System and Device Programming Projects

Operating System Part (Prof. Cabodi)

A.Y. 2022-2023

All standard rules are common with the projects of Prof. Quer (please see Prof. Quer's document for details). The following are specific rules for OS161 projects.
For all projects, the degree of freedom in the choice of what to implement, how to implement and test it, can be high.
Each group will be allowed to submit a preliminary/intermediate plan (a one to three pages document) briefly describing the planned task (high level description of data structures and functions to implement, strategies followed, etc.). Based on the submitted plan, we will agree on more detailed specifications and possibly provide additional support (e.g. total or partial implementation of some required syscalls).
All projects are identified by a code (e.g. project 1: PAGING). Let's use here the generic term CODE. All code changes for this assignment should be enabled by #if OPT_CODE statements. For this to work, you must add #include "opt-code.h" at the top of any file (or an included .h file) for which you make changes for this assignment.
Though it is clear that project c1, on memory management, addresses a topic that is much more familiar to students in early May (when this document is published), the subjects/contents/abilities needed by c2 and c3 will be available over the next few weeks.

Constraints

Groups can be of 2 or 3 students. We have no limit on the number of groups for a given project. Group specific decisions can be agreed after the confirmation of the assignment and/or submission of a preliminary/intermediate report/plan.
A group can ask for a simplified/reduced project, which means that we can agree on a relaxed/reduced set of requirements. The simplified project needs less working time (about half of the standard project) and leads to a maximum of 3 points (instead of 6). In order to ask for a reduced project, please write me on slack, channel os161_projects, after submitting your choices.

Evaluation criteria

The project will be evaluated based on: (a) a short report describing the work done, (b) the source code of the kernel, (c) an oral interview with all group members.
A clear workload organization within the group, with each group member responsible of a given sub-problem (a subtask, a module, a data structure) will be a plus in term of evaluation. So this (a good/clear team organization) is better, for instance, than "we always worked together on a shared desktop". A well-defined set of tests, aimed at showing that the implementation is correct (so not just running "p testbin/palin") is another important aspect to be considered: this means running existing user programs and/or kernel commands as well as modifying them.

Contacts

Prof. Gianpiero Cabodi.

Project C1: Virtual Memory with Demand Paging (PAGING)

Project's summary

The project aims at expanding the memory management (dumbvm) module, by fully replacing it with a more powerful virtual memory manager based on process page tables. The project also requires working on the TLB.

Required Background and Working Environment

Lab 2 and basic knowledge of dumbvm are needed (see lessons os161-overview and os161-memory). Silberschatz chapters 9 and 10 are also a pre-requisite.

Problem Definition

The project goal is to replace dumbvm with a new virtual-memory system that relaxes some (not all) of dumbvm's limitations.
The new system will implement demand paging (with a page table) with page replacement, according to the following requirements:

- New TLB support is needed, by implementing a replacement policy for the TLB, so that the kernel will not crash if the TLB fills up.
- On-demand loading of pages: this will allow programs that have address spaces larger than physical memory to run, provided that they do not touch more pages than will fit in physical memory.
- In addition, page replacement (based on victim selection) is needed, so that a new frame can be found when no more free frames are available.
- Different page table policies can be implemented: e.g. per process page table or Inverted PT, victim selection policies, free frame management, etc. The choice can be discussed and deferred to a later moment.

TLB Management

In the System/161 machine, each TLB entry includes a 20-bit virtual page number and a 20-bit physical-page number as well as the following five fields:

- global (1 bit): If set, ignore the pid bits in the TLB.
- valid (1 bit): When the valid bit is set, the TLB entry is supposed to contain a valid translation. This implies that the virtual page is present in physical memory. A TLB miss exception (EX TLBL or EX TLBS) occurs when no valid TLB entry that maps the required virtual page is present in the TLB.
- dirty (1 bit): In class, we used the term "dirty bit" (or "modify bit") to refer to a bit that is set by the MMU to indicate that a page has been modified. OS/161's "dirty" bit is not like this—it indicates whether it is possible to modify a particular page. OS/161 can clear this bit to indicate

that a page is read-only, and to force the MMU to generate an EX MOD exception if there is an attempt to write to the page.

