Appendix A Reference Guide

This chapter is a reference for the Pintos code. The reference guide does not cover all of the code in Pintos, but it does cover those pieces that students most often find troublesome. You may find that you want to read each part of the reference guide as you work on the project where it becomes important.

We recommend using "tags" to follow along with references to function and variable names (see ⟨undefined⟩ [Tags], page ⟨undefined⟩).

A.1 Loading

This section covers the Pintos loader and basic kernel initialization.

A.1.1 The Loader

The first part of Pintos that runs is the loader, in '`threads/loader.S`'. The PC BIOS loads the loader into memory. The loader, in turn, is responsible for finding the kernel on disk, loading it into memory, and then jumping to its start. It's not important to understand exactly how the loader works, but if you're interested, read on. You should probably read along with the loader's source. You should also understand the basics of the 80x86 architecture as described by chapter 3, "Basic Execution Environment," of [IA32-v1].

The PC BIOS loads the loader from the first sector of the first hard disk, called the *master boot record* (MBR). PC conventions reserve 64 bytes of the MBR for the partition table, and Pintos uses about 128 additional bytes for kernel command-line arguments. This leaves a little over 300 bytes for the loader's own code. This is a severe restriction that means, practically speaking, the loader must be written in assembly language.

The Pintos loader and kernel don't have to be on the same disk, nor does is the kernel required to be in any particular location on a given disk. The loader's first job, then, is to find the kernel by reading the partition table on each hard disk, looking for a bootable partition of the type used for a Pintos kernel.

When the loader finds a bootable kernel partition, it reads the partition's contents into memory at physical address 128 kB. The kernel is at the beginning of the partition, which might be larger than necessary due to partition boundary alignment conventions, so the loader reads no more than 512 kB (and the Pintos build process will refuse to produce kernels larger than that). Reading more data than this would cross into the region from 640 kB to 1 MB that the PC architecture reserves for hardware and the BIOS, and a standard PC BIOS does not provide any means to load the kernel above 1 MB.

The loader's final job is to extract the entry point from the loaded kernel image and transfer control to it. The entry point is not at a predictable location, but the kernel's ELF header contains a pointer to it. The loader extracts the pointer and jumps to the location it points to.

The Pintos kernel command line is stored in the boot loader. The `pintos` program actually modifies a copy of the boot loader on disk each time it runs the kernel, inserting whatever command-line arguments the user supplies to the kernel, and then the kernel at boot time reads those arguments out of the boot loader in memory. This is not an elegant solution, but it is simple and effective.

A.1.2 Low-Level Kernel Initialization

The loader's last action is to transfer control to the kernel's entry point, which is `start()` in '`threads/start.S`'. The job of this code is to switch the CPU from legacy 16-bit "real mode" into the 32-bit "protected mode" used by all modern 80x86 operating systems.

The startup code's first task is actually to obtain the machine's memory size, by asking the BIOS for the PC's memory size. The simplest BIOS function to do this can only detect up to 64 MB of RAM, so that's the practical limit that Pintos can support. The function stores the memory size, in pages, in global variable `init_ram_pages`.

The first part of CPU initialization is to enable the A20 line, that is, the CPU's address line numbered 20. For historical reasons, PCs boot with this address line fixed at 0, which means that attempts to access memory beyond the first 1 MB (2 raised to the 20th power) will fail. Pintos wants to access more memory than this, so we have to enable it.

Next, the loader creates a basic page table. This page table maps the 64 MB at the base of virtual memory (starting at virtual address 0) directly to the identical physical addresses. It also maps the same physical memory starting at virtual address `LOADER_PHYS_BASE`, which defaults to `0xc0000000` (3 GB). The Pintos kernel only wants the latter mapping, but there's a chicken-and-egg problem if we don't include the former: our current virtual address is roughly `0x20000`, the location where the loader put us, and we can't jump to `0xc0020000` until we turn on the page table, but if we turn on the page table without jumping there, then we've just pulled the rug out from under ourselves.

After the page table is initialized, we load the CPU's control registers to turn on protected mode and paging, and set up the segment registers. We aren't yet equipped to handle interrupts in protected mode, so we disable interrupts. The final step is to call `main()`.

A.1.3 High-Level Kernel Initialization

The kernel proper starts with the `main()` function. The `main()` function is written in C, as will be most of the code we encounter in Pintos from here on out.

When `main()` starts, the system is in a pretty raw state. We're in 32-bit protected mode with paging enabled, but hardly anything else is ready. Thus, the `main()` function consists primarily of calls into other Pintos modules' initialization functions. These are usually named *`module`*`_init()`, where *module* is the module's name, '*`module`*`.c`' is the module's source code, and '*`module`*`.h`' is the module's header.

The first step in `main()` is to call `bss_init()`, which clears out the kernel's "BSS", which is the traditional name for a segment that should be initialized to all zeros. In most C implementations, whenever you declare a variable outside a function without providing an initializer, that variable goes into the BSS. Because it's all zeros, the BSS isn't stored in the image that the loader brought into memory. We just use `memset()` to zero it out.

Next, `main()` calls `read_command_line()` to break the kernel command line into arguments, then `parse_options()` to read any options at the beginning of the command line. (Actions specified on the command line execute later.)

`thread_init()` initializes the thread system. We will defer full discussion to our discussion of Pintos threads below. It is called so early in initialization because a valid thread structure is a prerequisite for acquiring a lock, and lock acquisition in turn is important to other Pintos subsystems. Then we initialize the console and print a startup message to the console.

The next block of functions we call initializes the kernel's memory system. `palloc_init()` sets up the kernel page allocator, which doles out memory one or more pages at a time (see Section A.5.1 [Page Allocator], page 38). `malloc_init()` sets up the allocator that handles allocations of arbitrary-size blocks of memory (see Section A.5.2 [Block Allocator], page 40). `paging_init()` sets up a page table for the kernel (see Section A.7 [Page Table], page 42).

In projects 2 and later, `main()` also calls `tss_init()` and `gdt_init()`.

The next set of calls initializes the interrupt system. `intr_init()` sets up the CPU's *interrupt descriptor table* (IDT) to ready it for interrupt handling (see Section A.4.1 [Interrupt Infrastructure], page 35), then `timer_init()` and `kbd_init()` prepare for handling timer interrupts and keyboard interrupts, respectively. `input_init()` sets up to merge serial and keyboard input into one stream. In projects 2 and later, we also prepare to handle interrupts caused by user programs using `exception_init()` and `syscall_init()`.

Now that interrupts are set up, we can start the scheduler with `thread_start()`, which creates the idle thread and enables interrupts. With interrupts enabled, interrupt-driven serial port I/O becomes possible, so we use `serial_init_queue()` to switch to that mode. Finally, `timer_calibrate()` calibrates the timer for accurate short delays.

If the file system is compiled in, as it will starting in project 2, we initialize the IDE disks with `ide_init()`, then the file system with `filesys_init()`.

Boot is complete, so we print a message.

Function `run_actions()` now parses and executes actions specified on the kernel command line, such as `run` to run a test (in project 1) or a user program (in later projects).

Finally, if '`-q`' was specified on the kernel command line, we call `shutdown_power_off()` to terminate the machine simulator. Otherwise, `main()` calls `thread_exit()`, which allows any other running threads to continue running.

A.1.4 Physical Memory Map

Memory Range	**Owner**	**Contents**
`00000000–000003ff`	CPU	Real mode interrupt table.
`00000400–000005ff`	BIOS	Miscellaneous data area.
`00000600–00007bff`	—	—
`00007c00–00007dff`	Pintos	Loader.
`0000e000–0000efff`	Pintos	Stack for loader; kernel stack and `struct thread` for initial kernel thread.
`0000f000–0000ffff`	Pintos	Page directory for startup code.
`00010000–00020000`	Pintos	Page tables for startup code.
`00020000–0009ffff`	Pintos	Kernel code, data, and uninitialized data segments.
`000a0000–000bffff`	Video	VGA display memory.
`000c0000–000effff`	Hardware	Reserved for expansion card RAM and ROM.
`000f0000–000fffff`	BIOS	ROM BIOS.

`00100000–03ffffff`	Pintos	Dynamic memory allocation.

A.2 Threads

A.2.1 `struct thread`

The main Pintos data structure for threads is `struct thread`, declared in ‘`threads/thread.h`’.

`struct thread` [Structure]

Represents a thread or a user process. In the projects, you will have to add your own members to `struct thread`. You may also change or delete the definitions of existing members.

Every `struct thread` occupies the beginning of its own page of memory. The rest of the page is used for the thread's stack, which grows downward from the end of the page. It looks like this:

```
                  4 kB +---------------------------------+
                       |          kernel stack           |
                       |                |                |
                       |                |                |
                       |                V                |
                       |         grows downward          |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
sizeof (struct thread) +---------------------------------+
                       |              magic              |
                       |                :                |
                       |                :                |
                       |              status             |
                       |               tid               |
                  0 kB +---------------------------------+
```

This has two consequences. First, `struct thread` must not be allowed to grow too big. If it does, then there will not be enough room for the kernel stack. The base `struct thread` is only a few bytes in size. It probably should stay well under 1 kB.

Second, kernel stacks must not be allowed to grow too large. If a stack overflows, it will corrupt the thread state. Thus, kernel functions should not allocate large structures or arrays as non-static local variables. Use dynamic allocation with `malloc()` or `palloc_get_page()` instead (see Section A.5 [Memory Allocation], page 38).

`tid_t tid` [Member of `struct thread`]

The thread's thread identifier or *tid*. Every thread must have a tid that is unique over the entire lifetime of the kernel. By default, `tid_t` is a `typedef` for `int` and each new thread receives the numerically next higher tid, starting from 1 for the initial process. You can change the type and the numbering scheme if you like.

`enum thread_status status` [Member of `struct thread`]

The thread's state, one of the following:

`THREAD_RUNNING` [Thread State]

The thread is running. Exactly one thread is running at a given time. `thread_current()` returns the running thread.

`THREAD_READY` [Thread State]

The thread is ready to run, but it's not running right now. The thread could be selected to run the next time the scheduler is invoked. Ready threads are kept in a doubly linked list called `ready_list`.

`THREAD_BLOCKED` [Thread State]

The thread is waiting for something, e.g. a lock to become available, an interrupt to be invoked. The thread won't be scheduled again until it transitions to the `THREAD_READY` state with a call to `thread_unblock()`. This is most conveniently done indirectly, using one of the Pintos synchronization primitives that block and unblock threads automatically (see Section A.3 [Synchronization], page 29).

There is no *a priori* way to tell what a blocked thread is waiting for, but a backtrace can help (see Section B.4 [Backtraces], page 55).

`THREAD_DYING` [Thread State]

The thread will be destroyed by the scheduler after switching to the next thread.

`char name[16]` [Member of `struct thread`]

The thread's name as a string, or at least the first few characters of it.

`uint8_t * stack` [Member of `struct thread`]

Every thread has its own stack to keep track of its state. When the thread is running, the CPU's stack pointer register tracks the top of the stack and this member is unused. But when the CPU switches to another thread, this member saves the thread's stack pointer. No other members are needed to save the thread's registers, because the other registers that must be saved are saved on the stack.

When an interrupt occurs, whether in the kernel or a user program, an `struct intr_frame` is pushed onto the stack. When the interrupt occurs in a user program, the `struct intr_frame` is always at the very top of the page. See Section A.4 [Interrupt Handling], page 35, for more information.

`int priority` [Member of `struct thread`]

A thread priority, ranging from `PRI_MIN` (0) to `PRI_MAX` (63). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. Pintos as provided ignores thread priorities, but you will implement priority scheduling in project 1 (see ⟨undefined⟩ [Priority Scheduling], page ⟨undefined⟩).

`struct list_elem allelem` [Member of `struct thread`]

This "list element" is used to link the thread into the list of all threads. Each thread is inserted into this list when it is created and removed when it exits. The `thread_foreach()` function should be used to iterate over all threads.

`struct list_elem elem` [Member of `struct thread`]

A "list element" used to put the thread into doubly linked lists, either `ready_list` (the list of threads ready to run) or a list of threads waiting on a semaphore in `sema_down()`. It can do double duty because a thread waiting on a semaphore is not ready, and vice versa.

`uint32_t * pagedir` [Member of `struct thread`]

Only present in project 2 and later. See ⟨undefined⟩ [Page Tables], page ⟨undefined⟩.

`unsigned magic` [Member of `struct thread`]

Always set to `THREAD_MAGIC`, which is just an arbitrary number defined in '`threads/thread.c`', and used to detect stack overflow. `thread_current()` checks that the `magic` member of the running thread's `struct thread` is set to `THREAD_MAGIC`. Stack overflow tends to change this value, triggering the assertion. For greatest benefit, as you add members to `struct thread`, leave `magic` at the end.

A.2.2 Thread Functions

'`threads/thread.c`' implements several public functions for thread support. Let's take a look at the most useful:

`void thread_init` (*void*) [Function]

Called by `main()` to initialize the thread system. Its main purpose is to create a `struct thread` for Pintos's initial thread. This is possible because the Pintos loader puts the initial thread's stack at the top of a page, in the same position as any other Pintos thread.

Before `thread_init()` runs, `thread_current()` will fail because the running thread's `magic` value is incorrect. Lots of functions call `thread_current()` directly or indirectly, including `lock_acquire()` for locking a lock, so `thread_init()` is called early in Pintos initialization.

`void thread_start` (*void*) [Function]

Called by `main()` to start the scheduler. Creates the idle thread, that is, the thread that is scheduled when no other thread is ready. Then enables interrupts, which as a side effect enables the scheduler because the scheduler runs on return from the timer interrupt, using `intr_yield_on_return()` (see Section A.4.3 [External Interrupt Handling], page 37).

