15-410, Operating System Design & Implementation

# Pebbles Kernel Specification

February 8, 2006

## Contents

# 1 Introduction

This document defines the correct behavior of kernels for the Spring 2006 edition of 15-410. The goal of this document is to supply information about behavior rather than implementation details. In Project 2 you will be given a kernel binary exhibiting these behaviors upon which to build your thread library; later, in Project 3, you will construct a kernel which behaves this way.

## 1.1 Overview

The 410 kernel environment supports multiple address spaces via hardware paging, preemptive multitasking, and a small set of important system calls. Also, the kernel supplies device drivers for the keyboard, the console, and the interval timer.

# 2 User Execution Environment

The "Pebbles" kernel supports multiple independent *tasks*, each of which serves as a protection domain. A task's resources include various memory regions and "invisible" kernel resources (such as a queue of task-exit notifications). Some versions of the kernel support file I/O, in which case file descriptors are task resources as well.

Execution proceeds by the kernel scheduling *threads*. Each thread represents an independently-schedulable register set; all memory references and all system calls issued by a thread represent accesses to resources defined and owned by the thread's enclosing task. A task may contain multiple threads, in which case all have equal access to all task resources. A carefully designed set of cooperating library routines can leverage this feature to provide a simplified version of POSIX "pthreads."

Multiprocessor versions of the kernel may simultaneously run multiple threads of a single task, one thread for each of several tasks, or a mixture.

When a task begins execution of a new program, the operating system builds several memory regions from the executable file and command line arguments:

- A read-only code region containing machine instructions
- An optional read-only-constant data region
- A read/write data region containing some variables.
- A single automatic stack region containing a mixture of variables and procedure call return information. The stack begins at some "large" address and memory accesses typically cause the kernel to add new pages, growing the region downward toward the top of the data region. Of course, if they collide, disaster will result.

In addition, the task may add memory regions as specified below. All memory added to a task's address space after it begins running is zeroed before any thread of the task can access it.

Pebbles allows one task to create another though the use of the `fork()` and `exec()` system calls, which you will not need for Project 2 (the shell program which we provide so you can launch your test programs does use them).

# 3 The System Call Interface

## 3.1 Invocation and Return

User code will make requests of the kernel by issuing a software interrupt using the `INT` instruction. Interrupt numbers are defined in `410user/lib/inc/syscall_int.h`.

To invoke a system call, the following protocol is followed. If the system call takes one 32-bit parameter, it is placed in the `%esi` register. Then the appropriate interrupt, as defined in `syscall_nums.h`, is raised via the `INT x` instruction (each system call has been assigned its own `INT` instruction, hence its own value of `x`). If the system call expects more than one 32-bit parameter, you should construct in memory a "system call packet" containing the parameters and place the *address* of the packet in `%esi`.

To create such a packet in C you could use a structure:

```
struct read_line_parms {
  int len;
  char *buf;
} rlp;
```

If you use this approach, it is probably a good idea for you to think about the declarations of your "packet" structures. In particular, you probably want to consider how widely known these types must be.

After filling in the packet, you would arrange for its address (e.g., `&rlp`) to be placed in `%esi`. When the system call completes, the return value, if any, will be available in the `%eax` register.

## 3.2 Semantics of System Call Interface

The 410 kernel verifies that every byte of every system call argument lies in a memory region which the invoking thread's task has appropriate permission to access. System calls will return an integer error code less than zero if any part of any argument is invalid. The kernel *does not* kill a user thread that invokes a system call with a bad argument. No action taken by user code should *ever* cause the kernel to crash, hang, or otherwise fail to perform its job.

## 3.3 System Call Stub Library

While the kernel provides system calls for your use, it does not provide a "C library" which accesses those calls. Before your programs can get the kernel to do anything for them, you will need to implement an assembly code "stub" for each system call.

