Pebbles Kernel Specification

Spring 2018

Contents

1 Introduction

This document defines the correct behavior of the Pebbles kernel for the Spring 2018 edition of CMPSC473.

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. You can think of a task as a similar concept to a Linux process. A task's resources include various memory regions and "invisible" kernel resources (such as a queue of task-exit notifications).

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."

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.

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.

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.

3.2 Semantics of the System Call Interface

The 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.

A thread may be terminated *only* if:

- The thread invokes the `vanish()` sytem call.
- Some thread in the containing task invokes the `task_vanish()` system call.
- The thread receives a hardware exception due to executing an instruction it should not be able to execute and has no software exception handler registered (see the specification of `vanish()` below).

In *no other circumstances* may the kernel end the execution of a thread.

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.

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.

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 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 you 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`. Think carefully about what sort of function you want to expose, what parameters it should take, and what would be a good name for this function.

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 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.

-
  ```
  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;
  ```

  `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 "clearly 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.

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

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

- `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.

[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_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 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.

- `void misbehave(int mode)` - Causes the kernel to schedule in peculiar ways for testing purposes. You should not be using this syscall. You only need to create a syscall stub for this.