Pintos Threads Project: Reference Guide

(CSU version*, Spring 2020)

1 Pintos

Pintos is an educational operating system for the x86 architecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way. In the Pintos projects, you and your project team will strengthen its support in all three of these areas.

Pintos could, theoretically, run on a regular IBM-compatible PC. Unfortunately, it is impractical to supply every CIS345/545 student a dedicated PC for use with Pintos. Therefore, we will run Pintos projects in a system emulator, that is, a program that emulates an x86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it. Emulators also give you the ability to inspect and debug an operating system while it runs. In this class, we will use the Bochs emulator.

1.1 Source Tree

After you extract the file `~cis345s/pub/pintos_csu.tar.gz`, you can find the following directories under the directory `pintos_csu/src`:

`threads/`
The base Pintos kernel. Most of the modifications you will make for this project will be in this directory.

`userprog/`
The user program loader (future System Calls project).

`vm/`
Virtual memory (future Memory Management project).

`filesys/`
The Pintos file system (future File System project)

`devices/`
Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in this project.

`lib/`
An implementation of a subset of the C standard library. The code in this directory is compiled into both the Pintos kernel and user programs that run inside it. You can include header files from this directory using the `#include <...>` notation. You should not have to modify this code.

*Revised based on the Pintos projects at Stanford University and UC Berkeley

`lib/kernel/`
Library functions that are only included in the Pintos kernel (not the user programs). It contains implementations of some data types that you should use in your kernel code: bitmaps, doubly linked lists, and hash tables. In the kernel, headers in this directory can be included using the `#include <...>` notation.

`lib/user/`
Library functions that are included only in Pintos user programs (not the kernel). In user programs, headers in this directory can be included using the `#include <...>` notation.

`tests/`
Tests for each project. You can add extra tests, but do not modify the given tests.

`examples/`
Example user programs that you can use in the project of System Calls.

`Makefile.build`
Describes how to build the kernel. Modify this file if you would like to add source code.

1.2 Building Pintos

This section describes the interesting files inside the threads/build directory.

`threads/build/Makefile`
A copy of Makefile.build. Do not change this file, because your changes will be overwritten if you make clean and re-compile. Make your changes to Makefile.build instead.

`threads/build/kernel.o`
Object file for the entire kernel. This is the result of linking object files compiled from each individual kernel source file into a single object file. It contains debug information, so you can run GDB or backtrace on it.

`threads/build/kernel.bin`
Memory image of the kernel, that is, the exact bytes loaded into memory to run the Pintos kernel. This is just kernel.o with debug information stripped out, which saves a lot of space, which in turn keeps the kernel from bumping up against a 512 kB size limit imposed by the kernel loader's design.

`threads/build/loader.bin`
Memory image for the kernel loader, a small chunk of code written in assembly language that reads the kernel from disk into memory and starts it up. It is exactly 512 bytes long, a size fixed by the PC BIOS. Subdirectories of build contain object files (.o) and dependency files (.d), both produced by the compiler. The dependency files tell make which source files need to be recompiled when other source or header files are changed.

1.3 Running Pintos

We have supplied a program, called pintos, for running Pintos in a emulators. The Pintos kernel takes a list of arguments, which tell the kernel what actions to perform. These actions are specified in the file `threads/init.c` on line 271 and look something like this.

```
static const struct action actions[] =
{
{"run", 2, run_task},
```

```
#ifdef FILESYS
{"ls", 1, fsutil_ls},
{"cat", 2, fsutil_cat},
{"rm", 2, fsutil_rm},
{"extract", 1, fsutil_extract},
{"append", 2, fsutil_append},
#endif
{NULL, 0, NULL},
};
```

The preprocessor macro `FILESYS` was not defined when we built Pintos earlier (it will be enabled the future projects). So, we can only take one action: `run`. This `run` action will run a test specified by the next command-line argument. The number next to each action's name tells Pintos how many arguments there are (including the name of the action itself). For example, the `run` action's arguments are `"run <test_name>"`. The tests that you can run are the file names in `tests/threads` (without the file extensions).

Let's try it out! First `cd` into the threads directory. Then run `"pintos run alarm-multiple"` which passes the arguments `"run alarm-multiple"` to the Pintos kernel. In these arguments, run instructs the kernel to run a test named `alarm-multiple`.

2 Threads

2.1 Understanding Threads

The first step is to trace the code for the thread system. Pintos already implements thread creation and thread completion, a simple scheduler to switch between threads, and synchronization primitives (semaphores and locks).