- nocache (1 bit): Unused in System/161. In a real processor, indicates that the hardware cache will be disabled when accessing this page.
- pid (6 bits): A context or address-space ID that can be used to allow entries to remain in the TLB after a context switch.

In OS/161, the global and pid fields are unused. This means that all of the valid entries in the TLB should describe pages in the address space of the currently running process and the contents of the TLB should be invalidated when there is a context switch. In the dumbvm system, TLB invalidation is accomplished by the as activate function, which invalidates all of the TLB entries. OS/161's context-switch code calls as activate after every context switch (as long as the newly running thread has an address space). If you preserve this functionality in as activate in your new VM system, you will have taken care of TLB invalidation.

For this project, you are expected to write code to manage the TLB. When a TLB miss occurs, OS/161's exception handler should load an appropriate entry into the TLB. If there is free space in the TLB, the new entry should go into free space. Otherwise, OS/161 should choose a TLB entry to evict and evict it to make room for the new entry. As described above, OS/161 should also take care to ensure that all TLB entries refer to the currently running process.

Round-Robin TLB Replacement

You must implement a very simple (but dumb) round-robin TLB replacement policy. This is like first-in- first-out except that we do not actually worry about when each page was replaced and the algorithm works as follows:

```
int tlb_get_rr_victim(void) {
  int victim;
  static unsigned int next_victim = 0;
  victim = next_victim;
  next_victim = (next_victim + 1) % NUM_TLB;
  return victim;
}
```

Read-Only Text Segment

In the dumbvm system, all three address-space segments (text, data, and stack) are both readable and writable by the application. For this assignment, you should change this so that each application's text segment is read-only. Your kernel should set up TLB entries so that any attempt by an application to modify its text section will cause the MIPS MMU to generate a read-only memory exception (VM_FAULT_READONLY). If such an exception occurs, your kernel should terminate the process that attempted to modify its text segment. Your kernel should not crash.

On-Demand Page Loading

Currently, when OS/161 loads a new program into an address space using runprogram, it pre-allocates physical frames for all of the program's virtual pages, and it pre-loads all of the pages into physical memory.

For this assignment, you are required to change this so that physical frames are allocated on-demand and virtual pages are loaded on demand. "On demand" means that that the page should be loaded (and physical space should be allocated for it) the first time that the application tries to use (read or write) that page. Pages that are never used by an application should never be loaded into memory and should not consume a physical frame.

In order to do this, your kernel will need to have some means of keeping track of which parts of physical memory are in use and which parts can be allocated to hold newly-loaded virtual pages. Your kernel will also need a way to keep track of which pages from each address space have been loaded into physical memory and where in physical memory they have been loaded.

Since a program's pages will not be pre-loaded into physical memory when the program starts running, and since the TLB only maps pages that are in memory, the program will generate TLB miss exceptions as it tries to access its virtual pages. Here is a high-level description of what the OS/161 kernel must do when the MMU generates a TLB miss exception for a particular page:

- Determine whether the page is already in memory.
- If it is already in memory, load an appropriate entry into the TLB (replacing an existing TLB entry if necessary) and then return from the exception.
- If it is not already in memory, then
    - Allocate a place in physical memory to store the page.
    - Load the page, using information from the program's ELF file to do so.
    - Update OS/161's information about this address space.
    - Load an appropriate entry into the TLB (replacing an existing TLB entry if necessary) and return from the exception.

Until you implement page replacement, you will not be able to run applications that touch more pages than will fit into physical memory, but you should be able to run large programs provided that those programs do not touch more pages than will fit.

Page Replacement

You should implement a page replacement policy of your choosing. A very simple (even poor) algorithm that works is preferred to a more complex algorithm that you can't get to work. Do not worry about implementing techniques to avoid or control thrashing. Pages that need to be written to disk should be written to a file named SWAPFILE. This file will be limited to 9 MB (i.e., 9 * 1024 * 1024 bytes). If at run time more than 9 MB of swap space is required your

kernel should call panic("Out of swap space"). Make the maximum size of the swap file easy to change at compile time, in case we need to change this requirement before final submissions.