`void thread_tick` (*void*) [Function]

Called by the timer interrupt at each timer tick. It keeps track of thread statistics and triggers the scheduler when a time slice expires.

`void thread_print_stats` (*void*) [Function]

Called during Pintos shutdown to print thread statistics.

`tid_t thread_create` (*const char ****name***, *int* ***priority***, *thread_func ****func***, *void ****aux***) [Function]

Creates and starts a new thread named *name* with the given *priority*, returning the new thread's tid. The thread executes *func*, passing *aux* as the function's single argument.

`thread_create()` allocates a page for the thread's `struct thread` and stack and initializes its members, then it sets up a set of fake stack frames for it (see Section A.2.3 [Thread Switching], page 28). The thread is initialized in the blocked state, then unblocked just before returning, which allows the new thread to be scheduled (see [Thread States], page 25).

`void thread_func (void *aux)` [Type]

This is the type of the function passed to `thread_create()`, whose *aux* argument is passed along as the function's argument.

`void thread_block` (*void*) [Function]

Transitions the running thread from the running state to the blocked state (see [Thread States], page 25). The thread will not run again until `thread_unblock()` is called on it, so you'd better have some way arranged for that to happen. Because `thread_block()` is so low-level, you should prefer to use one of the synchronization primitives instead (see Section A.3 [Synchronization], page 29).

`void thread_unblock` (*struct thread ****thread***) [Function]

Transitions *thread*, which must be in the blocked state, to the ready state, allowing it to resume running (see [Thread States], page 25). This is called when the event that the thread is waiting for occurs, e.g. when the lock that the thread is waiting on becomes available.

`struct thread * thread_current` (*void*) [Function]

Returns the running thread.

`tid_t thread_tid` (*void*) [Function]

Returns the running thread's thread id. Equivalent to `thread_current ()->tid`.

`const char * thread_name` (*void*) [Function]

Returns the running thread's name. Equivalent to `thread_current ()->name`.

`void thread_exit` (*void*) `NO_RETURN` [Function]

Causes the current thread to exit. Never returns, hence `NO_RETURN` (see Section B.3 [Function and Parameter Attributes], page 54).

`void thread_yield` (*void*) [Function]

Yields the CPU to the scheduler, which picks a new thread to run. The new thread might be the current thread, so you can't depend on this function to keep this thread from running for any particular length of time.

`void thread_foreach` (*thread_action_func ****action***, *void ****aux***) [Function]

Iterates over all threads *t* and invokes `action(t, aux)` on each. *action* must refer to a function that matches the signature given by `thread_action_func()`:

`void thread_action_func (struct thread *thread, void *aux)` [Type]

Performs some action on a thread, given *aux*.

`int thread_get_priority` (*void*) [Function]
`void thread_set_priority` (*int* `new_priority`) [Function]

Stub to set and get thread priority. See ⟨undefined⟩ [Priority Scheduling], page ⟨undefined⟩.

`int thread_get_nice` (*void*) [Function]
`void thread_set_nice` (*int* `new_nice`) [Function]
`int thread_get_recent_cpu` (*void*) [Function]
`int thread_get_load_avg` (*void*) [Function]

Stubs for the advanced scheduler. See ⟨undefined⟩ [4.4BSD Scheduler], page ⟨undefined⟩.

A.2.3 Thread Switching

`schedule()` is responsible for switching threads. It is internal to '`threads/thread.c`' and called only by the three public thread functions that need to switch threads: `thread_block()`, `thread_exit()`, and `thread_yield()`. Before any of these functions call `schedule()`, they disable interrupts (or ensure that they are already disabled) and then change the running thread's state to something other than running.

`schedule()` is short but tricky. It records the current thread in local variable *cur*, determines the next thread to run as local variable *next* (by calling `next_thread_to_run()`), and then calls `switch_threads()` to do the actual thread switch. The thread we switched to was also running inside `switch_threads()`, as are all the threads not currently running, so the new thread now returns out of `switch_threads()`, returning the previously running thread.

`switch_threads()` is an assembly language routine in '`threads/switch.S`'. It saves registers on the stack, saves the CPU's current stack pointer in the current `struct thread`'s `stack` member, restores the new thread's `stack` into the CPU's stack pointer, restores registers from the stack, and returns.

The rest of the scheduler is implemented in `thread_schedule_tail()`. It marks the new thread as running. If the thread we just switched from is in the dying state, then it also frees the page that contained the dying thread's `struct thread` and stack. These couldn't be freed prior to the thread switch because the switch needed to use it.

Running a thread for the first time is a special case. When `thread_create()` creates a new thread, it goes through a fair amount of trouble to get it started properly. In particular, the new thread hasn't started running yet, so there's no way for it to be running inside `switch_threads()` as the scheduler expects. To solve the problem, `thread_create()` creates some fake stack frames in the new thread's stack:

- The topmost fake stack frame is for `switch_threads()`, represented by `struct switch_threads_frame`. The important part of this frame is its `eip` member, the return address. We point `eip` to `switch_entry()`, indicating it to be the function that called `switch_entry()`.

- The next fake stack frame is for `switch_entry()`, an assembly language routine in '`threads/switch.S`' that adjusts the stack pointer,[1] calls `thread_schedule_tail()` (this special case is why `thread_schedule_tail()` is separate from `schedule()`), and returns. We fill in its stack frame so that it returns into `kernel_thread()`, a function in '`threads/thread.c`'.
- The final stack frame is for `kernel_thread()`, which enables interrupts and calls the thread's function (the function passed to `thread_create()`). If the thread's function returns, it calls `thread_exit()` to terminate the thread.

A.3 Synchronization

If sharing of resources between threads is not handled in a careful, controlled fashion, the result is usually a big mess. This is especially the case in operating system kernels, where faulty sharing can crash the entire machine. Pintos provides several synchronization primitives to help out.

A.3.1 Disabling Interrupts

The crudest way to do synchronization is to disable interrupts, that is, to temporarily prevent the CPU from responding to interrupts. If interrupts are off, no other thread will preempt the running thread, because thread preemption is driven by the timer interrupt. If interrupts are on, as they normally are, then the running thread may be preempted by another at any time, whether between two C statements or even within the execution of one.

Incidentally, this means that Pintos is a "preemptible kernel," that is, kernel threads can be preempted at any time. Traditional Unix systems are "nonpreemptible," that is, kernel threads can only be preempted at points where they explicitly call into the scheduler. (User programs can be preempted at any time in both models.) As you might imagine, preemptible kernels require more explicit synchronization.

You should have little need to set the interrupt state directly. Most of the time you should use the other synchronization primitives described in the following sections. The main reason to disable interrupts is to synchronize kernel threads with external interrupt handlers, which cannot sleep and thus cannot use most other forms of synchronization (see Section A.4.3 [External Interrupt Handling], page 37).

Some external interrupts cannot be postponed, even by disabling interrupts. These interrupts, called *non-maskable interrupts* (NMIs), are supposed to be used only in emergencies, e.g. when the computer is on fire. Pintos does not handle non-maskable interrupts.

Types and functions for disabling and enabling interrupts are in '`threads/interrupt.h`'.

`enum intr_level` [Type]

> One of `INTR_OFF` or `INTR_ON`, denoting that interrupts are disabled or enabled, respectively.

`enum intr_level intr_get_level` (*void*) [Function]

> Returns the current interrupt state.

[1] This is because `switch_threads()` takes arguments on the stack and the 80x86 SVR4 calling convention requires the caller, not the called function, to remove them when the call is complete. See [SysV-i386] chapter 3 for details.

`enum intr_level intr_set_level` (*enum intr_level* `level`) [Function]

Turns interrupts on or off according to *level*. Returns the previous interrupt state.

`enum intr_level intr_enable` (*void*) [Function]

Turns interrupts on. Returns the previous interrupt state.

`enum intr_level intr_disable` (*void*) [Function]

Turns interrupts off. Returns the previous interrupt state.

A.3.2 Semaphores

A *semaphore* is a nonnegative integer together with two operators that manipulate it atomically, which are:

- "Down" or "P": wait for the value to become positive, then decrement it.
- "Up" or "V": increment the value (and wake up one waiting thread, if any).

A semaphore initialized to 0 may be used to wait for an event that will happen exactly once. For example, suppose thread A starts another thread B and wants to wait for B to signal that some activity is complete. A can create a semaphore initialized to 0, pass it to B as it starts it, and then "down" the semaphore. When B finishes its activity, it "ups" the semaphore. This works regardless of whether A "downs" the semaphore or B "ups" it first.

A semaphore initialized to 1 is typically used for controlling access to a resource. Before a block of code starts using the resource, it "downs" the semaphore, then after it is done with the resource it "ups" the resource. In such a case a lock, described below, may be more appropriate.

Semaphores can also be initialized to values larger than 1. These are rarely used.

Semaphores were invented by Edsger Dijkstra and first used in the THE operating system ([Dijkstra]).

Pintos' semaphore type and operations are declared in '`threads/synch.h`'.

`struct semaphore` [Type]

Represents a semaphore.

`void sema_init` (*struct semaphore* *`sema`, *unsigned* `value`) [Function]

Initializes *sema* as a new semaphore with the given initial *value*.

`void sema_down` (*struct semaphore* *`sema`) [Function]

Executes the "down" or "P" operation on *sema*, waiting for its value to become positive and then decrementing it by one.

`bool sema_try_down` (*struct semaphore* *`sema`) [Function]

Tries to execute the "down" or "P" operation on *sema*, without waiting. Returns true if *sema* was successfully decremented, or false if it was already zero and thus could not be decremented without waiting. Calling this function in a tight loop wastes CPU time, so use `sema_down()` or find a different approach instead.

`void sema_up (`*struct semaphore* `*sema)` [Function]

Executes the "up" or "V" operation on *sema*, incrementing its value. If any threads are waiting on *sema*, wakes one of them up.

Unlike most synchronization primitives, `sema_up()` may be called inside an external interrupt handler (see Section A.4.3 [External Interrupt Handling], page 37).

Semaphores are internally built out of disabling interrupt (see Section A.3.1 [Disabling Interrupts], page 29) and thread blocking and unblocking (`thread_block()` and `thread_unblock()`). Each semaphore maintains a list of waiting threads, using the linked list implementation in '`lib/kernel/list.c`'.

A.3.3 Locks

A *lock* is like a semaphore with an initial value of 1 (see Section A.3.2 [Semaphores], page 30). A lock's equivalent of "up" is called "release", and the "down" operation is called "acquire".

Compared to a semaphore, a lock has one added restriction: only the thread that acquires a lock, called the lock's "owner", is allowed to release it. If this restriction is a problem, it's a good sign that a semaphore should be used, instead of a lock.

Locks in Pintos are not "recursive," that is, it is an error for the thread currently holding a lock to try to acquire that lock.

Lock types and functions are declared in '`threads/synch.h`'.

`struct lock` [Type]

Represents a lock.

`void lock_init (`*struct lock* `*lock)` [Function]

Initializes *lock* as a new lock. The lock is not initially owned by any thread.

`void lock_acquire (`*struct lock* `*lock)` [Function]

Acquires *lock* for the current thread, first waiting for any current owner to release it if necessary.

`bool lock_try_acquire (`*struct lock* `*lock)` [Function]

Tries to acquire *lock* for use by the current thread, without waiting. Returns true if successful, false if the lock is already owned. Calling this function in a tight loop is a bad idea because it wastes CPU time, so use `lock_acquire()` instead.

`void lock_release (`*struct lock* `*lock)` [Function]

Releases *lock*, which the current thread must own.

`bool lock_held_by_current_thread (`*const struct lock* `*lock)` [Function]

Returns true if the running thread owns *lock*, false otherwise. There is no function to test whether an arbitrary thread owns a lock, because the answer could change before the caller could act on it.

A.3.4 Monitors

A *monitor* is a higher-level form of synchronization than a semaphore or a lock. A monitor consists of data being synchronized, plus a lock, called the *monitor lock*, and one or more *condition variables*. Before it accesses the protected data, a thread first acquires the monitor lock. It is then said to be "in the monitor". While in the monitor, the thread has control over all the protected data, which it may freely examine or modify. When access to the protected data is complete, it releases the monitor lock.

Condition variables allow code in the monitor to wait for a condition to become true. Each condition variable is associated with an abstract condition, e.g. "some data has arrived for processing" or "over 10 seconds has passed since the user's last keystroke". When code in the monitor needs to wait for a condition to become true, it "waits" on the associated condition variable, which releases the lock and waits for the condition to be signaled. If, on the other hand, it has caused one of these conditions to become true, it "signals" the condition to wake up one waiter, or "broadcasts" the condition to wake all of them.

The theoretical framework for monitors was laid out by C. A. R. Hoare ([Hoare]). Their practical usage was later elaborated in a paper on the Mesa operating system ([Lampson]).

Condition variable types and functions are declared in '`threads/synch.h`'.

`struct condition` [Type]

Represents a condition variable.

`void cond_init` (*struct condition* *`cond`) [Function]

Initializes *cond* as a new condition variable.

`void cond_wait` (*struct condition* *`cond`, *struct lock* *`lock`) [Function]

Atomically releases *lock* (the monitor lock) and waits for *cond* to be signaled by some other piece of code. After *cond* is signaled, reacquires *lock* before returning. *lock* must be held before calling this function.

Sending a signal and waking up from a wait are not an atomic operation. Thus, typically `cond_wait()`'s caller must recheck the condition after the wait completes and, if necessary, wait again. See the next section for an example.

`void cond_signal` (*struct condition* *`cond`, *struct lock* *`lock`) [Function]

If any threads are waiting on *cond* (protected by monitor lock *lock*), then this function wakes up one of them. If no threads are waiting, returns without performing any action. *lock* must be held before calling this function.

