University of Karlsruhe
System Architecture Group
Gerd Liefländer

Teaching Assistant:
Philipp Kupferschmied

System Architecture 2008/09 Programming Assignment 2

Submission Deadlines: 19.12.2008, 12:00h (theoretical part)
16.01.2009, 12:00h (practical part)

This document contains both the theoretical and the practical assignments. You might want to read the entire document once before reading it again to answer the questions. The next step is then to analyze the problems and write a simple design document. Afterwards, as the last step, start coding and submit your solution—on time.

Note: Do not start the advanced part until you have finished the basic part. Please submit the basic part before starting the advanced part. The grading will be 70 % of the coding marks for the basic part and 30 % for the advanced part. The second part is more rewarding and will give real insights into OS internals.

1 Introduction

In this assignment you will implement a set of file- and process-related system calls. Upon completion, your operating system will be able to run multiple copies of a single application at user level and perform basic file I/O. If you attempt the advanced part, you will be able to run multiple applications. A substantial part of this assignment is understanding how OS/161 works and determining what code is required to implement the required functionality. Expect to spend at least as long browsing and digesting OS/161 code as actually writing and debugging your own code.

Your current OS/161 system has minimal support for running executables, nothing that could be considered a true process. This assignment starts the transformation of OS/161 into something closer to a true multi-programming (or multi-tasking) operating system. After this assignment, it will be capable of running multiple processes from actual compiled programs stored in your account. The programs will be loaded into OS/161 and executed in user mode by System/161; this will occur under the control of your kernel. First, however, you must implement part of the interface between user mode programs ("userland") and the kernel. As usual, we provide part of the code you will need. Your job is to design and build the missing pieces.

Our code can run one user-level C program at a time as long as it does not want to do anything but shut the system down. We have provided sample applications that do this (`reboot`, `halt`, `poweroff`) as well as others that make use of features you might be adding in this and future assignments.

So far, all the code you have written for OS/161 has only been run within, and only been used by, the operating system kernel. In a real operating system, the kernel's main function is to provide support for user-level programs. Most such support is accessed via *system calls*. We give you one system call, `reboot()`, which is implemented in the function `sys_reboot()` in `main.c`. If you place a breakpoint on `sys_reboot` in GDB and run the `reboot` program, you can use GDB's `backtrace` to see how it got there.

For the advanced part of this assignment, you will add a subsystem that keeps track of multiple tasks. You must decide what data structures you will need to hold data pertinent to a *process* (Hint: Look at kernel includes for your favorite operating system for suggestions, specifically the `proc` structure). The first step is to read and understand the parts of the system that we have written for you.

1.1 User-Level Programs

The MIPS simulator System/161 can run normal C programs if they are compiled using a cross-compiler, such as `cs161-gcc`. The latter runs on a host (e.g., a Linux x86 machine) and generates

MIPS executables; it is the same compiler also used to compile the OS/161 kernel. To create new user programs, you will need to edit the `Makefile` in `bin/`, `sbin/`, or `testbin/` (depending on where you put your program) and then create a directory similar to those that already exist. Use an existing program and its `Makefile` as a template.

1.2 Design

In the beginning, you should tackle this assignment by producing a *design document*. This document should clearly reflect the development of your solution, not merely explain what you programmed. If you try to code first and design later, or even if you design hastily and rush into coding, you will most certainly end up in a software "tar pit". Do not try! Plan everything you will do. Do not even think about coding until you can precisely explain to your partners what problems you need to solve and how the pieces relate to each other.
Note that it can often be hard to write (or talk) about a new software design, you are facing problems that you have not seen before, and therefore even finding terminology to describe your ideas can be difficult. There is no magic solution to this problem, but it gets easier with practice. The important thing is to go ahead and try. Always try to describe your ideas and designs to someone else. In order to reach an understanding, you may have to invent terminology and notation, this is fine. If you do this, by the time you have completed your design, you will find that you have the ability to efficiently discuss problems that you have never seen before. Why do you think that CS is filled with so much jargon?
To help you get started, we have provided the following questions as a guide for reading through the code. Once you have prepared the answers, you should be ready to develop a strategy for designing your code for this assignment.

2 Code Walk-Through

Before answering the following questions, you should obtain a copy of the source code, as described in Section 3. Additionally, you might want to have a look at tools such as *Cscope* or *Ctags*, which allow for an easier navigation within the source code (e.g., to find the global definition of a specific function, to find functions calling a specific function, and so on).