Instrumentation

You should be tracking and printing several statistics related to the performance of the virtual memory sub-system (including TLB misses and TLB replacements) so be certain to implement this as described so we can easily examine and compare these statistics.

You can collect the following statistics:

- TLB Faults: The number of TLB misses that have occurred (not including faults that cause a program to crash).
- TLB Faults with Free: The number of TLB misses for which there was free space in the TLB to add the new TLB entry (i.e., no replacement is required).
- TLB Faults with Replace: The number of TLB misses for which there was no free space for the new TLB entry, so replacement was required.
- TLB Invalidations: The number of times the TLB was invalidated (this counts the number times the entire TLB is invalidated NOT the number of TLB entries invalidated)
- TLB Reloads: The number of TLB misses for pages that were already in memory.
- Page Faults (Zeroed): The number of TLB misses that required a new page to be zero-filled.
- Page Faults (Disk): The number of TLB misses that required a page to be loaded from disk.
- Page Faults from ELF: The number of page faults that require getting a page from the ELF file.
- Page Faults from Swapfile: The number of page faults that require getting a page from the swap file.
- Swapfile Writes: The number of page faults that require writing a page to the swap file.

Note that the sum of “TLB Faults with Free” and “TLB Faults with Replace” should be equal to “TLB Faults.” Also, the sum of “TLB Reloads,” “Page Faults (Disk),” and “Page Faults (Zeroed)” should be equal to “TLB Faults.” So this means that you should not count TLB faults that do not get handled (i.e., result in the program being killed). The code for printing out stats will print a warning if this these equalities do not hold. In addition the sum of “Page Faults from ELF” and “Page Faults from Swapfile” should be equal to “Page Faults (Disk)”.

When it is shut down (e.g., in vm_shutdown), your kernel should display the statistics it has gathered. The display should look like the example below.

Files/directories

You should begin with a careful review of the existing OS161 code with which you will be working. The rest of this section identifies some important files for you to consider.

In kern/vm

addrspace.c: The machine-independent part of your virtual-memory implementation (alternative to kern/arch/mips/vm/dumbvm.c) should go in this directory.

In kern/syscall

loadelf.c: This file contains the functions responsible for loading an ELF executable from the filesystem into virtual-memory space. You should already be familiar with this file from Lab 2. Since you will be implementing on-demand page loading, you will need to change the behaviour that is implemented here.

In kern/vm

kmalloc.c: This file contains implementations of kmalloc and kfree, to support dynamic memory allocation for the kernel. It should not be necessary for you to change the code in this file, but you do need to understand how the kernel's dynamic memory allocation works so that your physical-memory manager will interact properly with it.

In kern/include

addrspace.h: define the addrspace interface. You will need to make changes here, at least to define an appropriate addrspace structure.

vm.h: Some VM-related definitions, including prototypes for some key functions, such as vm_fault (the TLB miss handler) and alloc_kpages (used, among other places, in kmalloc).

In kern/arch/mips/vm

dumbvm.c: This file shpuld not be used (when disabling option dumbvm). However, you can use the code here (with improvements done in lab 2). This code also includes examples of how to do things like manipulate the TLB.

ram.c: This file includes functions that the kernel uses to manage physical memory (RAM) while the kernel is booting up, before the VM system has been initialized. Since your VM system will essentially be taking over management of physical memory, you need to understand how these functions work.

In kern/arch/mips/include

In this directory, the file tlb.h defines the functions that are used to manipulate the TLB. In addition, vm.h includes some macros and constants related to address translation on the MIPS. Note that this vm.h is different from the vm.h in kern/include.

You are free and will need to modify existing kernel code in addition you'll probably need some code to create and use some new abstractions. If you uses any or all of the following abstractions please place that code in the directory kern/vm using the following file names:

- coremap.c: keep track of free physical frames
- pt.c: page tables and page table entry manipulation go here

• segments.c: code for tracking and manipulating segments

• vm_tlb.c: code for manipulating the tlb (including replacement)

• swapfile.c: code for managing and manipulating the swapfile

• vmstats.c: code for tracking stats

If you need them, corresponding header files should be placed in os161-1.11/kern/include in files named: addrspace.h, coremap.h, pt.h, segments.h, vm_tlb.h, vmstats.h, and swapfile.h.