`void cond_broadcast` (*struct condition* *`cond`, *struct lock* *`lock`) [Function]

Wakes up all threads, if any, waiting on *cond* (protected by monitor lock *lock*). *lock* must be held before calling this function.

A.3.4.1 Monitor Example

The classical example of a monitor is handling a buffer into which one or more "producer" threads write characters and out of which one or more "consumer" threads read characters. To implement this we need, besides the monitor lock, two condition variables which we will call *not_full* and *not_empty*:

```
char buf[BUF_SIZE];      /* Buffer. */
size_t n = 0;            /* 0 <= n <= BUF_SIZE: # of characters in buffer. */
size_t head = 0;         /* buf index of next char to write (mod BUF_SIZE). */
size_t tail = 0;         /* buf index of next char to read (mod BUF_SIZE). */
struct lock lock;        /* Monitor lock. */
struct condition not_empty; /* Signaled when the buffer is not empty. */
struct condition not_full; /* Signaled when the buffer is not full. */

...initialize the locks and condition variables...

void put (char ch) {
  lock_acquire (&lock);
  while (n == BUF_SIZE)            /* Can't add to buf as long as it's full. */
    cond_wait (&not_full, &lock);
  buf[head++ % BUF_SIZE] = ch;     /* Add ch to buf. */
  n++;
  cond_signal (&not_empty, &lock); /* buf can't be empty anymore. */
  lock_release (&lock);
}

char get (void) {
  char ch;
  lock_acquire (&lock);
  while (n == 0)                  /* Can't read buf as long as it's empty. */
    cond_wait (&not_empty, &lock);
  ch = buf[tail++ % BUF_SIZE];    /* Get ch from buf. */
  n--;
  cond_signal (&not_full, &lock); /* buf can't be full anymore. */
  lock_release (&lock);
}
```

Note that `BUF_SIZE` must divide evenly into `SIZE_MAX + 1` for the above code to be completely correct. Otherwise, it will fail the first time `head` wraps around to 0. In practice, `BUF_SIZE` would ordinarily be a power of 2.

A.3.5 Optimization Barriers

An *optimization barrier* is a special statement that prevents the compiler from making assumptions about the state of memory across the barrier. The compiler will not reorder reads or writes of variables across the barrier or assume that a variable's value is unmodified across the barrier, except for local variables whose address is never taken. In Pintos, '`threads/synch.h`' defines the `barrier()` macro as an optimization barrier.

One reason to use an optimization barrier is when data can change asynchronously, without the compiler's knowledge, e.g. by another thread or an interrupt handler. The `too_many_loops()` function in '`devices/timer.c`' is an example. This function starts out by busy-waiting in a loop until a timer tick occurs:

```
/* Wait for a timer tick. */
int64_t start = ticks;
```

```
while (ticks == start)
  barrier ();
```

Without an optimization barrier in the loop, the compiler could conclude that the loop would never terminate, because `start` and `ticks` start out equal and the loop itself never changes them. It could then "optimize" the function into an infinite loop, which would definitely be undesirable.

Optimization barriers can be used to avoid other compiler optimizations. The `busy_wait()` function, also in '`devices/timer.c`', is an example. It contains this loop:

```
while (loops-- > 0)
  barrier ();
```

The goal of this loop is to busy-wait by counting `loops` down from its original value to 0. Without the barrier, the compiler could delete the loop entirely, because it produces no useful output and has no side effects. The barrier forces the compiler to pretend that the loop body has an important effect.

Finally, optimization barriers can be used to force the ordering of memory reads or writes. For example, suppose we add a "feature" that, whenever a timer interrupt occurs, the character in global variable `timer_put_char` is printed on the console, but only if global Boolean variable `timer_do_put` is true. The best way to set up '`x`' to be printed is then to use an optimization barrier, like this:

```
timer_put_char = 'x';
barrier ();
timer_do_put = true;
```

Without the barrier, the code is buggy because the compiler is free to reorder operations when it doesn't see a reason to keep them in the same order. In this case, the compiler doesn't know that the order of assignments is important, so its optimizer is permitted to exchange their order. There's no telling whether it will actually do this, and it is possible that passing the compiler different optimization flags or using a different version of the compiler will produce different behavior.

Another solution is to disable interrupts around the assignments. This does not prevent reordering, but it prevents the interrupt handler from intervening between the assignments. It also has the extra runtime cost of disabling and re-enabling interrupts:

```
enum intr_level old_level = intr_disable ();
timer_put_char = 'x';
timer_do_put = true;
intr_set_level (old_level);
```

A second solution is to mark the declarations of `timer_put_char` and `timer_do_put` as '`volatile`'. This keyword tells the compiler that the variables are externally observable and restricts its latitude for optimization. However, the semantics of '`volatile`' are not well-defined, so it is not a good general solution. The base Pintos code does not use '`volatile`' at all.

The following is *not* a solution, because locks neither prevent interrupts nor prevent the compiler from reordering the code within the region where the lock is held:

```
lock_acquire (&timer_lock);     /* INCORRECT CODE */
timer_put_char = 'x';
```

```
    timer_do_put = true;
    lock_release (&timer_lock);
```

The compiler treats invocation of any function defined externally, that is, in another source file, as a limited form of optimization barrier. Specifically, the compiler assumes that any externally defined function may access any statically or dynamically allocated data and any local variable whose address is taken. This often means that explicit barriers can be omitted. It is one reason that Pintos contains few explicit barriers.

A function defined in the same source file, or in a header included by the source file, cannot be relied upon as a optimization barrier. This applies even to invocation of a function before its definition, because the compiler may read and parse the entire source file before performing optimization.

A.4 Interrupt Handling

An *interrupt* notifies the CPU of some event. Much of the work of an operating system relates to interrupts in one way or another. For our purposes, we classify interrupts into two broad categories:

- *Internal interrupts*, that is, interrupts caused directly by CPU instructions. System calls, attempts at invalid memory access (*page faults*), and attempts to divide by zero are some activities that cause internal interrupts. Because they are caused by CPU instructions, internal interrupts are *synchronous* or synchronized with CPU instructions. `intr_disable()` does not disable internal interrupts.
- *External interrupts*, that is, interrupts originating outside the CPU. These interrupts come from hardware devices such as the system timer, keyboard, serial ports, and disks. External interrupts are *asynchronous*, meaning that their delivery is not synchronized with instruction execution. Handling of external interrupts can be postponed with `intr_disable()` and related functions (see Section A.3.1 [Disabling Interrupts], page 29).

The CPU treats both classes of interrupts largely the same way, so Pintos has common infrastructure to handle both classes. The following section describes this common infrastructure. The sections after that give the specifics of external and internal interrupts.

If you haven't already read chapter 3, "Basic Execution Environment," in [IA32-v1], it is recommended that you do so now. You might also want to skim chapter 5, "Interrupt and Exception Handling," in [IA32-v3a].

A.4.1 Interrupt Infrastructure

When an interrupt occurs, the CPU saves its most essential state on a stack and jumps to an interrupt handler routine. The 80x86 architecture supports 256 interrupts, numbered 0 through 255, each with an independent handler defined in an array called the *interrupt descriptor table* or IDT.

In Pintos, `intr_init()` in '`threads/interrupt.c`' sets up the IDT so that each entry points to a unique entry point in '`threads/intr-stubs.S`' named `intr`*`NN`*`_stub()`, where *NN* is the interrupt number in hexadecimal. Because the CPU doesn't give us any other way to find out the interrupt number, this entry point pushes the interrupt number on the stack. Then it jumps to `intr_entry()`, which pushes all the registers that the processor didn't already push for us, and then calls `intr_handler()`, which brings us back into C in '`threads/interrupt.c`'.

The main job of `intr_handler()` is to call the function registered for handling the particular interrupt. (If no function is registered, it dumps some information to the console and panics.) It also does some extra processing for external interrupts (see Section A.4.3 [External Interrupt Handling], page 37).

When `intr_handler()` returns, the assembly code in '`threads/intr-stubs.S`' restores all the CPU registers saved earlier and directs the CPU to return from the interrupt.

The following types and functions are common to all interrupts.

`void intr_handler_func (struct intr_frame *`*`frame`*`)` [Type]

This is how an interrupt handler function must be declared. Its *frame* argument (see below) allows it to determine the cause of the interrupt and the state of the thread that was interrupted.

`struct intr_frame` [Type]

The stack frame of an interrupt handler, as saved by the CPU, the interrupt stubs, and `intr_entry()`. Its most interesting members are described below.

`uint32_t edi` [Member of `struct intr_frame`]
`uint32_t esi` [Member of `struct intr_frame`]
`uint32_t ebp` [Member of `struct intr_frame`]
`uint32_t esp_dummy` [Member of `struct intr_frame`]
`uint32_t ebx` [Member of `struct intr_frame`]
`uint32_t edx` [Member of `struct intr_frame`]
`uint32_t ecx` [Member of `struct intr_frame`]
`uint32_t eax` [Member of `struct intr_frame`]
`uint16_t es` [Member of `struct intr_frame`]
`uint16_t ds` [Member of `struct intr_frame`]

Register values in the interrupted thread, pushed by `intr_entry()`. The `esp_dummy` value isn't actually used (refer to the description of `PUSHA` in [IA32-v2b] for details).

`uint32_t vec_no` [Member of `struct intr_frame`]

The interrupt vector number, ranging from 0 to 255.

`uint32_t error_code` [Member of `struct intr_frame`]

The "error code" pushed on the stack by the CPU for some internal interrupts.

`void (*eip) (`*void*`)` [Member of `struct intr_frame`]

The address of the next instruction to be executed by the interrupted thread.

`void * esp` [Member of `struct intr_frame`]

The interrupted thread's stack pointer.

`const char * intr_name (`*uint8_t* *`vec`*`)` [Function]

Returns the name of the interrupt numbered *vec*, or `"unknown"` if the interrupt has no registered name.

A.4.2 Internal Interrupt Handling

Internal interrupts are caused directly by CPU instructions executed by the running kernel thread or user process (from project 2 onward). An internal interrupt is therefore said to arise in a "process context."

In an internal interrupt's handler, it can make sense to examine the `struct intr_frame` passed to the interrupt handler, or even to modify it. When the interrupt returns, modifications in `struct intr_frame` become changes to the calling thread or process's state. For example, the Pintos system call handler returns a value to the user program by modifying the saved EAX register (see ⟨undefined⟩ [System Call Details], page ⟨undefined⟩).

There are no special restrictions on what an internal interrupt handler can or can't do. Generally they should run with interrupts enabled, just like other code, and so they can be preempted by other kernel threads. Thus, they do need to synchronize with other threads on shared data and other resources (see Section A.3 [Synchronization], page 29).

Internal interrupt handlers can be invoked recursively. For example, the system call handler might cause a page fault while attempting to read user memory. Deep recursion would risk overflowing the limited kernel stack (see Section A.2.1 [struct thread], page 24), but should be unnecessary.

`void intr_register_int` (*uint8_t* **`vec`**, *int* **`dpl`**, *enum intr_level* **`level`**, *intr_handler_func* ***`handler`**, *const char* ***`name`**) [Function]

Registers *handler* to be called when internal interrupt numbered *vec* is triggered. Names the interrupt *name* for debugging purposes.

If *level* is `INTR_ON`, external interrupts will be processed normally during the interrupt handler's execution, which is normally desirable. Specifying `INTR_OFF` will cause the CPU to disable external interrupts when it invokes the interrupt handler. The effect is slightly different from calling `intr_disable()` inside the handler, because that leaves a window of one or more CPU instructions in which external interrupts are still enabled. This is important for the page fault handler; refer to the comments in '`userprog/exception.c`' for details.

dpl determines how the interrupt can be invoked. If *dpl* is 0, then the interrupt can be invoked only by kernel threads. Otherwise *dpl* should be 3, which allows user processes to invoke the interrupt with an explicit INT instruction. The value of *dpl* doesn't affect user processes' ability to invoke the interrupt indirectly, e.g. an invalid memory reference will cause a page fault regardless of *dpl*.

A.4.3 External Interrupt Handling

External interrupts are caused by events outside the CPU. They are asynchronous, so they can be invoked at any time that interrupts have not been disabled. We say that an external interrupt runs in an "interrupt context."

In an external interrupt, the `struct intr_frame` passed to the handler is not very meaningful. It describes the state of the thread or process that was interrupted, but there is no way to predict which one that is. It is possible, although rarely useful, to examine it, but modifying it is a recipe for disaster.

Only one external interrupt may be processed at a time. Neither internal nor external interrupt may nest within an external interrupt handler. Thus, an external interrupt's handler must run with interrupts disabled (see Section A.3.1 [Disabling Interrupts], page 29).

An external interrupt handler must not sleep or yield, which rules out calling `lock_acquire()`, `thread_yield()`, and many other functions. Sleeping in interrupt context would effectively put the interrupted thread to sleep, too, until the interrupt handler was again scheduled and returned. This would be unfair to the unlucky thread, and it would deadlock if the handler were waiting for the sleeping thread to, e.g., release a lock.

An external interrupt handler effectively monopolizes the machine and delays all other activities. Therefore, external interrupt handlers should complete as quickly as they can. Anything that require much CPU time should instead run in a kernel thread, possibly one that the interrupt triggers using a synchronization primitive.

External interrupts are controlled by a pair of devices outside the CPU called *programmable interrupt controllers*, *PICs* for short. When `intr_init()` sets up the CPU's IDT, it also initializes the PICs for interrupt handling. The PICs also must be "acknowledged" at the end of processing for each external interrupt. `intr_handler()` takes care of that by calling `pic_end_of_interrupt()`, which properly signals the PICs.

The following functions relate to external interrupts.