Stub routines *must* be one per file and you should arrange for the Makefile infrastructure you are given to build them into `libsyscall.a` (see the `README` file in the tarball). While system call stubs resemble the trap handler wrappers you wrote for Project 1, they are different in one critical way. Since your kernel must always be ready to respond to any interrupt or trap, it can potentially use every wrapper during each execution, and all must be linked (once) into the kernel executable. However, the average user program does *not* invoke every system call during the course of its execution. In fact, many user programs contain only a trivial amount of code. If you create one huge system call stub file containing the code to invoke every system call, the linker will happily append the huge .o file to *every* user-level program you build and your "RAM disk" file system will overflow, probably when we are trying to grade your project. So don't do that.

While the project tarball contains a single `syscall.c`, full of blank system call stubs, this is only a convenience so that you can link test programs before you have completed all your stubs–as you write each stub, this file should get smaller until eventually being deleted.

When building your stub library, you *must* match the declarations we have provided in `410user/lib/inc/syscall.h` in every detail. Otherwise, our test programs will not link against your stub library. If you think there is a problem with a declaration we have given you, explain your thinking to us–don't just "fix" the declaration. Any system-call entry code which doesn't map straightforwardly from a declaration in `syscall.h` into code isn't a "genuine" stub routine and shouldn't be part of `libsyscall.a`–code specific to some application or facility should be in the appropriate place in the directory tree.

Please remember your x86 calling convention rules. If you modify any callee-saved registers inside your stub routines, you must restore their values before returning to your caller. The kernel, of course, always preserves the values of all user-modifiable registers except when it explicity modifies them according to the system call specifications.

# 4 System Call Specifications

## 4.1 Overview

The system calls provided by the 410 kernel can be broken into five groups, namely

- Life Cycle
- Thread Management
- Memory Management
- Console I/O
- Miscellaneous System Interaction

The following descriptions of system calls use C function declaration syntax even though the actual system call interface, as described in Section 3, is defined in terms of assembly-language primitives. This means that student teams must write a system call stub library, as described in Section 3.3, in order to invoke any system calls. This stub library is a deliverable.

Unless otherwise noted, system calls return zero on success and an error code less than zero if something goes wrong.

One system call, `thread_fork`, is presented without a C-style declaration. This is because the actions performed by `thread_fork` are outside of the scope of, and manipulate, the C language runtime environment. You will need to determine for yourself the correct manner and context for invoking `thread_fork`. It is `not` an oversight that `thread_fork` is "missing" from `syscall.h`, and you must not "fix" this oversight. If you feel a need to declare a C function called `thread_fork()`, think carefully about whether that is really the best name for the function, what parameters it should take, who needs to "see" the declaration, etc.

## 4.2 Task & Thread IDs

Task and thread identification numbers are monotonically increasing throughout the execution of the kernel. In other words, once there is a thread #35, there will not be another thread #35 until an intervening four billion threads have been created.

## 4.3 Life Cycle

This group contains system calls which manage the creation and destruction of tasks and threads.

- `int fork(void)` - Creates a new task. The new task receives an exact, coherent copy of all memory regions of the invoking task. The new task contains a single thread which is a copy of the thread invoking `fork()` except for the return value of the system call. If `fork()` succeeds, the invoking thread will receive the ID of the new task's thread and the newly created thread will receive the value zero.

  Errors are reported via a negative return value, in which case no new task has been created.

  Some kernel implementations reject calls to `fork()` which take place while the invoking task contains more than one thread.

- `thread_fork` - Creates a new thread in the current task (i.e., the new thread will share all task resources as described in Section 2). The value of `%esi` is ignored, i.e., the system call has no parameters.

  The invoking thread's return value in `%eax` is the thread ID of the newly-created thread; the new thread's return value is zero. All other registers in the new thread will be initialized to the same values as the corresponding registers in the old thread.

  Errors are reported via a negative return value, in which case no new thread has been created.

  Some kernel versions reject calls to `fork()` or `exec()` which take place while the invoking task contains more than one thread.

- `int exec(char *execname, char **argvec)` - Replaces the program currently running in the invoking task with the program stored in the file named `execname`. The argument `argvec` points to a null-terminated vector of null-terminated string arguments.

  The number of strings in the vector and the vector itself will be transported into the memory of the new task where they will serve as the first and second arguments of the the new program's `main()`, respectively. It is conventional that `argvec[0]` is the same string as `execname` and `argvec[1]` is the first command line parameter, etc. Some programs will behave oddly if this convention is not followed.

