15–410, Operating System Design & Implementation

# Pebbles Kernel Specification

September 23, 2020

## Contents

# 1 Introduction

This document defines the correct behavior of kernels for the Fall 2020 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 Pebbles 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

## 2.1 Tasks and Threads

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 Pebbles kernels 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 stack region containing a mixture of variables and procedure call return information. The stack begins at some "large" address and extends downward for some fixed distance. A memory access which "runs off the bottom of the stack" will cause an exception, which by default will kill the thread performing the access (see Sections 2.3 and 2.4).

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).

## 2.2 A Note on Terminology

Various people use the terms “process,” “thread,” and “task” in various incompatible fashions. In Unix history, originally every process had just one schedulable execution context, which ran in an address space that nobody else could access. Meanwhile, people who study scheduling often refer to things that can be executed as “tasks” (i.e., things that need to be done). Historically speaking, scheduling people didn’t much care which of the things to be executed lived in which address spaces, so it made sense that they used a different word than “process.”

When people started to write programs using multiple simultaneously schedulable execution contexts, those execution contexts were often referred to as “threads.” It would be possible to say that a process consists of one or more threads, i.e., there are single-threaded processes and multi-threaded processes, and indeed some people do describe the world this way. However, when the Mach group at CMU started adding multi-threading to a BSD Unix kernel, they chose to use “thread” for a schedulable execution context, to use “task” for a resource container shared by a collection of threads, and to leave “process” to describe the behavior of legacy programs: a single-threaded task.

Linux does something different. As far as the kernel is concerned, there are only processes. Processes can share resources in an almost endless number of ways: two processes might share one address space but have independent current working directories, or they might share one current working directory but have independent address spaces. Meanwhile, many programs are written to use threads as defined by the IEEE POSIX Threads standard (frequently known as “Pthreads”). Programs can use Linux’s Native POSIX Thread Libary (“NPTL”) to set up a cooperating group of Linux processes, e.g., one Linux process per Pthread thread. As one result, when a multi-threaded program is implemented as a cooperating group of Linux processes, the Linux `ps` (“process status”) command typically displays just one process out of each cooperating group.

In this class, we will use Mach’s straightforward resource-container/execution-context model, and we will also use Mach’s task/thread terminology. As a result, the specifications of Pebbles system calls in this document will discuss tasks and threads, not processes.

## 2.3 Exception Handling

When a thread issues an instruction which results in an exception, the hardware transfers control to the kernel. If it is not possible to repair the program’s execution and restart the instruction, it is the kernel’s responsibility to kill the thread—generally after printing a detailed explanation of the exception to the system console. This last-resort behavior is described in the documentation for the `vanish()` system call (below). In the Pebbles environment, a thread’s execution may be repaired in one of two ways. Each thread is entitled to catch and handle its own exceptions. However, in some situations an exception occurs not because a thread did something wrong but because the exception helps the kernel achieve some secret purpose. In those situations the kernel should secretly handle the exception.

### 2.3.1 Software Exception Handling

Each thread may optionally register with the kernel a “software exception handler” (see the `swexn()` system call description below). If a thread encounters an exception while it has an exception handler registered, the kernel will attempt to invoke that handler instead of carrying out the default exception outcome. As a side effect of invoking the handler, the handler is de-registered.

A software exception handler is passed the execution state of the thread at the time of the exception, in the form of a set of register values (see Figure 1). The handler may inspect the saved execution state to determine what went wrong and may take various actions as it sees fit (e.g., printing messages, changing application state, and/or adjusting the address space). When the handler is complete, it will generally instruct the kernel to register a handler for the next exception and/or atomically replace its current registers with a set of values derived from the saved register state that caused the exception. The replacement execution state may result in the original instruction being retried or in some other instruction being run instead.

Because exceptions are synchronous to the instruction stream, there is no question about which order multiple exceptions for a single thread should be processed in: each time the hardware reports an exception, the kernel will invoke the thread's user-space exception handler, if registered, to handle that exception. In general, the author of a software exception handler will wish to ensure that the handler does not itself encounter an exception, though this is not an absolute rule.