`void intr_register_ext` (*uint8_t* `vec`, *intr_handler_func* `*handler`, *const char* `*name`) [Function]

Registers *handler* to be called when external interrupt numbered *vec* is triggered. Names the interrupt *name* for debugging purposes. The handler will run with interrupts disabled.

`bool intr_context` (*void*) [Function]

Returns true if we are running in an interrupt context, otherwise false. Mainly used in functions that might sleep or that otherwise should not be called from interrupt context, in this form:

```
ASSERT (!intr_context ());
```

`void intr_yield_on_return` (*void*) [Function]

When called in an interrupt context, causes `thread_yield()` to be called just before the interrupt returns. Used in the timer interrupt handler when a thread's time slice expires, to cause a new thread to be scheduled.

A.5 Memory Allocation

Pintos contains two memory allocators, one that allocates memory in units of a page, and one that can allocate blocks of any size.

A.5.1 Page Allocator

The page allocator declared in '`threads/palloc.h`' allocates memory in units of a page. It is most often used to allocate memory one page at a time, but it can also allocate multiple contiguous pages at once.

The page allocator divides the memory it allocates into two pools, called the kernel and user pools. By default, each pool gets half of system memory above 1 MB, but the division can be changed with the '`-ul`' kernel command line option (see ⟨undefined⟩ [Why PAL_USER?], page ⟨undefined⟩). An allocation request draws from one pool or the other. If one pool becomes empty, the other may still have free pages. The user pool should be used for allocating memory for user processes and the kernel pool for all other allocations. This will only become important starting with project 3. Until then, all allocations should be made from the kernel pool.

Each pool's usage is tracked with a bitmap, one bit per page in the pool. A request to allocate *n* pages scans the bitmap for *n* consecutive bits set to false, indicating that those pages are free, and then sets those bits to true to mark them as used. This is a "first fit" allocation strategy (see [Wilson], page 69).

The page allocator is subject to fragmentation. That is, it may not be possible to allocate *n* contiguous pages even though *n* or more pages are free, because the free pages are separated by used pages. In fact, in pathological cases it may be impossible to allocate 2 contiguous pages even though half of the pool's pages are free. Single-page requests can't fail due to fragmentation, so requests for multiple contiguous pages should be limited as much as possible.

Pages may not be allocated from interrupt context, but they may be freed.

When a page is freed, all of its bytes are cleared to `0xcc`, as a debugging aid (see Section B.8 [Debugging Tips], page 64).

Page allocator types and functions are described below.

`void * palloc_get_page` (*enum palloc_flags* ***`flags`***) [Function]
`void * palloc_get_multiple` (*enum palloc_flags* ***`flags`***, *size_t* ***`page_cnt`***) [Function]

Obtains and returns one page, or *page_cnt* contiguous pages, respectively. Returns a null pointer if the pages cannot be allocated.

The *flags* argument may be any combination of the following flags:

`PAL_ASSERT` [Page Allocator Flag]

If the pages cannot be allocated, panic the kernel. This is only appropriate during kernel initialization. User processes should never be permitted to panic the kernel.

`PAL_ZERO` [Page Allocator Flag]

Zero all the bytes in the allocated pages before returning them. If not set, the contents of newly allocated pages are unpredictable.

`PAL_USER` [Page Allocator Flag]

Obtain the pages from the user pool. If not set, pages are allocated from the kernel pool.

`void palloc_free_page` (*void* ****`page`***) [Function]
`void palloc_free_multiple` (*void* ****`pages`***, *size_t* ***`page_cnt`***) [Function]

Frees one page, or *page_cnt* contiguous pages, respectively, starting at *pages*. All of the pages must have been obtained using `palloc_get_page()` or `palloc_get_multiple()`.

A.5.2 Block Allocator

The block allocator, declared in '`threads/malloc.h`', can allocate blocks of any size. It is layered on top of the page allocator described in the previous section. Blocks returned by the block allocator are obtained from the kernel pool.

The block allocator uses two different strategies for allocating memory. The first strategy applies to blocks that are 1 kB or smaller (one-fourth of the page size). These allocations are rounded up to the nearest power of 2, or 16 bytes, whichever is larger. Then they are grouped into a page used only for allocations of that size.

The second strategy applies to blocks larger than 1 kB. These allocations (plus a small amount of overhead) are rounded up to the nearest page in size, and then the block allocator requests that number of contiguous pages from the page allocator.

In either case, the difference between the allocation requested size and the actual block size is wasted. A real operating system would carefully tune its allocator to minimize this waste, but this is unimportant in an instructional system like Pintos.

As long as a page can be obtained from the page allocator, small allocations always succeed. Most small allocations do not require a new page from the page allocator at all, because they are satisfied using part of a page already allocated. However, large allocations always require calling into the page allocator, and any allocation that needs more than one contiguous page can fail due to fragmentation, as already discussed in the previous section. Thus, you should minimize the number of large allocations in your code, especially those over approximately 4 kB each.

When a block is freed, all of its bytes are cleared to `0xcc`, as a debugging aid (see Section B.8 [Debugging Tips], page 64).

The block allocator may not be called from interrupt context.

The block allocator functions are described below. Their interfaces are the same as the standard C library functions of the same names.

`void * malloc` (*size_t* `size`) [Function]

> Obtains and returns a new block, from the kernel pool, at least *size* bytes long. Returns a null pointer if *size* is zero or if memory is not available.

`void * calloc` (*size_t* `a`, *size_t* `b`) [Function]

> Obtains a returns a new block, from the kernel pool, at least `a * b` bytes long. The block's contents will be cleared to zeros. Returns a null pointer if *a* or *b* is zero or if insufficient memory is available.

`void * realloc` (*void **`block`, *size_t* `new_size`) [Function]

> Attempts to resize *block* to *new_size* bytes, possibly moving it in the process. If successful, returns the new block, in which case the old block must no longer be accessed. On failure, returns a null pointer, and the old block remains valid.
>
> A call with *block* null is equivalent to `malloc()`. A call with *new_size* zero is equivalent to `free()`.

`void free` (*void **`block`) [Function]

> Frees *block*, which must have been previously returned by `malloc()`, `calloc()`, or `realloc()` (and not yet freed).

A.6 Virtual Addresses

A 32-bit virtual address can be divided into a 20-bit *page number* and a 12-bit *page offset* (or just *offset*), like this:

```
 31               12 11        0
+-------------------+-----------+
|    Page Number    |   Offset  |
+-------------------+-----------+
         Virtual Address
```

Header '`threads/vaddr.h`' defines these functions and macros for working with virtual addresses:

`PGSHIFT` [Macro]
`PGBITS` [Macro]

The bit index (0) and number of bits (12) of the offset part of a virtual address, respectively.

`PGMASK` [Macro]

A bit mask with the bits in the page offset set to 1, the rest set to 0 (`0xfff`).

`PGSIZE` [Macro]

The page size in bytes (4,096).

`unsigned pg_ofs` (*const void **`va`) [Function]

Extracts and returns the page offset in virtual address *va*.

`uintptr_t pg_no` (*const void **`va`) [Function]

Extracts and returns the page number in virtual address *va*.

`void * pg_round_down` (*const void **`va`) [Function]

Returns the start of the virtual page that *va* points within, that is, *va* with the page offset set to 0.

`void * pg_round_up` (*const void **`va`) [Function]

Returns *va* rounded up to the nearest page boundary.

Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory (see ⟨undefined⟩ [Virtual Memory Layout], page ⟨undefined⟩). The boundary between them is `PHYS_BASE`:

`PHYS_BASE` [Macro]

Base address of kernel virtual memory. It defaults to `0xc0000000` (3 GB), but it may be changed to any multiple of `0x10000000` from `0x80000000` to `0xf0000000`.

User virtual memory ranges from virtual address 0 up to `PHYS_BASE`. Kernel virtual memory occupies the rest of the virtual address space, from `PHYS_BASE` up to 4 GB.

`bool is_user_vaddr` (*const void **`va`) [Function]
`bool is_kernel_vaddr` (*const void **`va`) [Function]

Returns true if *va* is a user or kernel virtual address, respectively, false otherwise.

The 80x86 doesn't provide any way to directly access memory given a physical address. This ability is often necessary in an operating system kernel, so Pintos works around it by mapping kernel virtual memory one-to-one to physical memory. That is, virtual address `PHYS_BASE` accesses physical address 0, virtual address `PHYS_BASE + 0x1234` accesses physical address `0x1234`, and so on up to the size of the machine's physical memory. Thus, adding `PHYS_BASE` to a physical address obtains a kernel virtual address that accesses that address; conversely, subtracting `PHYS_BASE` from a kernel virtual address obtains the corresponding physical address. Header '`threads/vaddr.h`' provides a pair of functions to do these translations:

`void * ptov` (*uintptr_t* `pa`) [Function]

Returns the kernel virtual address corresponding to physical address *pa*, which should be between 0 and the number of bytes of physical memory.

`uintptr_t vtop` (*void* *`va`) [Function]

Returns the physical address corresponding to *va*, which must be a kernel virtual address.

A.7 Page Table

The code in '`pagedir.c`' is an abstract interface to the 80x86 hardware page table, also called a "page directory" by Intel processor documentation. The page table interface uses a `uint32_t *` to represent a page table because this is convenient for accessing their internal structure.

The sections below describe the page table interface and internals.

A.7.1 Creation, Destruction, and Activation

These functions create, destroy, and activate page tables. The base Pintos code already calls these functions where necessary, so it should not be necessary to call them yourself.

`uint32_t * pagedir_create` (*void*) [Function]

Creates and returns a new page table. The new page table contains Pintos's normal kernel virtual page mappings, but no user virtual mappings.

Returns a null pointer if memory cannot be obtained.

`void pagedir_destroy` (*uint32_t* *`pd`) [Function]

Frees all of the resources held by *pd*, including the page table itself and the frames that it maps.

`void pagedir_activate` (*uint32_t* *`pd`) [Function]

Activates *pd*. The active page table is the one used by the CPU to translate memory references.

A.7.2 Inspection and Updates

These functions examine or update the mappings from pages to frames encapsulated by a page table. They work on both active and inactive page tables (that is, those for running and suspended processes), flushing the TLB as necessary.

`bool` **`pagedir_set_page`** (*uint32_t *`pd`, void *`upage`, void *`kpage`, bool `writable`*) [Function]

Adds to *pd* a mapping from user page *upage* to the frame identified by kernel virtual address *kpage*. If *writable* is true, the page is mapped read/write; otherwise, it is mapped read-only.

User page *upage* must not already be mapped in *pd*.

Kernel page *kpage* should be a kernel virtual address obtained from the user pool with `palloc_get_page(PAL_USER)` (see ⟨undefined⟩ [Why PAL_USER?], page ⟨undefined⟩).

Returns true if successful, false on failure. Failure will occur if additional memory required for the page table cannot be obtained.

`void *` **`pagedir_get_page`** (*uint32_t *`pd`, const void *`uaddr`*) [Function]

Looks up the frame mapped to *uaddr* in *pd*. Returns the kernel virtual address for that frame, if *uaddr* is mapped, or a null pointer if it is not.

`void` **`pagedir_clear_page`** (*uint32_t *`pd`, void *`page`*) [Function]

Marks *page* "not present" in *pd*. Later accesses to the page will fault.

Other bits in the page table for *page* are preserved, permitting the accessed and dirty bits (see the next section) to be checked.

This function has no effect if *page* is not mapped.

A.7.3 Accessed and Dirty Bits

80x86 hardware provides some assistance for implementing page replacement algorithms, through a pair of bits in the page table entry (PTE) for each page. On any read or write to a page, the CPU sets the *accessed bit* to 1 in the page's PTE, and on any write, the CPU sets the *dirty bit* to 1. The CPU never resets these bits to 0, but the OS may do so.

Proper interpretation of these bits requires understanding of *aliases*, that is, two (or more) pages that refer to the same frame. When an aliased frame is accessed, the accessed and dirty bits are updated in only one page table entry (the one for the page used for access). The accessed and dirty bits for the other aliases are not updated.

See ⟨undefined⟩ [Accessed and Dirty Bits], page ⟨undefined⟩, on applying these bits in implementing page replacement algorithms.

`bool` **`pagedir_is_dirty`** (*uint32_t *`pd`, const void *`page`*) [Function]
`bool` **`pagedir_is_accessed`** (*uint32_t *`pd`, const void *`page`*) [Function]

Returns true if page directory *pd* contains a page table entry for *page* that is marked dirty (or accessed). Otherwise, returns false.

`void` **`pagedir_set_dirty`** (*uint32_t *`pd`, const void *`page`, bool `value`*) [Function]
`void` **`pagedir_set_accessed`** (*uint32_t *`pd`, const void *`page`, bool `value`*) [Function]

If page directory *pd* has a page table entry for *page*, then its dirty (or accessed) bit is set to *value*.

A.7.4 Page Table Details

The functions provided with Pintos are sufficient to implement the projects. However, you may still find it worthwhile to understand the hardware page table format, so we'll go into a little detail in this section.

A.7.4.1 Structure

The top-level paging data structure is a page called the "page directory" (PD) arranged as an array of 1,024 32-bit page directory entries (PDEs), each of which represents 4 MB of virtual memory. Each PDE may point to the physical address of another page called a "page table" (PT) arranged, similarly, as an array of 1,024 32-bit page table entries (PTEs), each of which translates a single 4 kB virtual page to a physical page.

Translation of a virtual address into a physical address follows the three-step process illustrated in the diagram below:[2]

1. The most-significant 10 bits of the virtual address (bits 22...31) index the page directory. If the PDE is marked "present," the physical address of a page table is read from the PDE thus obtained. If the PDE is marked "not present" then a page fault occurs.
2. The next 10 bits of the virtual address (bits 12...21) index the page table. If the PTE is marked "present," the physical address of a data page is read from the PTE thus obtained. If the PTE is marked "not present" then a page fault occurs.
3. The least-significant 12 bits of the virtual address (bits 0...11) are added to the data page's physical base address, yielding the final physical address.

