University of Karlsruhe
System Architecture Group
Gerd Liefländer

Teaching Assistant:
Philipp Kupferschmied

System Architecture 2008/09
Programming Assignment - Theoretical Part 0

Submission Deadline: 07.11.2008, 12:00h

For the programming assignment, you have to form teams of **exactly two** members. Consequently, you have to find a partner with whom you can work (you can use the course forum for that purpose). The team must be registered on the couse website, see http://i30www.ira.uka.de/teaching/courses/os161registration2pe
The login information of the account is needed for the online submission of the theoretical part of each assignment. You also have to provide your group number (`sapwX`) when submitting your solution of the practical part of the assignment via email.
In order to attain the final exam bonus, you need to attain 50 % of the points available for theory and 50 % of the available points for practical assignments. The theoretical parts will be assessed via multiple-choice questions; correct answers account for one point each, incorrect answers subtract one point.
The theoretical part of the assignments has to be turned in about one week after their release—this assignment is due 07.11.2008, 12:00h. Submit your answers on the website ("Programming Assignments" in System Architecture tutorials). Further announcements might be made on the tutorial website, http://i30www.ira.uka.de/teaching/courses/tutorials.php?courseid=176&lid=en, so please watch regularly. The implementation part is to be submitted via email to your assigned tutor; its due date is later than that of the accompanying theoretical part.

1 Introduction

The aim of this assignment is to familiarize you with the environment you will be using for the remaining programming assignments.
Note that we view the code reading as compulsory. You will really struggle with the assignments if you fail to get an understanding of the code base. The code reading component is there to guide you towards acquiring that understanding.
Also note that this assignment is not indicative of the level of difficulty of the later assignments—these will be **much more challenging**!
This assignment will introduce you to the following components of the software suite you will use during the semester:

OS/161, an educational operating system which you will modify to implement the assignments

System/161, the machine simulator that is used to run OS/161

GDB, a powerful debugger that will make your life much easier

Mercurial, a distributed source code management system that will help you manage your code

1.1 OS/161

OS/161 is an educational operating system. It aims to strike a balance between giving students experience in working on a real operating system, and potentially overwhelming students with the complexity that exists in a fully fledged operating system, such as Linux. Compared to most deployed operating systems, OS/161 is quite small (approximately 20,000 lines of code and comments), and therefore it is much easier to develop an understanding of the entire code base, as you will begin to do during this assignment.

The source code distribution contains a full operating system source tree, including the kernel, libraries, various utilities (`ls`, `cat`, ...), and some test programs. OS/161 boots on the simulated machine in the same manner as a real system might boot on real hardware.

1.2 System/161

System/161 simulates a "real" machine to run OS/161 on. The machine features a MIPS R2000/R3000 CPU including an MMU, but no floating point unit or cache. It also features simplified hardware devices hooked up to the *lamebus*. These devices are much simpler than real hardware, and thus make it feasible for you to get your hands dirty without having to deal with the typical level of complexity of physical hardware. Using a simulator has several advantages: Unlike software you have written thus far, buggy OS software may result in completely locking up the machine, making it difficult to debug and requiring a reboot. A simulator enables debuggers to access the machine below the software architecture level as if debugging was built into the CPU. In some senses, the simulator is similar to an *in-circuit emulator* (ICE) that you might find in industry, only it is implemented in software.
The other major advantage is the speed of reboots. Rebooting real hardware takes minutes, and hence the development cycle can be frustratingly slow on real hardware. System/161 boots OS/161 in mere seconds.

1.3 GDB

You should already be familiar with GDB, the GNU debugger. GDB allows you to set *breakpoints* to stop your program under certain conditions, inspect the state of your program when it stops, modify its state, and continue where it left off. It is a powerful aid to the debugging process that is worth investing the time needed to learn it. GDB allows you to quickly find bugs that are very difficult to find with the typical `printf` style debugging.
At http://www.gnu.org/software/gdb/gdb.html you can find details **beyond** the level you need to know for these assignments; a brief and focused introductory tutorial will be provided in the programming part of this assignment.