2.1 `kern/userprog/`

This directory contains the files that are responsible for loading and running user-level programs. Currently, the only files in the directory are `loadelf.c`, `runprogram.c`, and `uio.c`, although you may want to add more of your own during this assignment. Understanding these files is the key to getting started with the assignment, especially the second part, the implementation of multi-programming. Note that you will have to look into files outside this directory to answer some of the questions.

2.1.1 `loadelf.c`

This file contains the functions responsible for loading an ELF executable from the file system and into virtual memory space. Of course, at this point, the virtual memory space does not provide what is normally meant by virtual memory. Although there is a translation between the addresses that executables *believe* they are using and physical addresses, there is no mechanism for providing more memory than physically exists.

2.1.2 `runprogram.c`

This file contains only one function, `runprogram()`, which is responsible for running a program from the kernel menu. Once you have designed your file system calls, a program started by `runprogram()` should have the standard file descriptors (`stdout` and `stderr`) available while it is running.
In the second part, `runprogram()` is a good base for writing the `execv()` system call, but only a base. When writing your design doc, you should determine what more is required for `execv()` that

`runprogram()` does not need to worry about. Once you have designed your process framework, `runprogram()` should be altered to start processes properly within this framework.

2.1.3 `uio.c`

This file contains functions for moving data between kernel and user space. Knowing when and how to cross this boundary is critical to properly implementing user-level programs, so this is a good file to read very carefully. You should also examine the code in `lib/copyinout.c`.

Question 2.1: What are the ELF magic numbers?

Question 2.2: What is the difference between `UIO_USERISPACE` and `UIO_USERSPACE`? When should one use `UIO_SYSSPACE` instead?

Question 2.3: In `runprogram()`, why is it important to call `vfs_close()` before going to user mode?

Question 2.4: What function forces the processor to switch to user mode?

Question 2.5: Is this function machine dependent?

Question 2.6: In what file are `copyin()` and `copyout()` defined? Where is `memmove()` defined? Why are `copyin()` and `copyout()` not implemented as simply as `memmove()`?

Question 2.7: What (briefly) is the purpose of `userptr_t`?

2.2 `kern/arch/mips/mips`: traps and syscalls

Exceptions are one key to operating systems; they are one of the mechanisms that enable the OS to regain control of execution and therefore do its job. You can think of exceptions as the interface between the processor and the operating system.

When the OS boots, it installs an *exception handler*, a carefully crafted piece of assembly code, at a specific address in memory. When the processor raises an exception, it invokes this handler, which sets up a *trap frame* on the stack and calls into the operating system.

This mechanism is used to handle three types of events:

- exceptions (error conditions triggered by the currently executing instruction)
- interrupts (notifications of events outside the processor code, e.g., from I/O devices)
- traps (or system calls (syscalls) or *software interrupts*; an explicit request to execute kernel code, think of this as calling a kernel function from user mode)

Specifically, `syscall.c` handles all system calls. Understanding at least the C code in this directory is key to becoming a real operating systems junkie, so we highly recommend reading through it carefully.

2.2.1 `trap.c`

`mips_trap()` is the central function for returning control to the operating system. This C function is called by the assembly exception handler.

`md_usermode()` is its complement; it returns control to user-level programs.

`kill_curthread()` handles broken/malicious/invalid user-level programs. Whenever the processor is in user mode and encounters an exceptional condition (e.g., an invalid instruction), it raises an exception. If there is no way to recover properly, the OS needs to terminate the process. The advanced part of this assignment will include writing a useful version of this function.

2.2.2 `syscall.c`

`mips_syscall()` delegates the actual work of a system call to the kernel function that implements it. Notice that `reboot` is the only case currently handled. You will also find a function `md_forkentry()`, which is a stub for your code implementing the fork system call.

Question 2.8: What is the numerical value of the exception code for a MIPS system call?

Question 2.9: Why does `mips_trap()` set `curspl` to `SPL_HIGH` "manually" instead of using `splhigh()`?

Question 2.10: How many bytes make up a MIPS instruction? (Answer this by reading `mips_syscall()` carefully, not by looking somewhere else.)

Question 2.11: What is the contents of the `struct trapframe`?
Question 2.12: Where is the `struct trapframe`, which is passed on to `mips_syscall()`, stored?
Question 2.13: What would be required to implement a system call that takes more than 4 arguments?