  Reasonable limits may be placed on the number of arguments that a user program may pass to `exec()`, and the length of each argument.

  The kernel does as much validation as possible of the `exec()` request before deallocating the old program's resources.

  On success, this system call does not return to the invoking program, since it is no longer running. If something goes wrong, an integer error code less than zero will be returned.

  Some kernel versions reject calls to `exec()` which take place while the invoking task contains more than one thread.

- `void exit(int status)` - Terminates execution of the calling thread immediately. If the invoking thread is the last thread in its task, the kernel deallocates all resources in use by the task and makes the `status` parameter available to the parent task (the task which created this one using `fork()`) via `wait()`. If the parent task is no longer running, the exit status is made available to the kernel-launched "init" task instead. If the invoking thread is not the last thread in its task, `status` will be ignored.

  If the kernel decides to kill a thread, the effect should be the same as if the thread had invoked `exit(-2)`, except that the kernel can generally be expected to display an appropriate message on the system console.

  The `exit()` of one thread, voluntary or involuntary, does not cause the kernel to destroy other threads in the same task.

- `int wait(int *status_ptr)` -

  Collects the exit status of a task, defined as the `status` parameter provided to the `exit()` system call by the final thread in the task, and stores it in the integer referenced by `status_ptr`.

  If no error occurs, the return value of `wait()` is the thread ID of the *first* thread of the exiting task, *not* the thread ID of the last thread in that task to `exit()`. This should make sense if you consider how `fork()`, `exit()`, and `wait()` interact.

  The `wait()` system call may be invoked simultaneously by any number of threads in a task; exit statuses may be matched to `wait()`'ing threads in any non-pathological way. If one or more threads invoke `wait()` while there are child tasks which have not yet exited, they will block until one exits.

Whenever a task has no un-exited child tasks, any pending or new calls to `wait()` will return an integer error code less than zero.

- `void task_exit(int status)` - Causes all threads of a task to exit. The behavior of the system call should be as if the invoking thread "exits last," i.e., the `status` parameter becomes the exit status of the task as described above.

  The threads must exit "in a timely fashion," meaning that it is `not` ok for `task_exit()` to "wait around" for threads to complete very-long-running or unbounded-time operations.

## 4.4 Thread Management

- `int gettid()` - Returns the thread ID of the invoking thread.

- `int yield(int tid)` - Defers execution of the invoking thread to a time determined by the scheduler, in favor of the thread with ID `tid`. If `tid` is -1, the scheduler may determine which thread to run next. The only threads whose scheduling should be affected by `yield()` are the calling thread and the thread that is `yield()`ed to. If the thread with ID `tid` is not runnable, blocked in a system call, or doesn't exist, then an integer error code less than zero is returned. Zero is returned on success.

- `int deschedule(int *reject)` - Atomically checks the integer pointed to by `reject` and acts on it. If the integer is non-zero, the call returns immediately with return value zero. If the integer pointed to by `reject` is zero, then the calling thread will not be run by the scheduler until a `make_runnable()` call is made specifying the `deschedule()`'d thread, at which point `deschedule()` will return zero.

  An integer error code less than zero is returned if reject is not a valid pointer.

  This system call is *atomic* with respect to `make_runnable()`: the process of examining `reject` and suspending the thread will not be interleaved with any execution of `make_runnable()` specifying the thread calling `deschedule()`.

- `int make_runnable(int tid)` - Makes the `deschedule()`d thread with ID `tid` runnable by the scheduler. On success, zero is returned. An integer error code less than zero will be returned unless `tid` is the ID of a thread which exists but is currently non-runnable due to a call to `deschedule()`.

- `unsigned int get_ticks(void)` - Returns the number of timer ticks which have occurred since system boot.

- `int sleep(int ticks)` - Deschedules the calling thread until at least `ticks` timer interrupts have occurred after the call. Returns immediately if `ticks` is zero. Returns an integer error code less than zero if `ticks` is negative. Returns zero otherwise.

## 4.5 Memory Management

- `int new_pages(void *base, int len)` - Allocates new memory to the invoking task, starting at `base` and extending for `len` bytes.