Some of this code might seem slightly mysterious. You can read through parts of the source code to see what's going on. If you like, you can add calls to `printf()` almost anywhere, then recompile and run to see what happens and in what order. You can also run the kernel in a debugger and set breakpoints at interesting spots, step through code and examine data, and so on.

When a thread is created, the creator specifies a function for the thread to run, as one of the arguments to `thread_create()`. The first time the thread is scheduled and runs, it starts executing from the beginning of that function. When the function returns, the thread terminates. Each thread, therefore, acts like a mini-program running inside Pintos, with the function passed to `thread_create()` acting like `main()`

At any given time, exactly one thread runs and the rest become inactive. The scheduler decides which thread to run next. (If no thread is ready to run, then the special "idle" thread runs.)

The mechanics of a context switch are in `threads/switch.S`, which is x86 assembly code. It saves the state of the currently running thread and restores the state of the next thread onto the CPU.

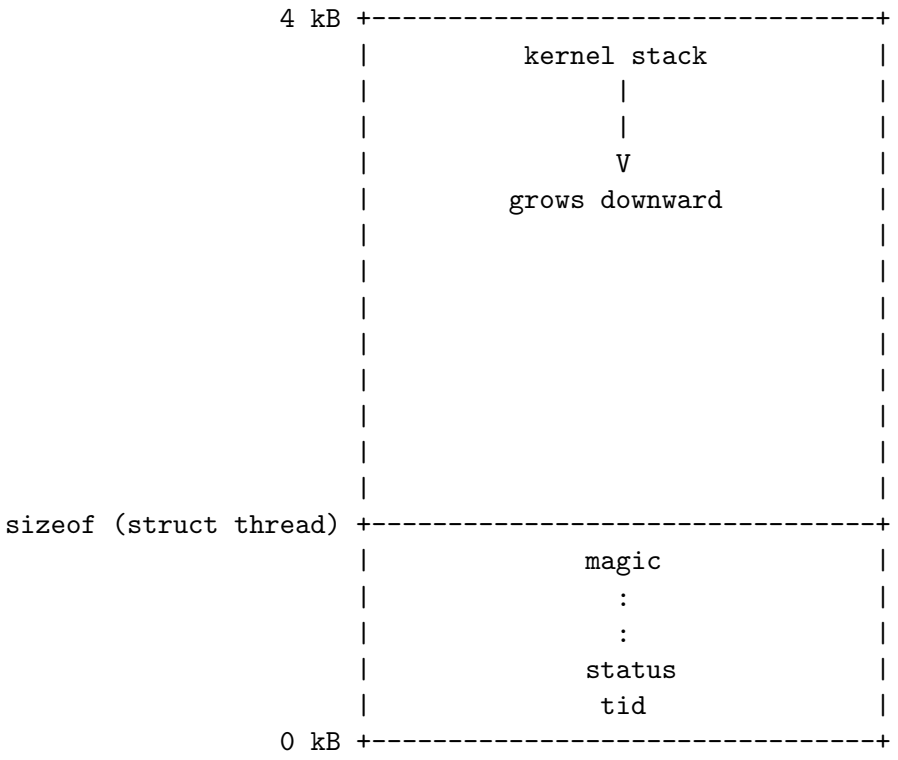
Using GDB, try tracing through a context switch to see what happens. You can set a breakpoint on `schedule()` to start out, and then single-step from there (use `"step"` instead of `next"`). Be sure to keep track of each thread's address and state, and what procedures are on the call stack for each thread (try `"backtrace"`). You will notice that when one thread calls `switch_threads()`, another thread starts running, and the first thing the new thread does is to return from `switch_threads()`. You will understand

the thread system once you understand why and how the `switch_threads()` that gets called is different from the `switch_threads()` that returns.

2.2 The Thread Struct

Each thread struct represents either a kernel thread or a user process. In each of the projects, you will have to add your own members to the thread struct. You may also need to change or delete the definitions of existing members.

Every thread struct occupies the beginning of its own 4KiB 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:



This layout 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

- **Member of struct thread:** `tid_t tid`
  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.

- **Member of struct thread:** `enum thread_status status`
  The thread's state, one of the following:

  - **Thread State:** `THRD_RUNNING`
    The thread is running. Exactly one thread is running at a given time. The `thread_current()` function returns the running thread.
  - **Thread State:** `THRD_READY`
    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 State:** `THRD_BLOCKED`
    The thread is basically waiting for a lock or a semaphore. The thread won't be scheduled again until it transitions to the `THRD_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.
  - **Thread State:** `THRD_SLEEP`
    The thread is waiting for a timer. That is, when a thread invokes `timer_sleep()`, its state will be changed to `THRD_SLEEP` and enters a timer queue. This will be implemented in this project.
  - **Thread State:** `THRD_DYING`
    The thread has exited and will be destroyed by the scheduler after switching to the next thread.