2.3 `lib/crt0`: the C startup code

The only file in here, `mips-crt0.S`, contains the MIPS assembly code that receives control first when a user-level program is started. Most importantly, it calls `main()`.
This is the code that your `execv()` implementation will be interfacing to, so be sure to at least check which values it expects to appear in which registers!

2.4 `lib/libc`: the user-level side

There is obviously a lot of code in the OS/161 C library (and a lot more yet in a real system's C library...). We do not expect you to read all of it, although it may be instructive in the long run—job interviewers have an uncanny habit of asking people to implement simple standard C functions on the whiteboard. For present purposes you need only look at the code that implements the user-level side of system calls.

2.4.1 `errno.c`

This is where the global variable `errno` is defined.

2.4.2 `syscalls-mips.S`

This file contains the machine-dependent code necessary for implementing the user-level side of MIPS system calls, so it might be important to you.

2.4.3 `syscalls.S`

This file is automatically created from `syscalls-mips.S` at compile time and is the actual file to be assembled and put into the C library. The actual names of the system calls are placed in this file using a script called `callno-parse.sh`, which reads them from the kernel's header files. This avoids having to make a second list of the system calls. In a real system, typically each system call stub is placed in its own source file to allow selectively linking them in. OS/161 puts them all together to simplify the Makefiles.
Question 2.14: What is the purpose of the `SYSCALL` macro?
Question 2.15: What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not by looking somewhere else.)

2.5 `kern/include`

The files `vfs.h` and `vnode.h` in this directory contain function declarations and comments that are directly relevant to this assignment.
Question 2.16: How are `vfs_open()` and `vfs_close()` used? What other `vfs_*()` calls are relevant?
Question 2.17: What are `VOP_READ`, `VOP_WRITE`? How are they used?
Question 2.18: What does `VOP_TRYSEEK` do?
Question 2.19: Where is the `struct thread` defined? What does this structure contain?

2.6 `fork()`

Answer these questions by reading the `fork()` documentation in `~/sysarch/root/man/syscall/fork.html` and the sections on `fork()` in the textbook.
Question 2.20: What is the purpose of the `fork()` system call?
Question 2.21: What process state is *shared* between the parent and the child?

Question 2.22: What process state is *copied* from parent to child?

3 Setting up the Assignment

In this section, you will create a new repository for the source code of this assignment.

3.1 Obtaining and setting up ASST2 in Mercurial

Only one of you must to do the following. Again, *s_hghost* has to perform the following commands.

```
$ mkdir -p ~/sysarch/sharedrepos/asst2-src
$ cd ~/sysarch/sharedrepos/asst2-src
$ hg init
$ cd ~/sysarch/
$ hg clone sharedrepos/asst2-src
$ wget http://i30www.ira.uka.de/~pkupfer/edu/2008ws/sysarch/asst2-src.tbz2
$ tar -xjf asst2-src.tbz2
$ rm asst2-src.tbz2
$ cd ~/sysarch/asst2-src
$ hg add
$ hg commit
$ hg tag asst-base
$ hg push
```

3.2 Obtaining a Working Copy

s_hghost has already cloned the shared repository during the setup process described above. *s_member* can clone it via

```
$ mkdir -p ~/sysarch
$ cd ~/sysarch
$ hg clone ssh://sysarch_hg/asst2-src
```

You are now ready to start with this assignment.

3.3 Building ASST2

Before proceeding any further, configure the sources and build the user land parts of OS/161 via

```
$ cd ~/sysarch/asst2-src
$ ./configure --ostree="$HOME/sysarch/root"
$ make
```

Note: `make` builds and installs the user-level applications shipped with OS/161. In the prior assignment, these were not required, hence we did not tell you to build them.
For your kernel development, we have provided you with another framework for you to run your solutions for ASST2. You have to configure your kernel before you can use this framework. The procedure for configuring and building a kernel is the same as in ASST0 and ASST1:

```
$ cd ~/sysarch/asst2-src/kern/conf
$ ./config ASST2
$ cd ../compile/ASST2
$ make
```

If you are told that the `compile/ASST2` directory does not exist, make sure you ran config for ASST2.

3.4 Command Line Arguments to OS/161

Your solutions to ASST2 will be tested by running OS/161 with command line arguments that correspond to the menu options in the OS/161 boot menu.
IMPORTANT: DO NOT change these menu option strings!