Note that this handler approach is used for *exceptions*, as opposed to *interrupts* or *traps*.

#### 2.3.2 Internal Exception Handling

Finally, note that page faults *generally* invoke the software exception mechanism, but kernels are free to add "secret" page faults in order to implement features such as copy-on-write; those "secret" page faults, which result though a thread has issued a *logically valid* memory request, are handled secretly by the kernel and do *not* cause a user-space software exception handler to be invoked.

## 2.4 Stack Growth

Many runs of many programs use very little stack space, but some runs of some programs use quite a bit. It can be difficult to predict the exact amount of stack space a program will use. Traditional Unix kernels conveniently provide traditional single-threaded C programs with an automatically-growing stack region. When a program starts running, the kernel has provided a stack of some size; if the program "runs off the bottom" of this stack, the hardware will invoke an exception handler

```
typedef struct ureg_t {
    unsigned int cause;
    unsigned int cr2;   /* Or else zero. */

    unsigned int ds;
    ...some registers omitted...
    unsigned int edi;
    unsigned int esi;
    ...some registers omitted...
    unsigned int error_code;
    unsigned int eip;
    unsigned int cs;
    unsigned int eflags;
    unsigned int esp;
    unsigned int ss;
} ureg_t;
```

Figure 1: `struct ureg_t` from `syscall.h`

in the kernel. The kernel typically responds to memory exceptions which are not "too far" past the bottom of the stack by allocating new memory, growing the stack, and transparently restarting the program. If a traditional Unix kernel can't allocate more memory for the program, it generally terminates the program's execution.

It is less clear how to handle stack growth in multi-threaded programs: there are multiple stack regions, and it's difficult for a kernel to know how much memory to allocate to each one before refusing to add more.

In the Pebbles run-time environment, stack-growth behavior is completely in the hands of user code. When the first thread starts running in a newly created address space, the kernel will have provided a stack of some size (see the `exec()` documentation), constructed as if via a call to the `new_pages()` system call; any future growth of that stack is then the responsibility of user-space application code. If a thread encounters a page fault, the user-space software exception handler has a chance to react; otherwise, the kernel's default policy will kill the thread. In the case of a page fault, the `cause` field of the ureg structure will be 14 (`SWEXN_CAUSE_PAGEFAULT`), the `eip` field will contain the address of the faulting instruction, the `cr2` field will contain the memory address which resulted in the fault, and the `error_code` field will contain the reason why that memory address was inaccessible (see Intel's documentation of the page-fault exception for details).

It is convenient to be able to run old-fashioned single-threaded C programs (e.g., cat, gcc) which expect automatic stack growth as part of a typical POSIX run-time environment. Naturally, these programs do not understand Pebbles software exception handlers. Therefore, one of your deliverables will be a piece of code which handles page faults and attempts to grow the stack as appropriate.

## 3 The System Call Interface

### 3.1 Invocation and Return

User code will make requests of the kernel by issuing a trap instruction (which Intel calls a "software interrupt") using the `INT` instruction. Interrupt numbers are defined in `spec/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 trap, as defined in `spec/syscall_int.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, with subsequent parameters occupying higher memory addresses, and place the *address* of the packet in `%esi`. The diagram below shows a system call packet for the `readline()` system call.

| | |
|---|---|
| 4(%esi) | buf |
| (%esi) | len |

When the system call completes, the return value, if any, will be available in the `%eax` register. Other registers are expected to be unchanged unless the specification of a particular system call documents specific changes.

### 3.2 Semantics of the System Call Interface

A Pebbles 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.

No action taken by user code should *ever* cause the kernel to crash, hang, or otherwise fail to perform its job.