- **Member of struct thread:** `char name[16]`
  The thread's name as a string, or at least the first few characters of it.

- **Member of struct thread:** `uint8_t *stack`
  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.

- **Member of struct thread:** `int priority`
  A thread priority, ranging from `PRIO_MIN` (0) to `PRIO_MAX` (63). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. Pintos currently ignores these priorities, but you will implement priority scheduling in this project.

- **Member of struct thread:** `struct list_elem allelem`
  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.

- **Member of struct thread:** `struct list_elem elem`
  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.

- **Member of struct thread:** `uint32_t *pagedir`
  (Only used in the future projects.) The page table for the process, if this is a user process.

- **Member of struct thread:** `unsigned magic`
  Always set to `THRD_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 `THRD_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.

2.3 Thread Functions

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

- **Function:** `void thread_init(void)`
  Called by `main()` to initialize the thread system. Its main purpose is to create a `struct thread` for Pintos' 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.

- **Function:** `void thread_start(void)`
  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()`.

- **Function:** `void thread_tick(void)`
  Called by the timer interrupt at each timer tick. It keeps track of thread statistics and triggers the scheduler when a time slice expires.

- **Function:** `void thread_print_stats(void)`
  Called during Pintos shutdown to print thread statistics.

- **Function:** `tid_t thread_create(const char *name, int priority, thread_func *func, void *aux)`
  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 threadss `thread struct` and stack and initializes its members, then it sets up a set of fake stack frames for it. The thread is initialized in the blocked state, then unblocked just before returning, which allows the new thread to be scheduled.

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

- **Function:** `void thread_block(void)`
  Transitions the running thread from the running state to the blocked state. 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.

- **Function:** `void thread_unblock(struct thread *thread)`
  Transitions thread, which must be in the blocked state, to the ready state, allowing it to resume running. This is called when the event that the thread is waiting for occurs, e.g. when the lock/semaphore that the thread is waiting on becomes available.

- **Function:** `struct thread *thread_current(void)`
  Returns the running thread.

- **Function:** `tid_t thread_tid(void)`
  Returns the running thread's thread id. Equivalent to `thread_current()->tid`.

- **Function:** `const char *thread_name(void)`
  Returns the running thread's name. Equivalent to `thread_current()->name`.

- **Function:** `void thread_exit(void) NO_RETURN`
  Causes the current thread to exit. Never returns, hence `NO_RETURN`.

- **Function:** `void thread_yield(void)`
  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.

- **Function:** `void thread_foreach (thread_action_func *action, void *aux)`
  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()`:

  - **Type:** `void thread_action_func(struct thread *thread, void *aux)`
    Performs some action on a thread, given `aux`.

- **Function:** `int thread_get_priority(void)`
  Function: `void thread_set_priority (int new_priority)`
  Stub to set and get thread priority.

2.4 Thread Switching

The `schedule()` function 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, calls `thread_schedule_tail()` (this special case is why the function `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.

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.

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.

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`.

- **Type:** `enum intr_level` One of `INTR_OFF` or `INTR_ON`, denoting that interrupts are disabled or enabled, respectively.
- **Function:** `enum intr_level intr_get_level (void)`
  Returns the current interrupt state.
- **Function:** `enum intr_level intr_set_level (enum intr_level level)`
  Turns interrupts on or off according to level. Returns the previous interrupt state.
- **Function:** `enum intr_level intr_enable (void)` Turns interrupts on. Returns the previous interrupt state.
- **Function:** `enum intr_level intr_disable (void)` Turns interrupts off. Returns the previous interrupt state.

This project only requires accessing a little bit of thread state from interrupt handlers. For the alarm clock, the timer interrupt needs to wake up sleeping threads. In the advanced scheduler, the timer interrupt needs to access a few global and per-thread variables. When you access these variables from kernel threads, you will need to disable interrupts to prevent the timer interrupt from interfering.

When you do turn off interrupts, take care to do so for the least amount of code possible, or you can end up losing important things such as timer ticks or input events. Turning off interrupts also increases the interrupt handling latency, which can make a machine feel sluggish if taken too far.

The synchronization primitives themselves in `synch.c` are implemented by disabling interrupts. You may need to increase the amount of code that runs with interrupts disabled here, but you should still try to keep it to a minimum.

Disabling interrupts can be useful for debugging, if you want to make sure that a section of code is not interrupted. You should remove debugging code before turning in your project. (Don't just comment it out, because that can make the code difficult to read.)