3.5 Running ASST2

For this assignment, we have supplied a user-level OS/161 program that you can use for testing. It is called `asst2`, and its sources live in `testbin/asst2`. You can test your assignment by typing `p /testbin/asst2` at the OS/161 menu prompt or by using command line arguments to System/161: `./sys161 kernel ‘‘p /testbin/asst2’’`
Note: On cygwin, you need to type `p /testbin/asst2.exe`.
Running the program produces output similar to the following prior to starting the assignment:

```
Unknown syscall 6
Unknown syscall 6
Unknown syscall 6
Unknown syscall 6
Unknown syscall 6
Unknown syscall 6
Unknown syscall 0
```

A completed `asst2` should produce the following output:

```
OS/161 kernel [? for menu]: p /testbin/asst2
Operation took 0.000212160 seconds
OS/161 kernel [? for menu]:
**********
* File Tester
**********
* write() works for stdout
**********
* write() works for stderr
**********
* opening new file "test.file"
* open() got fd 3
* writing test string
* wrote 45 bytes
* writing test string again
* wrote 45 bytes
* closing file
**********
* opening old file "test.file"
* open() got fd 3
* reading entire file into buffer
* attempting read of 500 bytes
* read 90 bytes
* attempting read of 410 bytes
* read 0 bytes
* reading complete
* file content okay
**********
* testing lseek
* reading 10 bytes of file into buffer
* attempting read of 10 bytes
* read 10 bytes
* reading complete
* file lseek okay
* closing file
**********
* testing fork
* Forked, in parent
* Forked, in child
Unknown syscall 0
```

4 Basic Assignment: File System Calls

In the basic part of this assignment you will implement a number of (file-related) system calls. The full range of system calls that we think you might want over the course of the semester is listed in `kern/include/kern/callno.h`. For now you should implement: `open`, `read`, `write`, `lseek`, `close`, and `dup2`. You should also implement the `fork` system call.

Note: You are implementing the kernel code that implements the system call functionality **within** the kernel. The C stubs that user-level applications call to invoke the system calls are already automatically generated when you build OS/161. If you find that you are having trouble with the second part of the assignment, you can still get a good mark by concentrating on the file-system related system calls. However, note that even if you do not implement `fork`, your implementation should not assume a single process.

It is crucial that your syscalls handle all error conditions gracefully (i.e., without crashing OS/161). You should consult the OS/161 documentation (in `~/sysarch/root/man/`) and understand fully the system calls that you must implement. You **must** return the error codes as described in the docs. Additionally, your syscalls must return the correct value (in case of success) or error code (in case of failure) as specified in the docs. Some of the auto-marking scripts rely on the return of appropriate error codes; adherence to the guidelines is as important as the correctness of the implementation.

The file `include/unistd.h` contains the user-level interface definition of the system calls that you will be writing for OS/161 (including ones you will implement in later assignments). This interface is different from that of the kernel functions that you will define to implement these calls! You need to design the latter interface and put it in `kern/include/syscall.h`. As you discovered (ideally) in assignment 0, the integer codes for the calls are defined in `kern/include/kern/callno.h`. You need to think about a variety of issues associated with implementing system calls. Perhaps, the most obvious one is: Can two different user-level processes (or user-level threads, if you choose to implement them) find themselves running a system call at the same time?

4.1 `open()`, `read()`, `write()`, `lseek()`, `close()`, and `dup2()`

For any given process, the first file descriptors (0, 1, and 2) are considered to be standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) respectively. For this basic assignment, the file descriptors 1 (`stdout`) and 2 (`stderr`) must start out attached to the console device (`con:`). You will probably modify `runprogram()` to achieve this. Your implementation must allow programs to use `dup2()` to change `stdin`, `stdout`, and `stderr` to point elsewhere.

Although these system calls may seem to be tied to the file**system**, they are really about manipulating file **descriptors**, or process-specific file system state. A large part of this assignment is designing and implementing a system to track this state. Some of this information (such as the `cwd`) is specific only to the process, but others (such as `offset`) is specific to the process and file descriptor. Do not rush the design. Think carefully about the state you need to maintain, how to organize it, and when and how it has to change.