Possible variants of the project

The project can be taken in one of three variants.

C1.1) Per-process page table, where the problem of the "empty" region between the two segments and the stack should be addressed

C1.2) Inverted Page Table, with "some" solution in order to speed-up linear search (not necessarily a hash, though possible).

C1.3) TLB support for the "dirty" bit (modified page): it has to be addressed by software, as the modify bit is not handled automatically upon write operations. If not taking this option, "data" pages can be considered as dirty/modified even when not written.

Project C2: Shell (SHELL)

Project's summary

The purpose of this project is to support running multiple processes at once from actual compiled programs stored on disk. These programs will be loaded into OS161 and executed in user mode, under the control of your kernel and the command shell in `bin/sh` (menu command: `p bin/sh`). The project is highly based on the availability of the `execv` and `dup2` system calls. The project can be limited to the EMUFS emulated file system

Required Background

Lab 2 to Lab 5 and basic knowledge of process management and file system are needed (see lessons os161-overview, os161-memory, os161-userprocess).

Working Environment

Your current OS161 system has minimal support for running executables -- nothing that could be considered a true process. Lab 2 to Lab 5 have provided limited support for process memory management, exit and file system.

The key files that are responsible for the loading and running of user-level programs are loadelf.c, runprogram.c, and uio.c, Understanding these files is the key to getting started with the implementation of multiprogramming.

kern/syscall/loadelf.c: This file contains the functions responsible for loading an ELF executable from the filesystem and into virtual memory space. (ELF is the name of the executable format produced by os161-gcc.) Of course, at this point this virtual memory space does not provide what is normally meant by virtual memory -- although there is translation between the addresses that executables "believe" they are using and physical addresses, there is no mechanism for providing more memory than exists physically.

kern/syscall/runprogram.c: This file contains only one function, runprogram(), which is responsible for running a program from the kernel menu. It 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 concern itself with. Additionally, once you have designed your process system, runprogram() should be altered to start processes properly within this framework; for example, a program started by runprogram() should have the standard file descriptors available while it's running.

kern/lib/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 userlevel programs, so this is a good file to read very carefully. You should also examine the code in kern/vm/copyinout.c.

kern/arch/mips: traps and syscalls

Exceptions are the key to operating systems; they are the mechanism that enables 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" (carefully crafted assembly code) at a specific address in memory. When the processor raises an exception, it invokes this exception handler, which sets up a "trap frame" and calls into the operating system. Since "exception" is such an overloaded term in computer science, operating system lingo for an exception is a "trap". Interrupts are exceptions, and more significantly for this assignment, so are system calls. Specifically, syscall/syscall.c handles traps that happen to be syscalls. Understanding at least the C code in this directory is key to being a real operating systems junkie, so we highly recommend reading through it carefully.

locore/trap.c: mips_trap() is the key function for returning control to the operating system. This is the C function that gets called by the assembly exception handler. enter_new_process() is the key function for returning control to user programs. kill_curthread() is the function for handling broken user programs; when the processor is in usermode and hits something it can't handle (say, a bad instruction), it raises an exception. There's no way to recover from this, so the OS needs to kill off the process. Part of this assignment will be to write a useful version of this function.

syscall/syscall.c: syscall() is the function that 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, enter_forked_process(), which is a stub where you will place your code to implement the fork()system call. It should get called from sys_fork().

user/lib/crt0/mips: This is the user program startup code. There's only one file in here, mips-crt0.S, which contains the MIPS assembly code that receives control first when a user-level program is started. It calls the user program's main(). This is the code that your execv() implementation will be interfacing to, so be sure to check what values it expects to appear in what registers and so forth.

user/lib/libc: This is the user-level C library. There's obviously a lot of code here. We don't expect you to read it all, although it may be instructive in the long run to do so. Job interviewers have an uncanny habit of asking people to implement standard C library functions on the whiteboard. For present purposes you need only look at the code that implements the user-level side of system calls, which we detail below.

user/lib/libc/unix/errno.c: This is where the global variable errno is defined.

user/lib/libc/arch/mips/syscalls-mips.S: This file contains the machine-dependent code necessary for implementing the user-level side of MIPS system calls.

syscalls.S: This file is created from syscalls-mips.S at compile time and is the actual file assembled into the C library. The actual names of the system calls are placed in this file using a script called syscalls/gensyscalls.sh that 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.