[2] Actually, virtual to physical translation on the 80x86 architecture occurs via an intermediate "linear address," but Pintos (and most modern 80x86 OSes) set up the CPU so that linear and virtual addresses are one and the same. Thus, you can effectively ignore this CPU feature.

```
 31                  22 21                  12 11                   0
+----------------------+----------------------+----------------------+
| Page Directory Index |   Page Table Index   |    Page Offset       |
+----------------------+----------------------+----------------------+
             |                    |                     |
     _______/             _______/                _____/
    /                    /                       /
   /    Page Directory  /      Page Table       /    Data Page
  /     .____________. /     .____________.    /   .____________.
  |1,023|____________| |1,023|____________|    |   |____________|
  |1,022|____________| |1,022|____________|    |   |____________|
  |1,021|____________| |1,021|____________|    \__\|____________|
  |1,020|____________| |1,020|____________|       /|____________|
  |     |            | |     |            |        |            |
  |     |            | \____\|            |_       |            |
  |     |      .     |      /|      .     | \      |      .     |
  \____\|      .     |_      |      .     |  |     |      .     |
       /|      .     | \     |      .     |  |     |      .     |
        |      .     |  |    |      .     |  |     |      .     |
        |            |  |    |            |  |     |            |
        |____________|  |    |____________|  |     |____________|
       4|____________|  |   4|____________|  |     |____________|
       3|____________|  |   3|____________|  |     |____________|
       2|____________|  |   2|____________|  |     |____________|
       1|____________|  |   1|____________|  |     |____________|
       0|____________|  \__\0|____________|  \____\|____________|
                           /                      /
```

Pintos provides some macros and functions that are useful for working with raw page tables:

`PTSHIFT` [Macro]
`PTBITS` [Macro]

The starting bit index (12) and number of bits (10), respectively, in a page table index.

`PTMASK` [Macro]

A bit mask with the bits in the page table index set to 1 and the rest set to 0 (`0x3ff000`).

`PTSPAN` [Macro]

The number of bytes of virtual address space that a single page table page covers (4,194,304 bytes, or 4 MB).

`PDSHIFT` [Macro]
`PDBITS` [Macro]

The starting bit index (22) and number of bits (10), respectively, in a page directory index.

`PDMASK` [Macro]

> A bit mask with the bits in the page directory index set to 1 and other bits set to 0 (`0xffc00000`).

`uintptr_t pd_no` (*const void* `*va`) [Function]
`uintptr_t pt_no` (*const void* `*va`) [Function]

> Returns the page directory index or page table index, respectively, for virtual address *va*. These functions are defined in '`threads/pte.h`'.

`unsigned pg_ofs` (*const void* `*va`) [Function]

> Returns the page offset for virtual address *va*. This function is defined in '`threads/vaddr.h`'.

A.7.4.2 Page Table Entry Format

You do not need to understand the PTE format to do the Pintos projects, unless you wish to incorporate the page table into your supplemental page table (see ⟨undefined⟩ [Managing the Supplemental Page Table], page ⟨undefined⟩).

The actual format of a page table entry is summarized below. For complete information, refer to section 3.7, "Page Translation Using 32-Bit Physical Addressing," in [IA32-v3a].

```
 31                                   12 11 9      6 5     2 1 0
+---------------------------------------+----+----+-+-+---+-+-+-+
|           Physical Address            | AVL|    |D|A|   |U|W|P|
+---------------------------------------+----+----+-+-+---+-+-+-+
```

Some more information on each bit is given below. The names are '`threads/pte.h`' macros that represent the bits' values:

`PTE_P` [Macro]

> Bit 0, the "present" bit. When this bit is 1, the other bits are interpreted as described below. When this bit is 0, any attempt to access the page will page fault. The remaining bits are then not used by the CPU and may be used by the OS for any purpose.

`PTE_W` [Macro]

> Bit 1, the "read/write" bit. When it is 1, the page is writable. When it is 0, write attempts will page fault.

`PTE_U` [Macro]

> Bit 2, the "user/supervisor" bit. When it is 1, user processes may access the page. When it is 0, only the kernel may access the page (user accesses will page fault).
>
> Pintos clears this bit in PTEs for kernel virtual memory, to prevent user processes from accessing them.

`PTE_A` [Macro]

> Bit 5, the "accessed" bit. See Section A.7.3 [Page Table Accessed and Dirty Bits], page 43.

`PTE_D` [Macro]

> Bit 6, the "dirty" bit. See Section A.7.3 [Page Table Accessed and Dirty Bits], page 43.

`PTE_AVL` [Macro]

Bits 9...11, available for operating system use. Pintos, as provided, does not use them and sets them to 0.

`PTE_ADDR` [Macro]

Bits 12...31, the top 20 bits of the physical address of a frame. The low 12 bits of the frame's address are always 0.

Other bits are either reserved or uninteresting in a Pintos context and should be set to 0.

Header '`threads/pte.h`' defines three functions for working with page table entries:

`uint32_t pte_create_kernel` (*uint32_t* `*page`, *bool* `writable`) [Function]

Returns a page table entry that points to *page*, which should be a kernel virtual address. The PTE's present bit will be set. It will be marked for kernel-only access. If *writable* is true, the PTE will also be marked read/write; otherwise, it will be read-only.

`uint32_t pte_create_user` (*uint32_t* `*page`, *bool* `writable`) [Function]

Returns a page table entry that points to *page*, which should be the kernel virtual address of a frame in the user pool (see ⟨undefined⟩ [Why PAL_USER?], page ⟨undefined⟩). The PTE's present bit will be set and it will be marked to allow user-mode access. If *writable* is true, the PTE will also be marked read/write; otherwise, it will be read-only.

`void * pte_get_page` (*uint32_t* `pte`) [Function]

Returns the kernel virtual address for the frame that *pte* points to. The *pte* may be present or not-present; if it is not-present then the pointer returned is only meaningful if the address bits in the PTE actually represent a physical address.

A.7.4.3 Page Directory Entry Format

Page directory entries have the same format as PTEs, except that the physical address points to a page table page instead of a frame. Header '`threads/pte.h`' defines two functions for working with page directory entries:

`uint32_t pde_create` (*uint32_t* `*pt`) [Function]

Returns a page directory that points to *page*, which should be the kernel virtual address of a page table page. The PDE's present bit will be set, it will be marked to allow user-mode access, and it will be marked read/write.

`uint32_t * pde_get_pt` (*uint32_t* `pde`) [Function]

Returns the kernel virtual address for the page table page that *pde*, which must be marked present, points to.

A.8 Hash Table

Pintos provides a hash table data structure in '`lib/kernel/hash.c`'. To use it you will need to include its header file, '`lib/kernel/hash.h`', with `#include <hash.h>`. No code provided with Pintos uses the hash table, which means that you are free to use it as is, modify its implementation for your own purposes, or ignore it, as you wish.

Most implementations of the virtual memory project use a hash table to translate pages to frames. You may find other uses for hash tables as well.

A.8.1 Data Types

A hash table is represented by `struct hash`.

`struct hash` [Type]

Represents an entire hash table. The actual members of `struct hash` are "opaque." That is, code that uses a hash table should not access `struct hash` members directly, nor should it need to. Instead, use hash table functions and macros.

The hash table operates on elements of type `struct hash_elem`.

`struct hash_elem` [Type]

Embed a `struct hash_elem` member in the structure you want to include in a hash table. Like `struct hash`, `struct hash_elem` is opaque. All functions for operating on hash table elements actually take and return pointers to `struct hash_elem`, not pointers to your hash table's real element type.

You will often need to obtain a `struct hash_elem` given a real element of the hash table, and vice versa. Given a real element of the hash table, you may use the '`&`' operator to obtain a pointer to its `struct hash_elem`. Use the `hash_entry()` macro to go the other direction.

type `* hash_entry` (*struct hash_elem* **elem*, *type*, *member*) [Macro]

Returns a pointer to the structure that *elem*, a pointer to a `struct hash_elem`, is embedded within. You must provide *type*, the name of the structure that *elem* is inside, and *member*, the name of the member in *type* that *elem* points to.

For example, suppose `h` is a `struct hash_elem *` variable that points to a `struct thread` member (of type `struct hash_elem`) named `h_elem`. Then, `hash_entry (h, struct thread, h_elem)` yields the address of the `struct thread` that `h` points within.

See Section A.8.5 [Hash Table Example], page 52, for an example.

Each hash table element must contain a key, that is, data that identifies and distinguishes elements, which must be unique among elements in the hash table. (Elements may also contain non-key data that need not be unique.) While an element is in a hash table, its key data must not be changed. Instead, if need be, remove the element from the hash table, modify its key, then reinsert the element.

For each hash table, you must write two functions that act on keys: a hash function and a comparison function. These functions must match the following prototypes:

`unsigned hash_hash_func (const struct hash_elem *element, void *aux)` [Type]

Returns a hash of *element*'s data, as a value anywhere in the range of `unsigned int`. The hash of an element should be a pseudo-random function of the element's key. It must not depend on non-key data in the element or on any non-constant data other than the key. Pintos provides the following functions as a suitable basis for hash functions.

`unsigned hash_bytes` (*const void* **buf*, *size_t* **size*) [Function]

Returns a hash of the *size* bytes starting at *buf*. The implementation is the general-purpose Fowler-Noll-Vo hash for 32-bit words.

`unsigned hash_string` (*const char* ***s**) [Function]

Returns a hash of null-terminated string *s*.

`unsigned hash_int` (*int* **i**) [Function]

Returns a hash of integer *i*.

If your key is a single piece of data of an appropriate type, it is sensible for your hash function to directly return the output of one of these functions. For multiple pieces of data, you may wish to combine the output of more than one call to them using, e.g., the '^' (exclusive or) operator. Finally, you may entirely ignore these functions and write your own hash function from scratch, but remember that your goal is to build an operating system kernel, not to design a hash function.

See Section A.8.6 [Hash Auxiliary Data], page 53, for an explanation of *aux*.

`bool hash_less_func (const struct hash_elem *a, const struct hash_elem *b, void *aux)` [Type]

Compares the keys stored in elements *a* and *b*. Returns true if *a* is less than *b*, false if *a* is greater than or equal to *b*.

If two elements compare equal, then they must hash to equal values.

See Section A.8.6 [Hash Auxiliary Data], page 53, for an explanation of *aux*.

See Section A.8.5 [Hash Table Example], page 52, for hash and comparison function examples.

A few functions accept a pointer to a third kind of function as an argument:

`void hash_action_func (struct hash_elem *element, void *aux)` [Type]

Performs some kind of action, chosen by the caller, on *element*.

See Section A.8.6 [Hash Auxiliary Data], page 53, for an explanation of *aux*.

A.8.2 Basic Functions

These functions create, destroy, and inspect hash tables.

`bool hash_init` (*struct hash* ***hash**, *hash_hash_func* ***hash_func**, *hash_less_func* ***less_func**, *void* ***aux**) [Function]

Initializes *hash* as a hash table with *hash_func* as hash function, *less_func* as comparison function, and *aux* as auxiliary data. Returns true if successful, false on failure. `hash_init()` calls `malloc()` and fails if memory cannot be allocated.

See Section A.8.6 [Hash Auxiliary Data], page 53, for an explanation of *aux*, which is most often a null pointer.

`void hash_clear` (*struct hash* ***hash**, *hash_action_func* ***action**) [Function]

Removes all the elements from *hash*, which must have been previously initialized with `hash_init()`.

If *action* is non-null, then it is called once for each element in the hash table, which gives the caller an opportunity to deallocate any memory or other resources used by the element. For example, if the hash table elements are dynamically allocated using `malloc()`, then *action* could `free()` the element. This is safe because `hash_clear()` will not access the memory in a given hash element after calling *action* on it. However, *action* must not call any function that may modify the hash table, such as `hash_insert()` or `hash_delete()`.

`void hash_destroy` (*struct hash* ***hash***, *hash_action_func* ***action***) [Function]

If *action* is non-null, calls it for each element in the hash, with the same semantics as a call to `hash_clear()`. Then, frees the memory held by *hash*. Afterward, *hash* must not be passed to any hash table function, absent an intervening call to `hash_init()`.

`size_t hash_size` (*struct hash* ***hash***) [Function]

Returns the number of elements currently stored in *hash*.

`bool hash_empty` (*struct hash* ***hash***) [Function]

Returns true if *hash* currently contains no elements, false if *hash* contains at least one element.

A.8.3 Search Functions

Each of these functions searches a hash table for an element that compares equal to one provided. Based on the success of the search, they perform some action, such as inserting a new element into the hash table, or simply return the result of the search.

`struct hash_elem * hash_insert` (*struct hash* ***hash***, *struct hash_elem* ***element***) [Function]

Searches *hash* for an element equal to *element*. If none is found, inserts *element* into *hash* and returns a null pointer. If the table already contains an element equal to *element*, it is returned without modifying *hash*.

`struct hash_elem * hash_replace` (*struct hash* ***hash***, *struct hash_elem* ***element***) [Function]

Inserts *element* into *hash*. Any element equal to *element* already in *hash* is removed. Returns the element removed, or a null pointer if *hash* did not contain an element equal to *element*.

The caller is responsible for deallocating any resources associated with the returned element, as appropriate. For example, if the hash table elements are dynamically allocated using `malloc()`, then the caller must `free()` the element after it is no longer needed.

The element passed to the following functions is only used for hashing and comparison purposes. It is never actually inserted into the hash table. Thus, only key data in the element needs to be initialized, and other data in the element will not be used. It often makes sense to declare an instance of the element type as a local variable, initialize the key data, and then pass the address of its `struct hash_elem` to `hash_find()` or `hash_delete()`. See Section A.8.5 [Hash Table Example], page 52, for an example. (Large structures should not be allocated as local variables. See Section A.2.1 [struct thread], page 24, for more information.)