While this assignment requires you to implement file system-related system calls, you actually have to write virtually no low-level file system code in this assignment. You will use the existing VFS layer to do most of the work. Your job is to construct the subsystem that implements the interface expected by user-level programs by invoking the appropriate VFS and `vnode` operations. While you are not restricted to only modifying these files, please place most of your function prototypes and data types for your file subsystem in `src/kern/include/file.h`, and place most of the functions' implementations and variable instantiations in `src/kern/userprog/file.c`.

4.2 `fork()`

For the basic assignment, you will implement a simplified form of the `fork()` system call. Your implementation of `fork` should be the same as that described in the OS/161 docs, except that it should return 1 to the parent (rather than the child's PID). The amount of code to implement fork is quite small; the main challenge is to understand what needs to be done. We strongly encourage you to implement the file-related system calls first, with `fork` in mind.
Some hints:

- Read the comments above `mips_usermode()` in `kern/arch/mips/mips/trap.c`.
- Read the comments in `kern/include/addrspace.h`, particularly `as_copy()`.
- You will need to copy the trapframe from the parent to the child. You should be careful how you do this, as there is a possible race condition (where?/why?).
- You may wish to base your implementation on the `thread_fork()` function in `kern/thread/thread.c`.

4.3 A Note on Errors and Error Handling

The syscall documentation in the OS/161 distribution contains a description of the error return values that you must implement. If there are conditions that can happen, but are not listed in the docs, return the most appropriate error code from `kern/errno.h`. If none seem particularly appropriate, consider adding a new one. If you are adding an error code for a condition for which Unix has a standard error code symbol, use the same symbol if possible. If not, feel free to make up your own, but note that error codes should always begin with `E` and should not be `EOF`. Consult Unix man pages to learn about Unix error codes. Note that if you add an error code to `src/kern/include/kern/errno.h`, you need to add a corresponding error message to the file `src/lib/libc/strerror.c` as well.

4.4 Design Questions

Here are some additional questions and issues to aid you in developing your design. They are by no means comprehensive, but they are a reasonable place to start developing your solution.

- What primitive operations exist to support the transfer of data to and from kernel space? Do you want to implement more on top of these?
- You will need to "bullet-proof" the OS/161 kernel from user program errors. There should be nothing a user program can do to crash the operating system when invoking the file system calls. It is okay in the basic assignment for the kernel to `panic()` for an unimplemented system call (e.g. `execv()`), or a user-level program error.
- Decide which functions you need to change and which structures you may need to create to implement the system calls.
- How you will keep track of open files? For which system calls is this useful?

For additional background, consult one or more of [1, 2, 3, 4] for details how similar existing operating systems structure their file system management.

4.5 Documenting your Solution

This is a compulsory aspect of this assignment. You must submit a small design document identifying the basic issues in this assignment, and then describe your solution to the problems you have identified. The design document you developed in the planning phase (outlined above) would be an ideal start. The document must be plain ASCII text. We expect such a document to be roughly 500–1000 words, i.e., clear and to the point The document will be used to guide our markers in their evaluation of your solution to the assignment In the case of a poor results in the functional testing combined with a poor design document, we will base our assessment on these components alone. If you cannot describe your own solution clearly, you cannot expect us to reverse engineer the code to a poor and complex solution to the assignment.
Create your design document at the top of the source tree to OS/161 (i.e., in `~/sysarch/asst2-src`), and include it in the Mercurial repository as follows.

```
$ cd ~/sysarch/asst2-src
$ hg add design.txt
```

When you later commit your changes into your repository, your design document will be included in the commit, and later in your submission. Also, please word wrap your design doc if your have not already done so. You can use GNU `fmt` to achieve this if your editor does not.

4.6 Basic Assignment Submission

As with the previous assignments, you will again submit a diff of your changes to the original tree. First, both team members need to commit their latest local changes, and push these changesets back to the shared repository, using

```
$ cd ~/sysarch/asst2-src
$ hg commit
$ hg push
```

You might have to manually resolve conflicts.
The one who is going to create the diff and submit it should verify that his/her local repository is up-to-date

```
$ cd ~/sysarch/asst2-src
$ hg pull
$ hg update
```

If everything is all right, the latest revision can be tagged, and the diff can be created:

```
$ cd ~/sysarch/asst2-src
$ hg tag asst-final
$ hg diff -r asst-base -r asst-final > ~/asst2.diff
```

Send the diff (`~/asst2.diff`) of your solutions to `os161@ira.uka.de`. Your subject line should contain your group name (*sapwX*) and the number of the assignment. (Please ensure that you have no whitespace between *sapw* and your group number).
Note: If for some reason you need to change and re-submit your assignment after you have tagged it "asst-final", you will need to either delete the "asst-final" tag, commit the new changes, re-tag, and re-diff your assignment, or choose a different final tag name and commit the new changes, tag with the new tag, and re-diff with the new tag. To delete a tag, use `hg tag --remove` *`tagname`*
Even though the generated diff output should represent all the changes you have made to the supplied code, occasionally students do something "ingenious" and generate non representative diff output. We strongly suggest to keep your Mercurial repository intact to allow for recovery of your work if need be.

