name in both shells.

For simplicity, the two previous commands *dir* and *target* are added into an initialization command file used by gdb (located in *os161/root/.gdbinit*): a new commands is defined, "*dbos161*", and also used by gdb itself (making useless their explicit execution). THEREFORE, IN THE PROVIDED VERSION OF OS161, THE TWO COMMANDS ARE NOT NEEDED IF YOU ARE WORKING WITH DUMBV. IF YOU CHANGE THE KERNEL VERSION, IT IS RECOMMENDED TO CAREFULLY EDIT *os161/root/.gdbinit*, in order to expect other versions. In fact you can edit this file, adding other commands for any further version.

The gbd version previously described refers to the command line version of the debugger (rather uncomfortable). IT IS STRONGLY NOT RECOMMENDED TO USE THIS VERSION. IT IS MUCH BETTER TO USE A GRAPHIC INTERFACE VERSION, DESCRIBED BELOW:

You have several options to execute gdb with a graphic interface:

- gdb with a window showing the source code (it is the simplest choice):

  ```
  mips-harvard-os161-gdb -tui kernel
  ```

- *ddd*, a gui for gdb (it is the suggested choice):

  ```
  ddd --debugger mips-harvard-os161-gdb kernel)
  ```

- `emacs,` a very powerful editor, but with a steep learning curve: in order to open a debugging window, after running emacs ("*emacs*" command), the debugger can be activated in Tools->Debugger, editing the suggested command (bottom row, in the lower part of the windows) with:
  `mips-harvard-os161-gdb -i=mi kernel`

`BE CAREFUL:` if sys161 crashed, you must restart it, and re-connect the debugger (restarting it or simply re-running the "*dbos161"* command). Then, IT IS NOT NECESSARY TO RESTART `mips-harvard-os161-gdb`, BUT ONLY *sys161*. Anyway, from the debugger, you must re-connect the socket, for example with the command "*dbos161*" (or any other equivalent command, eventually added to *os161/root/.gdbinit*),

VSCode

To use VSCode, refer to the "Setup VS Code for OS161.pdf" file, located in os161-kit/Doc_VSCode.zip. VSCode is also available in the Docker-based version, to use which refer to the installation kit.

Modify the kernel

Kernel source are located in the directory os161/os161-base-2.0.3/kern and related sub-directories: in this section this prefix is omitted, then all paths are expressed starting from "kern".

A new kernel version implies to edit and/or to add source files. For a full description, follow this description Bulding OS/161 (sections: "Configure a kernel" and "Compile a kernel").

Every new kernel version corresponds to a configuration file (written uppercase) in the directory: *os161/os161-base-2.0.2/kern/conf*. In the old OS161 versions, configurations used names like ASSTx (x=0,1,2,3,4, …). Instead now the configuration names are: DUMBVM (or DUMBVM-OPT) for the version having a "dumbvm" memory management; GENERIC (or GENERIC-OPT) for a new version, for example the first to start to work on; or other names, depending (and describing) the kind of activity performed in the code.

For the first execution it is suggested to start working on DUMBVM, without any change, then to start creating a new version, called HELLO.
The first task is to add an additional message to the lines printed out during the bootstrap. In order to accomplish this, it is required to add a *hello.c* file in the directory *kern/main*, having a function *hello()*, which will print a message on the screen using the kernel function *kprintf().*

Detailed instructions follow.

Create a file `kern/main/hello.c`

Write into this newly created file a function `hello` which uses `kprintf()` to write a message on screen. Even if not necessary, it is suggested to create a file `kern/include/hello.h`, having the prototype of the *hello* function.
BE CAREFUL: the provided C compiler strictly requires a void parameter if no other parameter is provided.
For example, the hello protype could be:

```
void hello (void);
```

Edit `kern/main/main.c` adding a call to *hello()*. In order to properly use `kprintf()`, you must include `types.h` and `lib.h` libraries.

The optional inclusion of `hello.h` (`#include "hello.h"`) must be added to both `main.c` and `hello.c`.

Edit `kern/conf/conf.kern` adding a new file `hello.c.` in the files list. For example

```
file      main/main.c

file      main/menu.c

defoption hello

optfile   hello   main/hello.c
```

Summing up, in order to call from *main.c* a function placed in *hello.c*, it is necessary (in *main.c*) to have the prototype of this function. This can be done explicitly (writing the prototype itself), or (better solution) including the file `kern/include/hello.h`.

Reconfigure and recompile the system

CONFIGURATION

In *kern/conf* generate the HELLO file (for example copying DUMBVM: PAY ATTENTION, **add to this file a row "options hello"**!) and execute the command

./config HELLO

In order to make `hello.c` and the call to `hello()` to be active/visible only with the proper configuration option "*hello*", the following steps are needed:

- Use the *hello* option, defined in `conf.kern` and make the *hello.c* file optional (enabling that option). As a consequence, it will be automatically generated an *opt-hello.h*, having `#define OPT_HELLO 1` or `#define OPT_HELLO 0`.
- Make optional the instructions using hello:

- The file `hello.h` file

```
#ifndef _HELLO_H_
#define _HELLO_H_

void hello(void);

#endif
```

can have an optional content adding the following lines

```
#ifndef _HELLO_H_
#define _HELLO_H_

#include "opt-hello.h"
#if OPT_HELLO
void hello(void);
#endif

#endif
```

- The *hello()* call from the main function can become optional with the following edits

```
#if OPT_HELLO
   hello();
#endif
```

COMPILATION

In *kern/compile/HELLO* execute the following commands:

```
bmake depend

bmake

bmake install
```

In case of compilation errors, you just need to repeat *bmake.*
Try to run OS161 in order to verify the print of your custom message during bootstrap.

Concurrent programming in OS161

Carry out this exercise with configuration THREADS, instead of DUMBVM or HELLO: it is recommended to make a new configuration only as exercise, even though there is no need (no source file will be edited).

Built-in thread tests

When OS161 starts, among the menu options available, thread tests can be started. These are functions NOT loaded as separate executables programs, but directly linked in the kernel (in fact, they are parts of the kernel itself).
Thread tests make use of synchronization techniques based on semaphores. It is suggested to trace the execution using a debugger, to verify how the scheduler works, how threads are created and what happens during the context switch. In order to achieve this task, it is recommended to trace functions like *thread_create()*, *thread_fork()*, *thread_yield()*, etc…

“tt1” test prints out numbers from 0 to 7 for every thread loop, “tt2” prints only when a thread starts and when it ends (it is useful to verify that the scheduler does not generate starvation). Threads are started and will run for some iterations. “tt3” test uses semaphores, which are not fully implemented in the base version of OS161: this task will be faced in a future laboratory exercise.

The source code starting tests can be found in *menu.c*.

HINT to debug thread-based programs
Since the *thread_yield()* function is called at random time intervals, in order to generate (and debug) repeatable execution sequences, it is suggested to use a seed to force a fixed initialization of the random number generator: look for the “*random*” instruction in *sys161.conf*.
Then use the “autoseed” functionality only when everything works properly.