Execution of a thread terminates in *exactly* these circumstances:

- The thread invokes the `vanish()` sytem call from user space,
- Some thread in the containing task invokes the `task_vanish()` system call from user space,
- The thread executes, in user space, without a registered software exception handler, an instruction it should not be able to execute (see the specification of `vanish()` below).

No part of the kernel may end the execution of a thread for any reason other than the three reasons listed above.

Many system calls have the property that there are multiple illegal invocations. For example, the `readfile()` system call takes a pointer parameter and a length parameter; for any given invocation of the system call, either parameter or both might be invalid. The kernel is allowed to carry out validity checks in any order which is convenient for it. In some situations, a validity check can be carried out "early" (before the kernel does a substantial amount of work related to a system call) or "late" (after some work has been done, perhaps including side effects visible to user code). In general, both "early" and "late" checks for validity are legal, as long as the way the system call invocation fails matches the description of the system call in a reasonable way.

## 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 `spec/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 a Pebbles 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 two 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. The exit status (see below) of a newly-created task is 0. If a thread in the task invoking `fork()` has a software exception handler registered, the corresponding thread in the newly-created task will have exactly the same handler registered.

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

  Some Pebbles 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 2.1). 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. A thread newly created by `thread_fork` has no software exception handler registered.

  Threads are runnable as soon as they are created.

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

  Some Pebbles 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 program where they will serve as the first and second arguments of the the new program's `main()`, respectively. Before the new program begins, `%EIP` will be set to the "entry point" (the first instruction of the `main()` wrapper, as advertised by the ELF linker). The stack pointer, `%ESP`, will be initialized appropriately so that the `main()` wrapper receives four parameters:

  1. `int argc` - count of strings in `argv`
  2. `char *argv[]` - argument-string vector
  3. `void *stack_high` - highest legal (byte) address of the initial stack
  4. `void *stack_low` - lowest legal (byte) address of the initial stack

  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.

After a successful `exec()` the thread that begins execution of the new program has no software exception handler registered.

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

- `void set_status(int status)` - Sets the exit status of the current task to `status`.

- `void vanish(void)` - 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 exit status of the task available to the parent task (the task which created this task using `fork()`) via `wait()`. If the parent task is no longer running, the exit status of the task is made available to the kernel-launched "init" task instead. The statuses of any child tasks that have not been collected via `wait()` should also be made available to the kernel-launched "init" task.

  If the kernel decides to kill a thread, the effect should be as follows:

  - The kernel should display an appropriate message on the console, generally including the reason the thread was killed and a register dump,
  - If the thread is the sole thread in its task, the kernel should do the equivalent of `set_status(-2)`,
  - The kernel should perform the equivalent of `vanish()` on behalf of the thread.

  The `vanish()` 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 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 *original* thread of the exiting task, *not* the thread ID of the last thread in that task to `vanish()`. This should make sense if you consider how `fork()` and `wait()` interact.

  The `wait()` system call may be invoked simultaneously by any number of threads in a task; exited child tasks may be matched to `wait()`'ing threads in any non-pathological way. Threads which cannot collect an already-exited child task when there exist child tasks which have not yet exited will generally block until a child task exits and collect the status of an exited child task. However, threads which will definitely not be able to collect the status of an exited child task in the future must not block forever; in that case, `wait()` will return an integer error code less than zero.

  The invoking thread may specify a `status_ptr` parameter of zero (`NULL`) to indicate that it wishes to collect the ID of an exited task but wishes to ignore the exit status of that task. Otherwise, if the `status_ptr` parameter does not refer to writable memory, `wait()` will return an integer error code less than zero instead of collecting a child task.

- `void task_vanish(int status)` - Causes all threads of a task to `vanish()`. The exit status of the task, as returned via `wait()`, will be the value of the `status` parameter.

The threads must `vanish()` "in a timely fashion," meaning that it is *not* ok for `task_vanish()` 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. Ideally, 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` does not exist, is awaiting an external event in a system call such as readline() or wait(), or has been suspended via a system call, 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 the stated effects of executing `make_runnable()` on this thread.
- `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()`.