1.4 Mercurial

Mercurial is a distributed source code management system used to track changes to a piece of software. We make use of Mercurial in this course to allow you to manage the large code base (compared to most previous assignments you have done), and recover from potential problems. Mercurial keeps a recoverable copy of specified versions of all your OS/161 files. It allows you to track the changes you have made and, more importantly, rollback to a known state if things get out of hand. Mercurial also enables you and your partner to work together in a coordinated way. You can even work on versions of your code stored at home and at the university. Mercurial provides lots of features you won't need for the assignments. We will give you step-by-step directions for the parts you need to know. You can find detailed docs on Mercurial at http://www.selenic.com/mercurial/

2 Code Review

This part is assessable, you can **turn in your answers on the System Architecture Tutorials page under "Programming Assignments"**. As the submission is web-based you will have to be on time, no delays allowed. The due date is given at the top of this document. Submissions that are on time are better than perfect but late solutions. You will have to submit 20 questions although there are more questions in this reading part. We recommend that you answer all questions for yourselves nonetheless.
This is probably the first time most of you will attempt to understand and carefully modify a large body of code that you did not write yourself. It is imperative that you take the time to read through the code to get an understanding of its overall structure and of what each part of the code does.

The code reading component of this assignment aims to guide you through the code base to help you comprehend its contents, identify what functionality is implemented where, and be able to make intelligent decisions on how to modify the code base to achieve the goals of the (later) assignments. You do not need to understand every line of code, but you should have a rough idea of the purpose of each file. **Invest the time now** to gain an overall understanding of the code base. Now is probably the **least busy** part of the semester for you. Do not waste it and struggle later.

2.1 The Top-Level Directory

Setup your OS/161 installation as described in the practical part of this assignment before reading on. The `asst0-src` directory is the top-level directory of OS/161. It contains a few files and subdirectories for distinct parts of the OS. The files are:

Makefile: This Makefile builds the OS/161 distribution, including all the provided utilities. It does **not** build the operating system kernel, though.

configure: The configure script prepares the build process—you should not need to understand or tamper with it... do not!

defs.mk: This file is generated by `./configure`. Unless something goes wrong, you should not touch it.

defs.mk.sample: If something does go wrong, you can try to build `defs.mk` manually using this file as a template.

In addition to these files, several subdirectories exist:

bin/ contains the source code for the user-level utilities available on OS/161. They are a subset of the typical Unix `/bin` tools, such as `cat`, `cp`, and `ls`.

include/ contains the include files used to build user-level programs on OS/161. Most importantly, they contain appropriate definitions for using the C library shipped with OS/161. These are **not** the kernel include files.

kern/ contains the sources of the OS/161 kernel itself. We will cover this in more detail later.

lib/ provides user-level library code for the libc.

sbin/ contains the source code for the user-level system management utilities found in `/sbin` on a Unix machine (e.g., `halt` and `reboot`).

testbin/ provides sample applications to test various aspects of OS/161 (e.g., memory management). They are most relevant to the course given at Harvard, but are included here for your perusal and potential use.

Your focus during this code walk through should be on the kernel sources. You will **not** need a detailed understanding of the utilities in `bin/` and `sbin/`. However, some general understanding of how they work and where things are is useful. Similar statements hold for `lib/` and `include/`.

2.2 The `kern/` Subdirectory

This directory and its subdirectories are where most (if not all) of your action will take place. The only file in this directory is a `Makefile`. This `Makefile` only installs various header files; it does not actually build anything. We will now examine the various subdirectories in detail. Take time to explore the code and answer the questions.

2.2.1 `kern/arch/`

This directory contains architecture-dependent code, which means code that is dependent on the architecture OS/161 runs on. Different machine architectures have their own specific architecture-dependent directory. Currently, the only supported architecture is MIPS.

2.2.2 `kern/arch/mips/conf/`

`conf.arch` tells the kernel config script where to find the machine-specific, low-level functions it needs (see `mips/mips/`).
`Makefile.mips` is copied over to the build directory when you configure a kernel.
Question 0.1: What is the VM system called that is configured for assignment 0?