5 Advanced Assignment

This is the second part of the assignment. Given you are doing the advanced version of the assignment, we assume that you are competent with managing your Mercurial repository and do not need simple directions. Basically, at the end of the assignment you will need to generate a `diff` between your final version and "asst2-base". It is up to you how you get there. Two obvious options are (1) to continue developing along your mainline repository or (2) to create another clone of your repository.

5.1 New System Call: `uptime()`

This syscall returns the number of seconds that have passed since the kernel was started. This is usually not a real system call, though similar ones exist. Unlike the other calls you have implemented so far, this one does not yet exist in the OS/161 framework—will need to add support for it to the user-level side as well as all the relevant kernel code.

5.2 User-Level Process Management System Calls

5.2.1 `getpid()`

A PID, or process ID, is a unique number that identifies a process. The implementation of `getpid()` is not terribly challenging, but PID allocation and reclamation are the important concepts that you must implement. It is not OK for your system to crash because, over the lifetime of its execution, you have used up all the PIDs. Design your PID system; implement all the tasks associated with PID maintenance, and only then implement `getpid()`.

5.2.2 `execv()`, `waitpid()`, `_exit()`

These system calls are probably the most difficult part of the whole assignment, but also the most rewarding. They enable multi-programming and make OS/161 a much more useful entity.
`fork()` is your mechanism for creating new processes. It should make a copy of the invoking process and make sure that the parent and child processes each observe the correct return value (i.e., 0 for the child and the newly created PID for the parent). You will want to think carefully through the design of `fork()` and consider it together with `execv()` to make sure that each system call is performing the correct functionality.
`execv()`, although "only" a system call, is really the heart of this assignment. It is responsible for taking newly created processes and make them execute something useful (i.e., something different from what the parent is executing). Essentially, it must replace the existing address space with a brand new one for the new executable (created by calling `as_create()` in the current `dumbvm` system) and then run it. While this is similar to starting a process straight out of the kernel (like `runprogram()` does), it is not quite that simple. Remember that this call is coming out of user space, into the kernel, and then returning back to user space. You must manage the memory that travels across these boundaries very carefully. Also, notice that `runprogram()` does not take an argument vector, but these must of course be handled correctly in `execv()`.
Although it may seem simple at first, `waitpid()` requires a fair bit of design. Read the specification carefully to understand the semantics and consider these semantics from the ground up in your design. You may also wish to consult the Unix man page, though keep in mind that you are not required to implement all the features Unix `waitpid()` supports, nor is the Unix parent/child model of waiting the only valid or viable option.
The implementation of `_exit()` is intimately connected to the implementation of `waitpid()`. They are essentially two halves of the same mechanism. Most of the time, the code for `_exit()` will be simple whereas the code for `waitpid()` will be relatively complicated, but it is perfectly legal to design it the other way round as well. If you find that both functions are becoming extremely complicated, it may be a sign that you should rethink your design.

5.2.3 kill_curthread()

Feel free to write `kill_curthread()` in as simple a manner as possible. Just keep in mind that essentially nothing about the current thread's user space state can be trusted if it has suffered a fatal exception—it must be taken off the processor in as judicious a manner as possible, without ever returning execution to user mode (for this process).

5.2.4 Shell

A shell is a user program that reads in user commands and calls on the kernel to execute them. When you log into your ATIS account, you are probably running a shell called `tcsh`.