  This system call is *atomic* with respect to `deschedule()`: the process of determining whether the target thread is non-runnable due to a call to `deschedule()` and making it runnable will not be interleaved with the stated effects of the target thread executing `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.
- `typedef struct ureg_t { // see Figure 1 } ureg_t`
  `typedef void (*swexn_handler_t)(void *arg, ureg_t *ureg)`
  `int swexn(void *esp3, swexn_handler_t eip, void *arg, ureg_t *newureg)` -

  If `esp3` and/or `eip` are zero, de-register an exception handler if one is currently registered.

  If both `esp3` and `eip` are non-zero, attempt to register a software exception handler. The parameter `esp3` specifies an exception stack; it points to an address one word higher than the first address that the kernel should use to push values onto the exception stack. The parameter `eip` points to the first instruction of the handler function.

Whether or not a handler is being registered or de-registered, if `newureg` is non-zero, the kernel is requested to adopt the specified register values (including `%EIP`!) before the `swexn()` system call returns to user space (see `syscall.h` for a description of the `ureg_t` structure and the register values it contains).

If a single invocation of the `swexn()` system call attempts to register or de-register a handler, and also attempts to specify new register values, and either request cannot be carried out, neither request will be.

If the invocation is invalid (e.g., the kernel is unable to obtain a complete set of registers by dereferencing a non-zero `newureg` pointer), an error code less than zero is returned (and no change is made to handler registration or register values).

The kernel **MUST** ensure that it does not allow a thread to assume register values which are unsafe in the sense of allowing the thread to crash the kernel. However, if a thread specifies `newureg` register values that will cause *the thread* to "crash," that is not the kernel's responsibility.

The kernel should also reject `eip` and `esp3` values which are wrong at the time of invocation. On the other hand, the kernel is not responsible for ensuring that values of those parameters which "appear reasonable" will always in the future lead to satisfactory execution for the thread.

It is not an error to register a new handler if one was previously registered or to de-register a handler when one was not registered.

When a software exception handler function (`swexn_handler_t`) begins running, it will be invoked via a stack frame which specifies two parameters, an opaque `void*` which was specified when the handler was registered, and a pointer to a `ureg` area, which will be stored on the exception stack. The return address of the function will be some invalid address. Before the first instruction of the handler is run, the handler is automatically de-registered by the kernel.

The kernel should ensure when it invokes a software exception handler that the register values are sufficiently reasonable that the handler can run, assuming a reasonable handler was registered. Also, registers whose value is genuinely undefined when the handler is launched should be blanked to zero.

## 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 represents memory already in the task's address space, if any portion of the region intersects a part of the address space reserved by the kernel,[1] 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.

[1]Kernels are expected not to make tasteless reservations, e.g., the 18th megabyte.

- `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

- `int 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.

  If the return code is zero or greater the low-order eight bits contain a character; otherwise, a negative integer failure code is returned.

- `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 Pebbles 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()`.

  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 not continue 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.

[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.

- `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 integers addressed by the two arguments. If either argument is invalid, an error code less than zero is returned and the values of *both* integers are undefined. Zero is returned on success.

## 4.7 Miscellaneous System Interaction

- `int readfile(char *filename, char *buf, int count, int offset)` - Attempts to fill the user-specified buffer `buf` with `count` bytes starting `offset` bytes from the beginning of the RAM disk file specified by `filename`. If there are fewer than `count` bytes left in the file starting at `offset`, then as many bytes as remain are copied.

  Returns an error code less than zero if no file with the given name exists, `count` is negative, `offset` is less than zero or greater than the size of the file, or `buf` is not a valid buffer large enough to store `count` bytes. In this case, the contents of `buf` are undefined. Otherwise, the number of bytes stored into the buffer is returned (note that this value may be zero in some cases).

  It is conventional that a file named "." exists which contains a list of the files which `readfile()` can access. Each entry in this file is null-terminated, and there is an extra null byte after the last filename's terminating null.

- `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.