Problem Definition

System calls and exceptions

Implement system calls (for file and process management) and exception handling. The full range of already defined (not implemented) system calls is listed in kern/include/kern/syscall.h. For this project, you should complete and/or implement:

- open, read, write, lseek, close, dup2, chdir, getcwd
- getpid
- fork, execv, waitpid, _exit

It's crucial that your syscalls handle all error conditions gracefully (i.e., without crashing OS/161.) You should consult the OS/161 man pages included in the distribution and understand fully the system calls that you must implement. You must return the error codes as described in the man pages.

Additionally, your syscalls must return the correct value (in case of success) or error code (in case of failure) as specified in the man pages. Some of the grading scripts rely on the return of appropriate error codes; adherence to the guidelines is as important as the correctness of the implementation.

The file user/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 this interface and put it in kern/include/syscall.h. The integer codes for the calls are defined in kern/include/kern/syscall.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 find themselves running a system call at the same time? Be sure to argue for or against this, and explain your final decision in the design document.

open(), read(), write(), lseek(), close(), dup2(), chdir(), and __getcwd()

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). These file descriptors should start out attached to the console device ("con:"), but your implementation must allow programs to use dup2() to change them to point elsewhere.

Although these system calls may seem to be tied to the filesystem, in fact, these system calls are really about manipulation of file descriptors, or process-specific filesystem state. A large part of this assignment is designing and implementing a system to track this state. Some of this information (such as the current working directory) is specific only to the process, but others (such as file offset) is specific to the process and file descriptor. Don't rush this design. Think carefully about the state you need to maintain, how to organize it, and when and how it has to change.

Note that there is a system call __getcwd() and then a library routine getcwd(). Once you've written the system call, the library routine should function correctly.

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've used up all the pids. Design your pid system; implement all the tasks associated with pid maintenance, and only then implement getpid().

fork(), execv(), waitpid(), _exit()

These system calls are probably the most difficult part of the assignment, but also the most rewarding. They enable multiprogramming and make OS/161 a much more useful entity.

fork() is the 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 (that is, 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 than 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 (as runprogram() does), it's not quite that simple. Remember that this call is coming out of userspace, into the kernel, and then returning back to userspace. You must manage the memory that travels across these boundaries very carefully. (Also, notice that runprogram() doesn't take an argument vector -- but this must of course be handled correctly in execv()).

A note on errors and error handling of system calls:

The man pages in the OS/161 distribution contain a description of the error return values that you must return. If there are conditions that can happen that are not listed in the man page, return the most appropriate error code from kern/include/kern/errno.h. If none seem particularly appropriate, consider adding a new one. If you're 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, should not be EOF, etc. Consult Unix man pages to learn about Unix error codes; on Linux systems man errno will do the trick.

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 filesrc/kern/include/kern/errmsg.h.

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 userspace 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, but without returning execution to the user level.

Testing using the shell

In user/bin/sh you will find a simple shell that will allow you to test your new system call interfaces. When executed, the shell prints a prompt, and allows you to type simple commands to run other programs. Each command and its argument list (an array of character pointers) is passed to the execv() system call, after calling fork() to get a new thread for its execution. The shell also allows you to run a job in the background using &. You can exit the shell by typing "exit".

Under OS/161, once you have the system calls for this assignment, you should be able to use the shell to execute the following user programs from the bin directory: cat, cp, false, pwd, and true. You will also find several of the programs in the testbin directory helpful.

Project C3: Buffer Cache and Pipes (BUFFER)

Project's summary

The project can be seen as a variant of project 2, in terms of file system support, but without the need to support shell, and provide `execv` and `dup2`. So `fork` is enough. In addition, the pipe system call is required, as well as an implementation of the buffer cache with the SFS file system. The purpose of this project is to improve the performance of file operations in terms of disk block references, as well as supporting inter-process data exchange through pipes. The project cannot be limited to the EMUFS emulated file system, but it needs to address the SFS file system.