There should be no busy waiting in your submission. A tight loop that calls `thread_yield()` is one form of busy waiting.

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 0 or values larger than 1.

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

- **Type:** `struct semaphore`
  Represents a semaphore.
- **Function:** `void sema_init (struct semaphore *sema, unsigned value)`
  Initializes sema as a new semaphore with the given initial value.
- **Function:** `void sema_down (struct semaphore *sema)`
  Executes the ”down” or ”P” operation on sema, waiting for its value to become positive and then decrementing it by one.
- **Function:** `bool sema_try_down (struct semaphore *sema)`
  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.
- **Function:** `void sema_up (struct semaphore *sema)`
  Executes the ”up” o”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.

Semaphores are internally built out of disabling interrupt 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`.

3.3 Locks

A **lock** is like a semaphore with an initial value of 1. A lock's equivalent of ”up: is called ”release”, and the ”down” operation is called ”acquire”.

Compared to a semaphore, a lock in Pintos 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`.

- **Type:** `struct lock`
  Represents a lock.

- **Function:** `void lock_init (struct lock *lock)`
  Initializes lock as a new lock. The lock is not initially owned by any thread.

- **Function:** `void lock_acquire (struct lock *lock)`
  Acquires lock for the current thread, first waiting for any current owner to release it if necessary.

- **Function:** `bool lock_try_acquire (struct lock *lock)`
  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.

- **Function:** `void lock_release (struct lock *lock)`
  Releases lock, which the current thread must own.

- **Function:** `bool lock_held_by_current_thread (const struct lock *lock)` 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.

4 Linked Lists

Pintos contains a linked list data structure in `lib/kernel/list.h` that is used for many different purposes. This linked list implementation is different from most other linked list implementations you may have encountered, because **it does not use any dynamic memory allocation**.

```
/* List element. */
struct list_elem
{
struct list_elem *prev; /* Previous list element. */
struct list_elem *next; /* Next list element. */
};
/* List. */
struct list
{
struct list_elem head; /* List head. */
struct list_elem tail; /* List tail. */
};
```

In a Pintos linked list, each list element contains a `"struct list_elem"`, which contains the pointers to the next and previous element. Because the list elements themselves have enough space to hold the prev and next pointers, we don't need to allocate any extra space to support our linked list. Here is an example of a linked list element which can hold an integer:

```
/* Integer linked list */
struct int_list_elem
{
int value;
```

```
struct list_elem elem;
};
```

Next, you must create a "truct list" to represent the whole list. Initialize it with list_init().

```
/* Declare and initialize a list */
struct list my_list;
list_init (&my_list);
```

Now, you can declare a list element and add it to the end of the list. Notice that the second argument of list_push_back() is the address of a struct list_elem, not the struct int_list_elem itself.

```
/* Declare a list element. */
struct int_list_elem three = {3, {NULL, NULL}};
/* Add it to the list */
list_push_back (&my_list, &three.elem);
```

We can use the list_entry() macro to convert a generic "struct list_elem" into our custom "struct int_list_elem" type. Then, we can grab the ”value” attribute and print it out:

```
/* Fetch elements from the list */
struct list_elem *first_list_element = list_begin (&my_list);
struct int_list_elem *first_integer = list_entry (first_list_element,
                                                  struct int_list_elem,
                                                  elem);
printf("The first element is: %d\n", first_integer->value);
```

By storing the prev and next pointers inside the structs themselves, we can avoid creating new ”linked list element” containers. However, this also means that a list_elem can only be part of one list a time. Additionally, our list should be homogeneous (it should only contain one type of element).

The list_entry() macro works by computing the offset of the elem field inside of "struct int_list_elem". In our example, this offset is 4 bytes. To convert a pointer to a generic "struct list_elem" to a pointer to our custom "struct int_list_elem", the list_entry() just needs to subtract 4 bytes! (It also casts the pointer, in order to satisfy the C type system.)

Linked lists have 2 sentinel elements: the head and tail elements of the "struct list" These sentinel elements can be distinguished by their NULL pointer values. Make sure to distinguish between functions that return the first actual element of a list and functions that return the sentinel head element of the list.

There are also functions that find min/max element in a link list. These functions require you to provide a list element comparison function.

5 Debugging Pintos

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

5.1 `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.

5.2 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.

Macro: ASSERT (expression)
: 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.

5.3 GDB

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

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

and issue the following GDB command:

```
target remote localhost:1234
```

Now GDB is connected to the emulator over a local network connection. You can now issue any normal GDB commands. If you issue the "c" command, the emulated 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.