Write A Simple Shell Your OS/161 shell should provide a prompt (e.g., "`wallaby> `") and accept user commands. Each command and its argument list (an array of character pointers) should be passed to the `execv()` system call, but only after calling `fork()` to get a new thread (probably in a new process) of execution. The argument list, argv, should contain the command name as its first string, `argv[0]`. Once the kernel has finished executing the command, your shell should provide another prompt and wait for input. You should be able to exit the shell by entering `exit`.
Your shell should also support the basic job control function "&", which executes a process in the background: A prompt appears, allowing the user to enter another command, even before the previous command has finished executing. Your shell should not waste an excessive amount of space remembering exit codes if you run a lot of jobs in the background. A skeleton for your shell is in `bin/sh/sh.c`. As the OS/161 system calls needed by your shell are remarkably similar to those supported by Linux, you should be able to write your shell and test it by compiling it for Linux. The supplied `Makefile` already does this. This way, you can write your shell before the rest of this assignment is finished. Keep in mind that this is the same shell you will use later as the user interface for OS/161; it must run as a user program and leave sufficient space for other user programs to run. Consequently, be careful to keep your shell lean. Avoid adding the ultimate X-interface you have always wanted to have; however, it must include the built-in commands `cd` and `exit`.

Test Your Shell Under OS/161, once you have the system calls for this assignment, you should be able to use the following user programs from the bin directory: `cat`, `cp`, `false`, `pwd`, and `true`. You may also find some of the programs in the `testbin/` directory helpful. Under Linux, you should be able to run anything as long as you do not try to use wildcards, fancy shell quoting, pipes, etc.
Note: You may co-develop or share your shell with anyone you wish. The shell is not an assessable component of the advanced assignment, but is required to test your system.

5.3 Design Questions

Here are some additional questions and thoughts to aid in writing your design document. They are not, by any means, meant to be a comprehensive list of all the issues you will want to consider.
Your system must allow user programs to receive arguments from the command line. For example, by the end of this assignment, your shell should be capable of executing user programs containing code fragments such as:

```
char *filename = "/bin/cp";
char *argv[4];
pid_t pid;

argv[0] = "cp";
argv[1] = "file1";
argv[2] = "file2";
argv[3] = NULL;
pid = fork();
if (pid == 0) execv(filename, argv);
```

This fragments should load the executable file `cp`, install it as a new process, and execute it. The new process will then find `file1` on the disk and copy it to `file2`.
Some questions to think about:
Passing arguments from one user program through the kernel into another user program is a bit of a chore. What form does this take in C? This is rather tricky, and there are many ways to be led astray. You will probably find that very detailed pictures and several walk-throughs will be most helpful. How will you determine: (a) the initial value of the stack pointer; (b) the initial content of registers; (c) the return value; (d) whether you can `exec` the program at all?
What new data structures will you need to manage multiple processes? What relationships do these new structures have with the rest of the system?
How will you manage file accesses? When your shell invokes the `cat` command, and the `cat` command starts to read `file1`, what *will* happen if the shell also tries to read `file1`? What would you *like* to happen?
How you will keep track of running processes? For which system calls is this useful?
How you will implement the `execv()` system call? How is the argument passing in this function different from that of other system calls?
What functionality should your shell have?

5.4 Advanced Assignment Submission

Submission for the advanced assignment is similar to the basic assignment, just make sure that your email states clearly this is the *advanced* part. Again, you must commit your latest changes, ensure that you local repository is up-to-data, and generate a diff based on the original source tree:

```
$ cd ~/sysarch/asst2-src
$ hg tag asst-adv-final
$ hg diff -r asst-base -r asst-adv-final > ~/asst2-adv.diff
```

Send the diff (`~/asst2-adv.diff`) of your solutions via email to `os161@ira.uka.de`. Again, the subject line should contain group name and assignment number, but you should also place a hint that you are submitting the advanced part of the assignment.

Note that you have to submit the basic part of the assignment in any case, even if you also implemented the advanced part. Submit the advanced part separately **after** you have submitted the basic part. Also note that submission of the advanced part requires a diff between your advanced solution and the initial revision (i.e., the sources as they are contained in the tarball), **not** between your advanced solution and your basic solution.

References

[1] Andrew S. Tanenbaum. *Modern Operating Systems*, chapter 10.6.3f. Prentice Hall, 2001.

[2] Marshall K. McKusick, Keith Bostic, and Michael J. Karels. *The Design and Implementation of the 4.4 BSD Operating System*, chapter 6.4f. Addison Wesley, 1996.

[3] Uresh Vahalia. *Unix Internals: The New Frontiers*, chapter 8. Prentice Hall, 1995.

[4] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer Conference Proceedings*, pages 238–247, 1986.