Required Background

Lab 2 to Lab 5 and basic knowledge of process management and file system are needed (see lessons os161-overview, os161-memory, os161-userprocess).

Working Environment

Your current OS161 system has minimal support for running executables -- nothing that could be considered a true process. Lab 2 to Lab 5 have provided limited support for process memory management, exit and file system. Two file systems are now supported, EMUFS is a pass-though file system relying on the host system, SFS is a simple file system implemented within OS161. For more info on using SFS on OS1616, have a look here: http://os161.eecs.harvard.edu/resources/sfs.html

The key files that are responsible for the loading and running of user-level programs are loadelf.c, runprogram.c, and uio.c, Understanding these files is the key to getting started with the implementation of multiprogramming.

kern/syscall/loadelf.c: This file contains the functions responsible for loading an ELF executable from the filesystem and into virtual memory space. (ELF is the name of the executable format produced by os161-gcc.) Of course, at this point this virtual memory space does not provide what is normally meant by virtual memory -- although there is translation between the addresses that executables "believe" they are using and physical addresses, there is no mechanism for providing more memory than exists physically.

kern/syscall/runprogram.c: This file contains only one function, runprogram(), which is responsible for running a program from the kernel menu. It 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 concern itself with. Additionally, once you have designed your process system, runprogram() should be altered to start processes properly within this framework; for example, a program started by runprogram() should have the standard file descriptors available while it's running.

kern/lib/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 userlevel

programs, so this is a good file to read very carefully. You should also examine the code in kern/vm/copyinout.c.

kern/fs/sfs: The support for the SFS file system is provided in this directory. More details on the few files that you need to modify in the buffer cache paragrafh

kern/arch/mips: traps and syscalls

Exceptions are the key to operating systems; they are the mechanism that enables 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" (carefully crafted assembly code) at a specific address in memory. When the processor raises an exception, it invokes this exception handler, which sets up a "trap frame" and calls into the operating system. Since "exception" is such an overloaded term in computer science, operating system lingo for an exception is a "trap". Interrupts are exceptions, and more significantly for this assignment, so are system calls. Specifically, syscall/syscall.c handles traps that happen to be syscalls. Understanding at least the C code in this directory is key to being a real operating systems junkie, so we highly recommend reading through it carefully.

locore/trap.c: mips_trap() is the key function for returning control to the operating system. This is the C function that gets called by the assembly exception handler. enter_new_process() is the key function for returning control to user programs. kill_curthread() is the function for handling broken user programs; when the processor is in usermode and hits something it can't handle (say, a bad instruction), it raises an exception. There's no way to recover from this, so the OS needs to kill off the process. Part of this assignment will be to write a useful version of this function.

syscall/syscall.c: syscall() is the function that 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, enter_forked_process(), which is a stub where you will place your code to implement the fork()system call. It should get called from sys_fork().

user/lib/crt0/mips: This is the user program startup code. There's only one file in here, mips-crt0.S, which contains the MIPS assembly code that receives control first when a user-level program is started. It calls the user program's main(). This is the code that your execv() implementation will be interfacing to, so be sure to check what values it expects to appear in what registers and so forth.

user/lib/libc: This is the user-level C library. There's obviously a lot of code here. We don't expect you to read it all, although it may be instructive in the long run to do so. Job interviewers have an uncanny habit of asking people to implement standard C library functions on the whiteboard. For present purposes you need only look at the code that implements the user-level side of system calls, which we detail below.

user/lib/libc/unix/errno.c: This is where the global variable errno is defined.

user/lib/libc/arch/mips/syscalls-mips.S: This file contains the machine-dependent code necessary for implementing the user-level side of MIPS system calls.

syscalls.S: This file is created from syscalls-mips.S at compile time and is the actual file assembled into the C library. The actual names of the system calls are placed in this file using a script called syscalls/gensyscalls.sh that 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.

Problem Definition

System calls and exceptions

Implement system calls (for file and process management) and exception handling. The full range of already defined (not implemented) system calls is listed in kern/include/kern/syscall.h. For this project, you should complete and/or implement:

- open, read, write, lseek, close, pipe
- getpid
- fork, waitpid, _exit

It's crucial that your syscalls handle all error conditions gracefully (i.e., without crashing OS/161.) You should consult the OS/161 man pages included in the distribution and understand fully the system calls that you must implement. You must return the error codes as described in the man pages.