2.2.3 `kern/arch/mips/include/`

Files in this directory are included to obtain machine-specific constants and functions.
Question 0.2: Which register number is used for the stack pointer (sp) in OS/161?
Question 0.3: What bus(es) does OS/161 support?
Question 0.4: What is the difference between `splhigh` and `spl0`?
Question 0.5: Why do we use `typedef`'ed types such as `u_int32_t` instead of simply `int`?
Question 0.6: What must be the first element in the process control block?

2.2.4 `kern/arch/mips/mips/`

Files in here provide machine-dependent low-level functions for the kernel.
Question 0.7: What does `splx` return?
Question 0.8: What is the highest interrupt level?
Question 0.9: Which function is called when user-level code generates a fatal fault?

2.2.5 `kern/compile/`

This is where you build kernels. In the compile directory, you will find one subdirectory for each kernel you want to build. In a real installation, these will often correspond to things like a debug build, a profiling build, ... In our world, each build directory will correspond to a programming assignment, e.g., ASST0. These directories are created when you configure a kernel (described in the next section). This directory and build organization is typical of Unix installations and is not necessarily universal across all operating systems.

2.2.6 `kern/conf/`

`config` is a shell script that takes a config file such as `ASST0`, turns it into a Makefile and some C code, and sets up the build directory. Later (i.e., **not** now), you need to build your kernels using

```
$ cd kern/conf
$ ./config ASST0
$ cd ../compile/ASST0
$ make
```

This will create the `ASST0` build directory and then actually build a kernel in it. **Note that you should specify the complete pathname `./config` when you configure OS/161.** If you omit the "`./`", you may end up running the configuration command for the system on which you are building OS/161, and that is almost guaranteed to produce rather strange results!

2.2.7 `kern/include/`

These are the include files that the kernel needs. The `kern/` subdirectory contains include files that are visible not only to the operating system itself but also to user-level programs.
Question 0.10: How frequently are hard-clock interrupts generated?
Question 0.11: What functions comprise the standard interface to a VFS device?
Question 0.12: How many characters are allowed in a volume name?
Question 0.13: How many direct blocks does an SFS file have?
Question 0.14: What is the standard interface to a file system (i.e., which functions do you have to implement to support a new file system)?

Question 0.15: What function puts a thread to sleep?
Question 0.16: How large are OS/161 PIDs?
Question 0.17: What operations can you do on a `vnode`?
Question 0.18: What is the maximum path length in OS/161?
Question 0.19: What is the system call number for a reboot?
Question 0.20: Where is `STDIN_FILENO` defined?

2.2.8 `kern/main/`

This is where the kernel is initialized and where the kernel main function is implemented.
Question 0.21: What does `kmain()` do?

2.3 `kern/thread/`

Threads are the fundamental abstraction on which the kernel is built.
Question 0.22: Is it OK to initialize the thread system before the scheduler? Why (not)?
Question 0.23: What is a zombie?
Question 0.24: How large is the initial run queue?

2.3.1 `kern/lib/`

These are library routines used throughout the kernel for managing sleep queues, run queues, kernel malloc, ...

2.3.2 `kern/userprog/`

This is where to add code to create and manage user level processes. As it stands now, OS/161 runs only kernel threads; there is no support for user level code (yet).

2.3.3 `kern/vm/`

This directory is also fairly vacant. Virtual memory would be mostly implemented in here.

2.3.4 `kern/fs/vfs/`

This is the file-system independent layer (`vfs` stands for "Virtual File System"). It establishes a framework into which you can easily add new file systems. You will want to review `vfs.h` and `vnode.h` before looking at this directory.
Question 0.25: What does a device name in OS/161 look like?
Question 0.26: What does a raw device name in OS/161 look like?
Question 0.27: What lock protects the `vnode` reference count?
Question 0.28: What device types are currently supported?

2.3.5 `kern/fs/sfs/`

This is the simple file system that OS/161 contains by default. You may augment this file system as part of a future assignment, so we will ask you questions about it then.

2.3.6 `kern/dev/`

This is where all the low level device management code is stored. You can safely ignore most of this directory.