`struct hash_elem * hash_find` (*struct hash* ***hash***, *struct hash_elem* ***element***) [Function]

Searches *hash* for an element equal to *element*. Returns the element found, if any, or a null pointer otherwise.

`struct hash_elem * hash_delete` (*struct hash* ***hash***, *struct hash_elem* ***element***) [Function]

Searches *hash* for an element equal to *element*. If one is found, it is removed from *hash* and returned. Otherwise, a null pointer is returned and *hash* is unchanged.

The caller is responsible for deallocating any resources associated with the returned element, as appropriate. For example, if the hash table elements are dynamically allocated using `malloc()`, then the caller must `free()` the element after it is no longer needed.

A.8.4 Iteration Functions

These functions allow iterating through the elements in a hash table. Two interfaces are supplied. The first requires writing and supplying a *hash_action_func* to act on each element (see Section A.8.1 [Hash Data Types], page 48).

`void hash_apply` (*struct hash* ***hash***, *hash_action_func* ***action***) [Function]

Calls *action* once for each element in *hash*, in arbitrary order. *action* must not call any function that may modify the hash table, such as `hash_insert()` or `hash_delete()`. *action* must not modify key data in elements, although it may modify any other data.

The second interface is based on an "iterator" data type. Idiomatically, iterators are used as follows:

```
struct hash_iterator i;

hash_first (&i, h);
while (hash_next (&i))
  {
    struct foo *f = hash_entry (hash_cur (&i), struct foo, elem);
    ...do something with f...
  }
```

`struct hash_iterator` [Type]

Represents a position within a hash table. Calling any function that may modify a hash table, such as `hash_insert()` or `hash_delete()`, invalidates all iterators within that hash table.

Like `struct hash` and `struct hash_elem`, `struct hash_elem` is opaque.

`void hash_first` (*struct hash_iterator* ***iterator***, *struct hash* ***hash***) [Function]

Initializes *iterator* to just before the first element in *hash*.

`struct hash_elem * hash_next` (*struct hash_iterator* ***iterator***) [Function]

Advances *iterator* to the next element in *hash*, and returns that element. Returns a null pointer if no elements remain. After `hash_next()` returns null for *iterator*, calling it again yields undefined behavior.

`struct hash_elem * hash_cur` (*struct hash_iterator* ***iterator***) [Function]

Returns the value most recently returned by `hash_next()` for *iterator*. Yields undefined behavior after `hash_first()` has been called on *iterator* but before `hash_next()` has been called for the first time.

A.8.5 Hash Table Example

Suppose you have a structure, called `struct page`, that you want to put into a hash table. First, define `struct page` to include a `struct hash_elem` member:

```
struct page
  {
    struct hash_elem hash_elem; /* Hash table element. */
    void *addr;                 /* Virtual address. */
    /* ...other members... */
  };
```

We write a hash function and a comparison function using *addr* as the key. A pointer can be hashed based on its bytes, and the '`<`' operator works fine for comparing pointers:

```
/* Returns a hash value for page p. */
unsigned
page_hash (const struct hash_elem *p_, void *aux UNUSED)
{
  const struct page *p = hash_entry (p_, struct page, hash_elem);
  return hash_bytes (&p->addr, sizeof p->addr);
}

/* Returns true if page a precedes page b. */
bool
page_less (const struct hash_elem *a_, const struct hash_elem *b_,
           void *aux UNUSED)
{
  const struct page *a = hash_entry (a_, struct page, hash_elem);
  const struct page *b = hash_entry (b_, struct page, hash_elem);

  return a->addr < b->addr;
}
```

(The use of `UNUSED` in these functions' prototypes suppresses a warning that *aux* is unused. See Section B.3 [Function and Parameter Attributes], page 54, for information about `UNUSED`. See Section A.8.6 [Hash Auxiliary Data], page 53, for an explanation of *aux*.)

Then, we can create a hash table like this:

```
struct hash pages;

hash_init (&pages, page_hash, page_less, NULL);
```

Now we can manipulate the hash table we've created. If `p` is a pointer to a `struct page`, we can insert it into the hash table with:

```
hash_insert (&pages, &p->hash_elem);
```

If there's a chance that *pages* might already contain a page with the same *addr*, then we should check `hash_insert()`'s return value.

To search for an element in the hash table, use `hash_find()`. This takes a little setup, because `hash_find()` takes an element to compare against. Here's a function that will find and return a page based on a virtual address, assuming that *pages* is defined at file scope:

```
/* Returns the page containing the given virtual address,
   or a null pointer if no such page exists. */
struct page *
page_lookup (const void *address)
{
  struct page p;
  struct hash_elem *e;

  p.addr = address;
  e = hash_find (&pages, &p.hash_elem);
  return e != NULL ? hash_entry (e, struct page, hash_elem) : NULL;
}
```

`struct page` is allocated as a local variable here on the assumption that it is fairly small. Large structures should not be allocated as local variables. See Section A.2.1 [struct thread], page 24, for more information.

A similar function could delete a page by address using `hash_delete()`.

A.8.6 Auxiliary Data

In simple cases like the example above, there's no need for the *aux* parameters. In these cases, just pass a null pointer to `hash_init()` for *aux* and ignore the values passed to the hash function and comparison functions. (You'll get a compiler warning if you don't use the *aux* parameter, but you can turn that off with the `UNUSED` macro, as shown in the example, or you can just ignore it.)

aux is useful when you have some property of the data in the hash table is both constant and needed for hashing or comparison, but not stored in the data items themselves. For example, if the items in a hash table are fixed-length strings, but the items themselves don't indicate what that fixed length is, you could pass the length as an *aux* parameter.

A.8.7 Synchronization

The hash table does not do any internal synchronization. It is the caller's responsibility to synchronize calls to hash table functions. In general, any number of functions that examine but do not modify the hash table, such as `hash_find()` or `hash_next()`, may execute simultaneously. However, these function cannot safely execute at the same time as any function that may modify a given hash table, such as `hash_insert()` or `hash_delete()`, nor may more than one function that can modify a given hash table execute safely at once.

It is also the caller's responsibility to synchronize access to data in hash table elements. How to synchronize access to this data depends on how it is designed and organized, as with any other data structure.

Appendix B Debugging Tools

Many tools lie at your disposal for debugging Pintos. This appendix introduces you to a few of them.

B.1 `printf()`

Don't underestimate the value of `printf()`. The way `printf()` is implemented in Pintos, you can call it from practically anywhere in the kernel, whether it's in a kernel thread or an interrupt handler, almost regardless of what locks are held.

`printf()` is useful for more than just examining data. It can also help figure out when and where something goes wrong, even when the kernel crashes or panics without a useful error message. The strategy is to sprinkle calls to `printf()` with different strings (e.g. `"<1>"`, `"<2>"`, ...) throughout the pieces of code you suspect are failing. If you don't even see `<1>` printed, then something bad happened before that point, if you see `<1>` but not `<2>`, then something bad happened between those two points, and so on. Based on what you learn, you can then insert more `printf()` calls in the new, smaller region of code you suspect. Eventually you can narrow the problem down to a single statement. See Section B.6 [Triple Faults], page 63, for a related technique.

B.2 `ASSERT`

Assertions are useful because they can catch problems early, before they'd otherwise be noticed. Ideally, each function should begin with a set of assertions that check its arguments for validity. (Initializers for functions' local variables are evaluated before assertions are checked, so be careful not to assume that an argument is valid in an initializer.) You can also sprinkle assertions throughout the body of functions in places where you suspect things are likely to go wrong. They are especially useful for checking loop invariants.

Pintos provides the `ASSERT` macro, defined in '`<debug.h>`', for checking assertions.

`ASSERT` (*expression*) [Macro]

Tests the value of *expression*. If it evaluates to zero (false), the kernel panics. The panic message includes the expression that failed, its file and line number, and a backtrace, which should help you to find the problem. See Section B.4 [Backtraces], page 55, for more information.

B.3 Function and Parameter Attributes

These macros defined in '`<debug.h>`' tell the compiler special attributes of a function or function parameter. Their expansions are GCC-specific.

`UNUSED` [Macro]

Appended to a function parameter to tell the compiler that the parameter might not be used within the function. It suppresses the warning that would otherwise appear.

`NO_RETURN` [Macro]

Appended to a function prototype to tell the compiler that the function never returns. It allows the compiler to fine-tune its warnings and its code generation.

`NO_INLINE` [Macro]

Appended to a function prototype to tell the compiler to never emit the function in-line. Occasionally useful to improve the quality of backtraces (see below).

`PRINTF_FORMAT (`*`format`*`, `*`first`*`)` [Macro]

Appended to a function prototype to tell the compiler that the function takes a `printf()`-like format string as the argument numbered *format* (starting from 1) and that the corresponding value arguments start at the argument numbered *first*. This lets the compiler tell you if you pass the wrong argument types.

B.4 Backtraces

When the kernel panics, it prints a "backtrace," that is, a summary of how your program got where it is, as a list of addresses inside the functions that were running at the time of the panic. You can also insert a call to `debug_backtrace()`, prototyped in '`<debug.h>`', to print a backtrace at any point in your code. `debug_backtrace_all()`, also declared in '`<debug.h>`', prints backtraces of all threads.

The addresses in a backtrace are listed as raw hexadecimal numbers, which are difficult to interpret. We provide a tool called `backtrace` to translate these into function names and source file line numbers. Give it the name of your '`kernel.o`' as the first argument and the hexadecimal numbers composing the backtrace (including the '`0x`' prefixes) as the remaining arguments. It outputs the function name and source file line numbers that correspond to each address.

If the translated form of a backtrace is garbled, or doesn't make sense (e.g. function A is listed above function B, but B doesn't call A), then it's a good sign that you're corrupting a kernel thread's stack, because the backtrace is extracted from the stack. Alternatively, it could be that the '`kernel.o`' you passed to `backtrace` is not the same kernel that produced the backtrace.

Sometimes backtraces can be confusing without any corruption. Compiler optimizations can cause surprising behavior. When a function has called another function as its final action (a *tail call*), the calling function may not appear in a backtrace at all. Similarly, when function A calls another function B that never returns, the compiler may optimize such that an unrelated function C appears in the backtrace instead of A. Function C is simply the function that happens to be in memory just after A. In the threads project, this is commonly seen in backtraces for test failures; see [`pass()` Fails], page 16, for more information.

B.4.1 Example

Here's an example. Suppose that Pintos printed out this following call stack, which is taken from an actual Pintos submission for the file system project:

```
Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319
0xc010325a 0x804812c 0x8048a96 0x8048ac8.
```

You would then invoke the `backtrace` utility like shown below, cutting and pasting the backtrace information into the command line. This assumes that '`kernel.o`' is in the current directory. You would of course enter all of the following on a single shell command line, even though that would overflow our margins here:

```
backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67
0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
```

The backtrace output would then look something like this:

```
0xc0106eff: debug_panic (lib/debug.c:86)
0xc01102fb: file_seek (filesys/file.c:405)
0xc010dc22: seek (userprog/syscall.c:744)
0xc010cf67: syscall_handler (userprog/syscall.c:444)
0xc0102319: intr_handler (threads/interrupt.c:334)
0xc010325a: intr_entry (threads/intr-stubs.S:38)
0x0804812c: (unknown)
0x08048a96: (unknown)
0x08048ac8: (unknown)
```

(You will probably not see exactly the same addresses if you run the command above on your own kernel binary, because the source code you compiled and the compiler you used are probably different.)

The first line in the backtrace refers to `debug_panic()`, the function that implements kernel panics. Because backtraces commonly result from kernel panics, `debug_panic()` will often be the first function shown in a backtrace.

The second line shows `file_seek()` as the function that panicked, in this case as the result of an assertion failure. In the source code tree used for this example, line 405 of '`filesys/file.c`' is the assertion

```
ASSERT (file_ofs >= 0);
```

(This line was also cited in the assertion failure message.) Thus, `file_seek()` panicked because it passed a negative file offset argument.

The third line indicates that `seek()` called `file_seek()`, presumably without validating the offset argument. In this submission, `seek()` implements the `seek` system call.

The fourth line shows that `syscall_handler()`, the system call handler, invoked `seek()`.

The fifth and sixth lines are the interrupt handler entry path.

The remaining lines are for addresses below `PHYS_BASE`. This means that they refer to addresses in the user program, not in the kernel. If you know what user program was running when the kernel panicked, you can re-run `backtrace` on the user program, like so: (typing the command on a single line, of course):

```
backtrace tests/filesys/extended/grow-too-big 0xc0106eff 0xc01102fb
0xc010dc22 0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96
0x8048ac8
```

The results look like this:

```
0xc0106eff: (unknown)
0xc01102fb: (unknown)
0xc010dc22: (unknown)
0xc010cf67: (unknown)
0xc0102319: (unknown)
0xc010325a: (unknown)
0x0804812c: test_main (...xtended/grow-too-big.c:20)
0x08048a96: main (tests/main.c:10)
```

```
0x08048ac8: _start (lib/user/entry.c:9)
```

You can even specify both the kernel and the user program names on the command line, like so:

```
backtrace kernel.o tests/filesys/extended/grow-too-big 0xc0106eff
0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 0xc010325a 0x804812c
0x8048a96 0x8048ac8
```

The result is a combined backtrace:

```
In kernel.o:
0xc0106eff: debug_panic (lib/debug.c:86)
0xc01102fb: file_seek (filesys/file.c:405)
0xc010dc22: seek (userprog/syscall.c:744)
0xc010cf67: syscall_handler (userprog/syscall.c:444)
0xc0102319: intr_handler (threads/interrupt.c:334)
0xc010325a: intr_entry (threads/intr-stubs.S:38)
In tests/filesys/extended/grow-too-big:
0x0804812c: test_main (...xtended/grow-too-big.c:20)
0x08048a96: main (tests/main.c:10)
0x08048ac8: _start (lib/user/entry.c:9)
```

Here's an extra tip for anyone who read this far: `backtrace` is smart enough to strip the `Call stack:` header and '`.`' trailer from the command line if you include them. This can save you a little bit of trouble in cutting and pasting. Thus, the following command prints the same output as the first one we used:

```
backtrace kernel.o Call stack: 0xc0106eff 0xc01102fb 0xc010dc22
0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8.
```

B.5 GDB

You can run Pintos under the supervision of the GDB debugger. First, start Pintos with the '`--gdb`' option, e.g. `pintos --gdb -- run mytest`. Second, open a second terminal on the same machine and use `pintos-gdb` to invoke GDB on '`kernel.o`':[1]

```
pintos-gdb kernel.o
```

and issue the following GDB command:

```
target remote localhost:1234
```

Now GDB is connected to the simulator over a local network connection. You can now issue any normal GDB commands. If you issue the '`c`' command, the simulated BIOS will take control, load Pintos, and then Pintos will run in the usual way. You can pause the process at any point with ⟨Ctrl+C⟩.