  `new_pages()` will fail, returning a negative integer error code, if `base` is not page-aligned, if `len` is not a positive integral multiple of the system page size, if any portion of the region already represents memory in the task's address space, if the new memory region would be too close[1] to the bottom of the automatic stack region, or if the operating system has insufficient resources to satisfy the request.

  Otherwise, the return code will be zero and the new memory will immediately be visible to all threads in the invoking task.

- `int remove_pages(void *base)` - Deallocates the specified memory region, which must presently be allocated as the result of a previous call to `new_pages()` which specified the same value of `base`. Returns zero if successful or returns a negative integer failure code.

## 4.6 Console I/O

- `char getchar()` - Returns a single character from the character input stream. If the input stream is empty the thread is descheduled until a character is available. If some other thread is descheduled on a `readline()` or `getchar()`, then the calling thread must block and wait its turn to access the input stream. Characters processed by the `getchar()` system call should not be echoed to the console.

- `int readline(int len, char *buf)` - Reads the next line from the console and copies it into the buffer pointed to by `buf`.

  If there is no line of input currently available, the calling thread is descheduled until one is. If some other thread is descheduled on a `readline()` or a `getchar()`, then the calling thread must block and wait its turn to access the input stream. The length of the buffer is indicated by `len`. If the line is smaller than the buffer, then the complete line including the newline character is copied into the buffer. If the length of the line exceeds the length of the buffer, only `len` characters should be copied into `buf`. Available characters should not be committed into `buf` until there is a newline character available, so the user has a chance to backspace over typing mistakes.

  Characters that will be consumed by a `readline()` should be echoed to the console as soon as possible. If there is no outstanding call to `readline()` no characters should be echoed. Echoed user input may be interleaved with output due to calls to `print()`. Characters not placed in the buffer should remain available for other calls to `readline()` and/or `getchar()`. Some kernel implementations may choose to regard characters which have been echoed to the screen but which have not been placed into a user buffer to be "dedicated" to `readline()` and not available to `getchar()`.

[1] Two pages is too close. Other values might be too close also.

The readline system call returns the number of bytes copied into the buffer. An integer error code less than zero is returned if `buf` is not a valid memory address, if `buf` falls in a read-only memory region of the task, or if `len` is "unreasonably" large.[2]

- `int print(int len, char *buf)` - Prints `len` bytes of memory, starting at `buf`, to the console. The calling thread should block until all characters have been printed to the console. Output of two concurrent `print()`s should not be intermixed. If `len` is larger than some reasonable maximum or if `buf` is not a valid memory address, an integer error code less than zero should be returned.

  Characters printed to the console invoke standard newline, backspace, and scrolling behaviors.

- `int set_term_color(int color)` - Sets the terminal print color for any future output to the console. If `color` does not specify a valid color, an integer error code less than zero should be returned. Zero is returned on success.

- `int set_cursor_pos(int row, int col)` - Sets the cursor to the location `(row, col)`. If the location is not valid, an integer error code less than zero is returned. Zero is returned on success.

- `int get_cursor_pos(int *row, int *col)` - Writes the current location of the cursor to the addresses provided as arguments. If the arguments are not valid addresses, then an error code less than zero is returned. Zero is returned on success.

## 4.7 Miscellaneous System Interaction

- `int ls(int size, char *buf)` - Fills in the user-specified buffer with the names of executable files stored in the system's RAM disk "file system." If there is enough room in the buffer for all of the (null-terminated) file names *and* an additional null byte after the last filename's terminating null, the system call will return the number of filenames successfully copied. Otherwise, an error code less than zero is returned and the contents of the buffer are undefined. For the curious among you, this system call is (very) loosely modeled on the System V `getdents()` call.

- `void halt()` - Ceases execution of the operating system. The exact operation of this system call depends on the kernel's implementation and execution environment. Kernels running under Simics should shut down the simulation via a call to `SIM_halt()`. However, implementations should be prepared to do something reasonable if `SIM_halt()` is a no-op, which will happen if the kernel is run on real hardware.

[2]Deciding on this threshold is easier than it may seem at first, so if you feel like you need to ask us for a clarification you should probably think further.