Additionally, your syscalls must return the correct value (in case of success) or error code (in case of failure) as specified in the man pages. Some of the grading scripts rely on the return of appropriate error codes; adherence to the guidelines is as important as the correctness of the implementation.

The file user/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 this interface and put it in kern/include/syscall.h. The integer codes for the calls are defined in kern/include/kern/syscall.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 find themselves running a system call at the same time? Be sure to argue for or against this, and explain your final decision in the design document.

open(), read(), write(), lseek(), close(), and pipe()

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). These file descriptors should start out attached to the console device ("con:"), your implementation doesn't need to change them to point elsewhere. The pipe system call is also requested: pipe creates an array of two file descriptors (connected by a bounded buffer).

Although these system calls may seem to be tied to the filesystem, in fact, these system calls are really about manipulation of file descriptors, or process-specific filesystem state. A large part of this assignment is designing and implementing a system to track this state.

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've used up all the pids. Design your pid system; implement all the tasks associated with pid maintenance, and only then implement getpid().

fork(), waitpid(), _exit()

These system calls are probably the most difficult part of the assignment, but also the most rewarding. They enable multiprogramming and make OS/161 a much more useful entity.

fork() is the 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 (that is, 0 for the child and the newly created pid for the parent).

A note on errors and error handling of system calls:

The man pages in the OS/161 distribution contain a description of the error return values that you must return. If there are conditions that can happen that are not listed in the man page, return the most appropriate error code from kern/include/kern/errno.h. If none seem particularly appropriate, consider adding a new one. If you're 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, should not be EOF, etc. Consult Unix man pages to learn about Unix error codes; on Linux systems man errno will do the trick.

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 filesrc/kern/include/kern/errmsg.h.

Buffer cache

In this project, you'll make a buffer cache capable of speeding up read and write requests with the sfs file system. So first of al you need to learn how to use the sfs file system (instead of the emu file system).

The current implementation of read/write system calls will redirect file IO to proper VOP_READ or VOP_WRITE block operations. Whenever the underlying file system is sfs, the operation will be mapped (se the vop_write and vop_read fields in struct vnode_ops, file vnode.h) to sfs_read and sfs_write. The pattern will end up calling sfs_io, and down the call tree, sfs_blockio, sfs_partialio. A similar strategy could be observed with sfs_readdir, sfs_writedir, redirected to sfs_metaio. Functions sfs_blockio, sfs_partialio and sfs_metaio internally perform disk block read/write by calling sfs_bmap (to find the physical disk block number from file and logical block) and sfs_readblock/sfs_writeblock.

The buffer cache should hopefully contain blocks that have previously been used and are currently used. Your buffer cache should work in such a way that blocks in the buffer are not evicted immediately (but kept for future use), and it should also contain blocks that *will likely be used in the future*.

To implement a good buffer cache, you will need an **eviction policy**, which decides which (unreferenced) block to evict when the cache is full but a new block should be added. LRU (least recently used) is a good first choice, but there are other good choices, such as policies that treat different kinds of blocks differently.

You will also want a **prefetching policy**, which decides which blocks to read in advance of their being needed.

You will also want to raise the number of entries in the buffer cache, although your buffer cache should work for any number of entries greater than or equal to 10.

Writeback

Next, support writing data back to the disk via the sync system call.

Writeback hints

Basics: You'll need to track which buffers are *dirty*, meaning modified in memory relative to the on-disk version.

Synchronization: The writeback operation must write an internally consistent version of each dirty buffer, meaning that buffer contents must not be modified or freed while the buffer is in flight to the disk. This requires both delaying writeback until concurrent writes complete, and delaying new concurrent writes and evictions until writeback completes.

A simple synchronization method that works is a per-bufentry *write reference count* that can be either zero or one. Consider completing these functions:

Pipes

The pipe system call is also requested, in order to support pipes, as well as the development of a test program based on fork, with parent and child exchanging data through a pipe. Details on how to implement pipes will be discussed later.