B.5.1 Using GDB

You can read the GDB manual by typing `info gdb` at a terminal command prompt. Here's a few commonly useful GDB commands:

[1] `pintos-gdb` is a wrapper around `gdb` (80x86) or `i386-elf-gdb` (SPARC) that loads the Pintos macros at startup.

`c` [GDB Command]

Continues execution until ⟨Ctrl+C⟩ or the next breakpoint.

`break` *function* [GDB Command]
`break` *file:line* [GDB Command]
`break` **address* [GDB Command]

Sets a breakpoint at *function*, at *line* within *file*, or *address*. (Use a '`0x`' prefix to specify an address in hex.)

Use `break main` to make GDB stop when Pintos starts running.

`p` *expression* [GDB Command]

Evaluates the given *expression* and prints its value. If the expression contains a function call, that function will actually be executed.

`l` **address* [GDB Command]

Lists a few lines of code around *address*. (Use a '`0x`' prefix to specify an address in hex.)

`bt` [GDB Command]

Prints a stack backtrace similar to that output by the `backtrace` program described above.

`p/a` *address* [GDB Command]

Prints the name of the function or variable that occupies *address*. (Use a '`0x`' prefix to specify an address in hex.)

`diassemble` *function* [GDB Command]

Disassembles *function*.

We also provide a set of macros specialized for debugging Pintos, written by Godmar Back `gback@cs.vt.edu`. You can type `help user-defined` for basic help with the macros. Here is an overview of their functionality, based on Godmar's documentation:

`debugpintos` [GDB Macro]

Attach debugger to a waiting pintos process on the same machine. Shorthand for `target remote localhost:1234`.

`dumplist` *list type element* [GDB Macro]

Prints the elements of *list*, which should be a `struct` list that contains elements of the given *type* (without the word `struct`) in which *element* is the `struct list_elem` member that links the elements.

Example: `dumplist all_list thread allelem` prints all elements of `struct thread` that are linked in `struct list all_list` using the `struct list_elem allelem` which is part of `struct thread`.

`btthread` *thread* [GDB Macro]

Shows the backtrace of *thread*, which is a pointer to the `struct thread` of the thread whose backtrace it should show. For the current thread, this is identical to the `bt` (backtrace) command. It also works for any thread suspended in `schedule()`, provided you know where its kernel stack page is located.

`btthreadlist` *list element* [GDB Macro]

Shows the backtraces of all threads in *list*, the `struct list` in which the threads are kept. Specify *element* as the `struct list_elem` field used inside `struct thread` to link the threads together.

Example: `btthreadlist all_list allelem` shows the backtraces of all threads contained in `struct list all_list`, linked together by `allelem`. This command is useful to determine where your threads are stuck when a deadlock occurs. Please see the example scenario below.

`btthreadall` [GDB Macro]

Short-hand for `btthreadlist all_list allelem`.

`btpagefault` [GDB Macro]

Print a backtrace of the current thread after a page fault exception. Normally, when a page fault exception occurs, GDB will stop with a message that might say:[2]

```
Program received signal 0, Signal 0.
0xc0102320 in intr0e_stub ()
```

In that case, the `bt` command might not give a useful backtrace. Use `btpagefault` instead.

You may also use `btpagefault` for page faults that occur in a user process. In this case, you may wish to also load the user program's symbol table using the `loadusersymbols` macro, as described above.

`hook-stop` [GDB Macro]

GDB invokes this macro every time the simulation stops, which Bochs will do for every processor exception, among other reasons. If the simulation stops due to a page fault, `hook-stop` will print a message that says and explains further whether the page fault occurred in the kernel or in user code.

If the exception occurred from user code, `hook-stop` will say:

```
pintos-debug: a page fault exception occurred in user mode
pintos-debug: hit 'c' to continue, or 's' to step to intr_handler
```

In Project 2, a page fault in a user process leads to the termination of the process. You should expect those page faults to occur in the robustness tests where we test that your kernel properly terminates processes that try to access invalid addresses. To debug those, set a break point in `page_fault()` in '`exception.c`', which you will need to modify accordingly.

In Project 3, a page fault in a user process no longer automatically leads to the termination of a process. Instead, it may require reading in data for the page the process was trying to access, either because it was swapped out or because this is the first time it's accessed. In either case, you will reach `page_fault()` and need to take the appropriate action there.

If the page fault did not occur in user mode while executing a user process, then it occurred in kernel mode while executing kernel code. In this case, `hook-stop` will print this message:

[2] To be precise, GDB will stop only when running under Bochs. When running under QEMU, you must set a breakpoint in the `page_fault` function to stop execution when a page fault occurs. In that case, the `btpagefault` macro is unnecessary.

```
    pintos-debug: a page fault occurred in kernel mode
```

followed by the output of the `btpagefault` command.

Before Project 3, a page fault exception in kernel code is always a bug in your kernel, because your kernel should never crash. Starting with Project 3, the situation will change if you use the `get_user()` and `put_user()` strategy to verify user memory accesses (see ⟨undefined⟩ [Accessing User Memory], page ⟨undefined⟩).

B.5.2 Example GDB Session

This section narrates a sample GDB session, provided by Godmar Back. This example illustrates how one might debug a Project 1 solution in which occasionally a thread that calls `timer_sleep()` is not woken up. With this bug, tests such as `mlfqs_load_1` get stuck.

This session was captured with a slightly older version of Bochs and the GDB macros for Pintos, so it looks slightly different than it would now. Program output is shown in normal type, user input in **strong** type.

First, I start Pintos:

```
$ pintos -v –gdb – -q -mlfqs run mlfqs-load-1
Writing command line to /tmp/gDAlqTB5Uf.dsk...
bochs -q
========================================================================
                       Bochs x86 Emulator 2.2.5
             Build from CVS snapshot on December 30, 2005
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000i[     ] Enabled gdbstub
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
Waiting for gdb connection on localhost:1234
```

Then, I open a second window on the same machine and start GDB:

```
$ pintos-gdb kernel.o
GNU gdb Red Hat Linux (6.3.0.0-1.84rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
```

Then, I tell GDB to attach to the waiting Pintos emulator:

```
(gdb) debugpintos
Remote debugging using localhost:1234
0x0000fff0 in ?? ()
Reply contains invalid hex digit 78
```

Now I tell Pintos to run by executing `c` (short for `continue`) twice:

```
(gdb) c
Continuing.
Reply contains invalid hex digit 78
(gdb) c
Continuing.
```

Now Pintos will continue and output:

```
Pintos booting with 4,096 kB RAM...
Kernel command line: -q -mlfqs run mlfqs-load-1
374 pages available in kernel pool.
373 pages available in user pool.
Calibrating timer...  102,400 loops/s.
Boot complete.
Executing 'mlfqs-load-1':
(mlfqs-load-1) begin
(mlfqs-load-1) spinning for up to 45 seconds, please wait...
(mlfqs-load-1) load average rose to 0.5 after 42 seconds
(mlfqs-load-1) sleeping for another 10 seconds, please wait...
```

...until it gets stuck because of the bug I had introduced. I hit ⟨Ctrl+C⟩ in the debugger window:

```
Program received signal 0, Signal 0.
0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
649   while (i <= PRI_MAX && list_empty (&ready_list[i]))
(gdb)
```

The thread that was running when I interrupted Pintos was the idle thread. If I run `backtrace`, it shows this backtrace:

```
(gdb) bt
#0  0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
#1  0xc0101778 in schedule () at ../../threads/thread.c:714
#2  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#3  0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551
#4  0xc010145a in kernel_thread (function=0xc01013ff , aux=0x0)
    at ../../threads/thread.c:575
#5  0x00000000 in ?? ()
```

Not terribly useful. What I really like to know is what's up with the other thread (or threads). Since I keep all threads in a linked list called `all_list`, linked together by a `struct list_elem` member named `allelem`, I can use the `btthreadlist` macro from the macro library I wrote. `btthreadlist` iterates through the list of threads and prints the backtrace for each thread:

```
(gdb) btthreadlist all_list allelem
pintos-debug: dumping backtrace of thread 'main' @0xc002f000
#0  0xc0101820 in schedule () at ../../threads/thread.c:722
#1  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#2  0xc0104755 in timer_sleep (ticks=1000) at ../../devices/timer.c:141
#3  0xc010bf7c in test_mlfqs_load_1 () at ../../tests/threads/mlfqs-load-1.c:49
#4  0xc010aabb in run_test (name=0xc0007d8c "mlfqs-load-1")
    at ../../tests/threads/tests.c:50
#5  0xc0100647 in run_task (argv=0xc0110d28) at ../../threads/init.c:281
#6  0xc0100721 in run_actions (argv=0xc0110d28) at ../../threads/init.c:331
#7  0xc01000c7 in main () at ../../threads/init.c:140

pintos-debug: dumping backtrace of thread 'idle' @0xc0116000
#0  0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
#1  0xc0101778 in schedule () at ../../threads/thread.c:714
#2  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#3  0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551
#4  0xc010145a in kernel_thread (function=0xc01013ff , aux=0x0)
    at ../../threads/thread.c:575
#5  0x00000000 in ?? ()
```

In this case, there are only two threads, the idle thread and the main thread. The kernel stack pages (to which the `struct thread` points) are at `0xc0116000` and `0xc002f000`, respectively. The main thread is stuck in `timer_sleep()`, called from `test_mlfqs_load_1`.

Knowing where threads are stuck can be tremendously useful, for instance when diagnosing deadlocks or unexplained hangs.

`loadusersymbols` [GDB Macro]

You can also use GDB to debug a user program running under Pintos. To do that, use the `loadusersymbols` macro to load the program's symbol table:

```
loadusersymbols program
```

where *program* is the name of the program's executable (in the host file system, not in the Pintos file system). For example, you may issue:

```
(gdb) loadusersymbols tests/userprog/exec-multiple
add symbol table from file "tests/userprog/exec-multiple" at
    .text_addr = 0x80480a0
(gdb)
```

After this, you should be able to debug the user program the same way you would the kernel, by placing breakpoints, inspecting data, etc. Your actions apply to every user program running in Pintos, not just to the one you want to debug, so be careful in interpreting the results: GDB does not know which process is currently active (because that is an abstraction the Pintos kernel creates). Also, a name that appears in both the kernel and the user program will actually refer to the kernel name. (The latter problem can be avoided by giving the user executable name on the GDB command line, instead of '`kernel.o`', and then using `loadusersymbols` to load '`kernel.o`'.) `loadusersymbols` is implemented via GDB's `add-symbol-file` command.

B.5.3 FAQ

GDB can't connect to Bochs.

If the `target remote` command fails, then make sure that both GDB and `pintos` are running on the same machine by running `hostname` in each terminal. If the names printed differ, then you need to open a new terminal for GDB on the machine running `pintos`.

GDB doesn't recognize any of the macros.

If you start GDB with `pintos-gdb`, it should load the Pintos macros automatically. If you start GDB some other way, then you must issue the command `source` *`pintosdir`*`/src/misc/gdb-macros`, where *pintosdir* is the root of your Pintos directory, before you can use them.

Can I debug Pintos with DDD?

Yes, you can. DDD invokes GDB as a subprocess, so you'll need to tell it to invokes `pintos-gdb` instead:

```
ddd --gdb --debugger pintos-gdb
```

Can I use GDB inside Emacs?

Yes, you can. Emacs has special support for running GDB as a subprocess. Type *`M-x gdb`* and enter your `pintos-gdb` command at the prompt. The Emacs manual has information on how to use its debugging features in a section titled "Debuggers."

GDB is doing something weird.

> If you notice strange behavior while using GDB, there are three possibilities: a bug in your modified Pintos, a bug in Bochs's interface to GDB or in GDB itself, or a bug in the original Pintos code. The first and second are quite likely, and you should seriously consider both. We hope that the third is less likely, but it is also possible.

B.6 Triple Faults

When a CPU exception handler, such as a page fault handler, cannot be invoked because it is missing or defective, the CPU will try to invoke the "double fault" handler. If the double fault handler is itself missing or defective, that's called a "triple fault." A triple fault causes an immediate CPU reset.

Thus, if you get yourself into a situation where the machine reboots in a loop, that's probably a "triple fault." In a triple fault situation, you might not be able to use `printf()` for debugging, because the reboots might be happening even before everything needed for `printf()` is initialized.

There are at least two ways to debug triple faults. First, you can run Pintos in Bochs under GDB (see Section B.5 [GDB], page 57). If Bochs has been built properly for Pintos, a triple fault under GDB will cause it to print the message "Triple fault: stopping for gdb" on the console and break into the debugger. (If Bochs is not running under GDB, a triple fault will still cause it to reboot.) You can then inspect where Pintos stopped, which is where the triple fault occurred.

Another option is what I call "debugging by infinite loop." Pick a place in the Pintos code, insert the infinite loop `for (;;);` there, and recompile and run. There are two likely possibilities:

- The machine hangs without rebooting. If this happens, you know that the infinite loop is running. That means that whatever caused the reboot must be *after* the place you inserted the infinite loop. Now move the infinite loop later in the code sequence.
- The machine reboots in a loop. If this happens, you know that the machine didn't make it to the infinite loop. Thus, whatever caused the reboot must be *before* the place you inserted the infinite loop. Now move the infinite loop earlier in the code sequence.

If you move around the infinite loop in a "binary search" fashion, you can use this technique to pin down the exact spot that everything goes wrong. It should only take a few minutes at most.

B.7 Modifying Bochs

An advanced debugging technique is to modify and recompile the simulator. This proves useful when the simulated hardware has more information than it makes available to the OS. For example, page faults have a long list of potential causes, but the hardware does not report to the OS exactly which one is the particular cause. Furthermore, a bug in the kernel's handling of page faults can easily lead to recursive faults, but a "triple fault" will cause the CPU to reset itself, which is hardly conducive to debugging.

In a case like this, you might appreciate being able to make Bochs print out more debug information, such as the exact type of fault that occurred. It's not very hard. You start by retrieving the source code for Bochs 2.2.6 from http://bochs.sourceforge.net and saving the file 'bochs-2.2.6.tar.gz' into a directory. The script 'pintos/src/misc/bochs-2.2.6-build.sh' applies a number of patches contained in 'pintos/src/misc' to the Bochs tree, then builds Bochs and installs it in a directory of your choice. Run this script without arguments to learn usage instructions. To use your 'bochs' binary with pintos, put it in your PATH, and make sure that it is earlier than '/usr/class/cs140/`uname -m`/bin/bochs'.

Of course, to get any good out of this you'll have to actually modify Bochs. Instructions for doing this are firmly out of the scope of this document. However, if you want to debug page faults as suggested above, a good place to start adding printf()s is BX_CPU_C::dtranslate_linear() in 'cpu/paging.cc'.

B.8 Tips

The page allocator in 'threads/palloc.c' and the block allocator in 'threads/malloc.c' clear all the bytes in memory to 0xcc at time of free. Thus, if you see an attempt to dereference a pointer like 0xcccccccc, or some other reference to 0xcc, there's a good chance you're trying to reuse a page that's already been freed. Also, byte 0xcc is the CPU opcode for "invoke interrupt 3," so if you see an error like Interrupt 0x03 (#BP Breakpoint Exception), then Pintos tried to execute code in a freed page or block.

An assertion failure on the expression sec_no < d->capacity indicates that Pintos tried to access a file through an inode that has been closed and freed. Freeing an inode clears its starting sector number to 0xcccccccc, which is not a valid sector number for disks smaller than about 1.6 TB.

Appendix C Installing Pintos

This chapter explains how to install a Pintos development environment on your own machine. If you are using a Pintos development environment that has been set up by someone else, you do not need to read this chapter or follow these instructions.

The Pintos development environment is targeted at Unix-like systems. It has been most extensively tested on GNU/Linux, in particular the Debian and Ubuntu distributions, and Solaris. It is not designed to install under any form of Windows.

Prerequisites for installing a Pintos development environment include the following, on top of standard Unix utilities:

- Required: GCC. Version 4.0 or later is preferred. Version 3.3 or later should work. If the host machine has an 80x86 processor, then GCC should be available as `gcc`; otherwise, an 80x86 cross-compiler should be available as `i386-elf-gcc`. A sample set of commands for installing GCC 3.3.6 as a cross-compiler are included in '`src/misc/gcc-3.3.6-cross-howto`'.
- Required: GNU binutils. Pintos uses `addr2line`, `ar`, `ld`, `objcopy`, and `ranlib`. If the host machine is not an 80x86, versions targeting 80x86 should be available with an '`i386-elf-`' prefix.
- Required: Perl. Version 5.8.0 or later is preferred. Version 5.6.1 or later should work.
- Required: GNU make, version 3.80 or later.
- Recommended: QEMU, version 0.11.0 or later. If QEMU is not available, Bochs can be used, but its slowness is frustrating.
- Recommended: GDB. GDB is helpful in debugging (see Section B.5 [GDB], page 57). If the host machine is not an 80x86, a version of GDB targeting 80x86 should be available as '`i386-elf-gdb`'.
- Recommended: X. Being able to use an X server makes the virtual machine feel more like a physical machine, but it is not strictly necessary.
- Optional: Texinfo, version 4.5 or later. Texinfo is required to build the PDF version of the documentation.
- Optional: TeX. Also required to build the PDF version of the documentation.
- Optional: VMware Player. This is a third platform that can also be used to test Pintos.

Once these prerequisites are available, follow these instructions to install Pintos:

1. Install Bochs, version 2.2.6, as described below (see Section C.1 [Building Bochs for Pintos], page 66).
2. Install scripts from '`src/utils`'. Copy '`backtrace`', '`pintos`', '`pintos-gdb`', '`pintos-mkdisk`', '`pintos-set-cmdline`', and '`Pintos.pm`' into the default `PATH`.
3. Install '`src/misc/gdb-macros`' in a public location. Then use a text editor to edit the installed copy of '`pintos-gdb`', changing the definition of `GDBMACROS` to point to where you installed '`gdb-macros`'. Test the installation by running `pintos-gdb` without any arguments. If it does not complain about missing '`gdb-macros`', it is installed correctly.
4. Compile the remaining Pintos utilities by typing `make` in '`src/utils`'. Install '`squish-pty`' somewhere in `PATH`. To support VMware Player, install '`squish-unix`'. If your Perl is older than version 5.8.0, also install '`setitimer-helper`'; otherwise, it is unneeded.

5. Pintos should now be ready for use. If you have the Pintos reference solutions, which are provided only to faculty and their teaching assistants, then you may test your installation by running `make check` in the top-level '`tests`' directory. The tests take between 20 minutes and 1 hour to run, depending on the speed of your hardware.
6. Optional: Build the documentation, by running `make dist` in the top-level '`doc`' directory. This creates a '`WWW`' subdirectory within '`doc`' that contains HTML and PDF versions of the documentation, plus the design document templates and various hardware specifications referenced by the documentation. Building the PDF version of the manual requires Texinfo and $\TeX$ (see above). You may install '`WWW`' wherever you find most useful.

   The '`doc`' directory is not included in the '`.tar.gz`' distributed for Pintos. It is in the Pintos CVS tree available via `:pserver:anonymous@footstool.stanford.edu:/var/lib/cvs`, in the `pintos` module. The CVS tree is *not* the authoritative source for Stanford course materials, which should be obtained from the course website.

C.1 Building Bochs for Pintos

Upstream Bochs has bugs and warts that should be fixed when used with Pintos. Thus, Bochs should be installed manually for use with Pintos, instead of using the packaged version of Bochs included with an operating system distribution.

Two different Bochs binaries should be installed. One, named simply `bochs`, should have the GDB stub enabled, by passing '`--enable-gdb-stub`' to the Bochs `configure` script. The other, named `bochs-dbg`, should have the internal debugger enabled, by passing '`--enable-debugger`' to `configure`. (The `pintos` script selects a binary based on the options passed to it.) In each case, the X, terminal, and "no GUI" interfaces should be configured, by passing '`--with-x --with-x11 --with-term --with-nogui`' to `configure`.

This version of Pintos is designed for use with Bochs 2.2.6. A number of patches for this version of Bochs are included in '`src/misc`':

'`bochs-2.2.6-big-endian.patch`'
: Makes the GDB stubs work on big-endian systems such as Solaris/Sparc, by doing proper byteswapping. It should be harmless elsewhere.

'`bochs-2.2.6-jitter.patch`'
: Adds the "jitter" feature, in which timer interrupts are delivered at random intervals (see Section 1.2.4 [Debugging versus Testing], page 5).

'`bochs-2.2.6-triple-fault.patch`'
: Causes Bochs to break to GDB when a triple fault occurs and the GDB stub is active (see Section B.6 [Triple Faults], page 63).

'`bochs-2.2.6-ms-extensions.patch`'
: Needed for Bochs to compile with GCC on some hosts. Probably harmless elsewhere.

'`bochs-2.2.6-solaris-tty.patch`'
: Needed for Bochs to compile in terminal support on Solaris hosts. Probably harmless elsewhere.

'`bochs-2.2.6-page-fault-segv.patch`'
: Makes the GDB stub report a SIGSEGV to the debugger when a page-fault exception occurs, instead of "signal 0." The former can be ignored with `handle SIGSEGV nostop` but the latter cannot.

'`bochs-2.2.6-paranoia.patch`'
: Fixes compile error with modern versions of GCC.

'`bochs-2.2.6-solaris-link.patch`'
: Needed on Solaris hosts. Do not apply it elsewhere.

To apply all the patches, `cd` into the Bochs directory, then type:

```
patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-big-endian.patch
patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-jitter.patch
patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-triple-fault.patch
patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-ms-extensions.patch
patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-solaris-tty.patch
patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-page-fault-segv.patch
patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-paranoia.patch
patch -p1 < $PINTOSDIR/src/misc/bochs-2.2.6-solaris-link.patch
```

You will have to supply the proper `$PINTOSDIR`, of course. You can use `patch`'s '`--dry-run`' option if you want to test whether the patches would apply cleanly before trying to apply them.

Sample commands to build and install Bochs for Pintos are supplied in '`src/misc/bochs-2.2.6-build.sh`'.

Bibliography

Hardware References

[IA32-v1]. IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture. Basic 80x86 architecture and programming environment. Available via `developer.intel.com`. Section numbers in this document refer to revision 18.

[IA32-v2a]. IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference A-M. 80x86 instructions whose names begin with A through M. Available via `developer.intel.com`. Section numbers in this document refer to revision 18.

[IA32-v2b]. IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference N-Z. 80x86 instructions whose names begin with N through Z. Available via `developer.intel.com`. Section numbers in this document refer to revision 18.

[IA32-v3a]. IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide. Operating system support, including segmentation, paging, tasks, interrupt and exception handling. Available via `developer.intel.com`. Section numbers in this document refer to revision 18.

[FreeVGA]. FreeVGA Project. Documents the VGA video hardware used in PCs.

[kbd]. Keyboard scancodes. Documents PC keyboard interface.

[ATA-3]. AT Attachment-3 Interface (ATA-3) Working Draft. Draft of an old version of the ATA aka IDE interface for the disks used in most desktop PCs.

[PC16550D]. National Semiconductor PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs. Datasheet for a chip used for PC serial ports.

[8254]. Intel 8254 Programmable Interval Timer. Datasheet for PC timer chip.

[8259A]. Intel 8259A Programmable Interrupt Controller (8259A/8259A-2). Datasheet for PC interrupt controller chip.

[MC146818A]. Motorola MC146818A Real Time Clock Plus Ram (RTC). Datasheet for PC real-time clock chip.

Software References

[ELF1]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book I: Executable and Linking Format. The ubiquitous format for executables in modern Unix systems.

[ELF2]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book II: Processor Specific (Intel Architecture). 80x86-specific parts of ELF.

[ELF3]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book III: Operating System Specific (UNIX System V Release 4). Unix-specific parts of ELF.

[SysV-ABI]. System V Application Binary Interface: Edition 4.1. Specifies how applications interface with the OS under Unix.

[SysV-i386]. System V Application Binary Interface: Intel386 Architecture Processor Supplement: Fourth Edition. 80x86-specific parts of the Unix interface.

[SysV-ABI-update]. System V Application Binary Interface—DRAFT—24 April 2001. A draft of a revised version of [SysV-ABI] which was never completed.

[SUSv3]. The Open Group, Single UNIX Specification V3, 2001.

[Partitions]. A. E. Brouwer, Minimal partition table specification, 1999.

[IntrList]. R. Brown, Ralf Brown's Interrupt List, 2000.

Operating System Design References

[Christopher]. W. A. Christopher, S. J. Procter, T. E. Anderson, *The Nachos instructional operating system.* Proceedings of the USENIX Winter 1993 Conference. `http://portal.acm.org/citation.cfm?id=1267307`.

[Dijkstra]. E. W. Dijkstra, *The structure of the "THE" multiprogramming system.* Communications of the ACM 11(5):341–346, 1968. `http://doi.acm.org/10.1145/363095.363143`.

[Hoare]. C. A. R. Hoare, *Monitors: An Operating System Structuring Concept.* Communications of the ACM, 17(10):549–557, 1974. `http://www.acm.org/classics/feb96/`.

[Lampson]. B. W. Lampson, D. D. Redell, *Experience with processes and monitors in Mesa.* Communications of the ACM, 23(2):105–117, 1980. `http://doi.acm.org/10.1145/358818.358824`.

[McKusick]. M. K. McKusick, K. Bostic, M. J. Karels, J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996.

[Wilson]. P. R. Wilson, M. S. Johnstone, M. Neely, D. Boles, *Dynamic Storage Allocation: A Survey and Critical Review.* International Workshop on Memory Management, 1995. `http://www.cs.utexas.edu/users/oops/papers.html#allocsrv`.

License

Pintos, including its documentation, is subject to the following license:

A few individual files in Pintos were originally derived from other projects, but they have been extensively modified for use in Pintos. The original code falls under the original license, and modifications for Pintos are additionally covered by the Pintos license above.

In particular, code derived from Nachos is subject to the following license: