ESTR 3102: Pintos Guide

Written by: Stephen Tsung-Han Sher
Modified by: Baotong Lu

Perface

This guide is modified from the documentation written by Stephen Tsung-Han Sher on USC's website. We thank Stephen Tsung-Han Sher for granting us the right to modify the documentation and use it as the guide in our course. This edited version is only used for the teaching of this course(ESTR 3102, CUHK). If you want to use this guide for other purposes, please contact Stephen Tsung-Han Sher.

The guide serves to help you along the projects of Pintos for ESTR 3102. The aim of this document is to minimize the amount of time you spend being confused about syntax or Pintos in general and more time designing and implementing the projects.

In this document you will find a general guideline to the projects, as well as hints and tips for the trickier aspects of Pintos. This guide will **not** explain every single detail of Pintos; this guide will leave some aspects of Pintos for you to discover and figure out yourself.

Since this guide is **not** written by an expert on Pintos, it is always a better idea to refer to the official Pintos documentation on Stanford's website. This guide is not a stand-alone documentation, but a supplement to the Stanford's documentation.

You are highly recommended to read this document and Stanford's official Pintos documentation simultaneously when you start working on Pintos.

If you find anything incorrect or needs to be updated in this guide, please contact the TA first.

Contents

1 Introduction

1.1 Get the Pintos

You should ignore the content of installing Pintos in Stanford's official Pintos documentation. The virtual machine(VM) we provide for you has already installed the Pintos. If you have not gotten the VM, please follow the steps in Lab 1 to setup the VM on your own computer.

The Pintos is installed under **'/home/csci3150/os-pintos/pintos/'**. Do not modify the installation path or any environment variable of Pintos, otherwise your Pintos cannot run normally. Please refer to 1.1.1 Source Tree Overview section of the official documentation for a general understanding of Pintos.

Sublime Text, which provides the good interface to read and write code, is also installed on this virtual machine. You may open the pintos folder by Sublime Text to improve your efficiency.

1.2 Build the Pintos

As the next step, build the source code supplied for the first project:

1) Open the terminal and **cd** into the directory '/home/csci3150/os-pintos/pintos/src/threads/'

2) Execute the following command:

```
~$ make
```

This will create a **'build'** directory under 'threads' directory. The kernel is built inside 'build' directory. After the above step, you can find the file 'kernel.o' inside 'build' directory. It is the object file for the entire kernel.

1.3 Running Pintos

The Pintos developer supplied a program for conveniently running Pintos in a simulator, called **pintos**. In the simplest case, you can invoke pintos as **pintos arguments ...**. Each argument is passed to the Pintos kernel for it to act on.

Next step, running your first program on Pintos:

1) **cd** into '/home/csci3150/os-pintos/pintos/src/threads/build/'

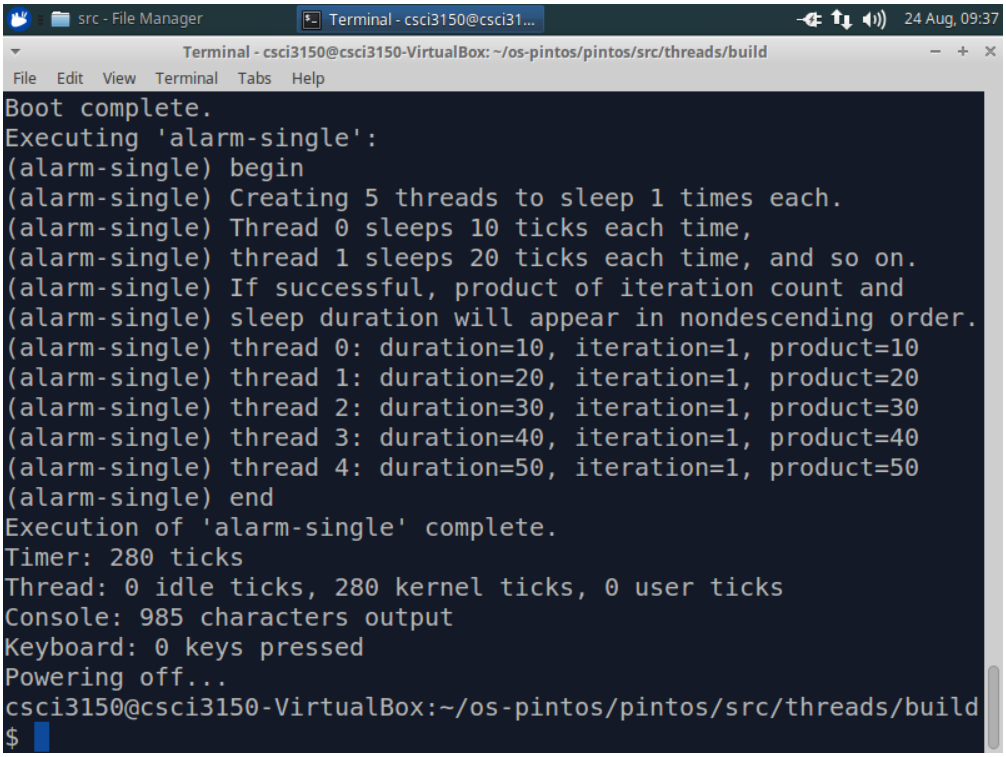
2) Execute the following command:

```
~$ pintos -q run alarm-single
```

This command passes the arguments '-q run alarm-single' to the Pintos kernel. In these arguments, 'run' instructs kernel to run a test and 'alarm-single' is the test to run, '-q' causes Pintos

to exit as soon as the test is done.

If the Pintos works well, the terminal will print the running information of alarm-single test:



Another thing to note is that we use QEMU as the simulator to run Pintos in our provided virtual machine, so please ignore all the related content of another simulator Bochs in Stanford's official documentation. As you imagine, you are running Pintos in a simulator QEMU in a virtual machine XUbuntu in your own machine.

1.4 Testing

Your test result grade will be based on our tests. Each project has several tests, e.g., project 1 has 27 tests. Next we will show how to test your program.

If you want to run all tests of one project, **cd** into the `build` directory of that project(e.g. for project 1, it is `/home/csci3150/os-pintos/pintos/src/threads/build/`), then invoke the following command in this directory:

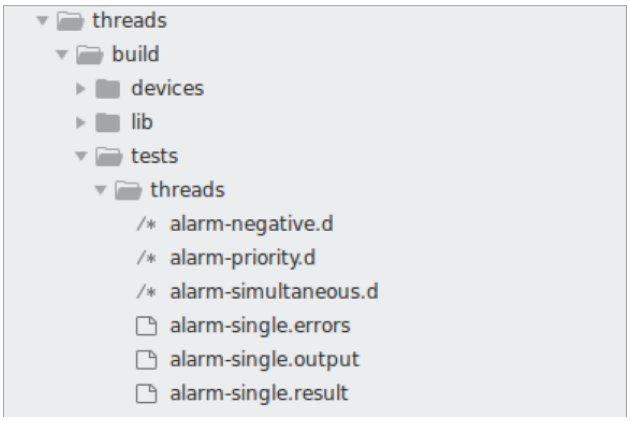
```
~$ make check
```

This will build and run each test and print "pass" or "fail" message for each one. After running all the tests, `make check` also prints a summary of the test results.

You can also run individual tests one at a time. A given test t writes its output to **'t.output'** and the verdict("pass" or "fail") to **'t.result'**. For example, if you only want to run alarm-single test in project 1, you should execute the following command under `build` directory.

```
~$ make tests/threads/alarm-single.result
```

If `make` says that the test result is up-to-date, but you want to re-run it anyway, either run `make clean` under `build` directory or delete the `.output` file by hand. And then you can re-run the above command and get the test information of that single test. You can find `.output` and `.result` file in the `tests` directory under `build` directory of that project. For example, `alarm-single.result` and `alarm-single.output` are under `/home/csci3150/os-pintos/pintos/src/threads/build/tests/threads/`. The hierarchy of the folder is shown in the figure below.



All of the tests(source files) are in directory `/home/csci3150/os-pintos/pintos/src/tests`(Note: different from the above directory). When you want to run a single test, you should first find the correct path of that test in this directory. You can modify some of the tests if that helps in debugging, but we will run the originals when testing your submission.

1.5 Debugging in Pintos

There are two ways recommended to approach debugging in Pintos: printing to console and using GDB. This section will cover both.

1.5.1 Printing to Console

Back in *C++*, you're most likely used to using `std::cout` or `std::cerr` to print to the console to get information on certain variables and program progress. This is a messy way to debug, but definitely a versatile way to do so.

You can definitely do the same with `printf` statements in Pintos; the problem is, in order to pass Pintos tests your console needs to print out very specific messages. These `printf` statements are written already for you in the test files, therefore you won't need to write your own `printf` statements to pass tests[1].

Therefore if you put your own `printf` statements, you will fail the automated Pintos tests even if your project has the proper behavior.

[1]With one exception in Project 2, however this will be pointed out in that section

Instead of going through and commenting out every single `printf` statement to run tests and uncomment the same statements to debug, I recommend you defining a quick boolean macro:

```
#define DEBUG true
```

By putting this macro in front of every debug `printf` statement:

```
if(DEBUG) printf("Your␣debug␣statement␣here");
```

You can toggle on & off all the debugging `printf` statements just by changing the macro you defined.

1.5.2 Running GDB with Pintos

You can find Stanford's official Pintos documentation on debugging here.

Since Pintos is run in Qemu, and Qemu is a simulator, you'll need to remotely connect a GDB application to the Pintos instance in this simulator to debug it. Don't worry, it's not that hard.

Say the test you want to run GDB on is the following:

```
~$ pintos -q  run alarm-single
```

You'll want to run this line with the `-gdb` flag(Note: two '-' symbol in front of `gdb`):

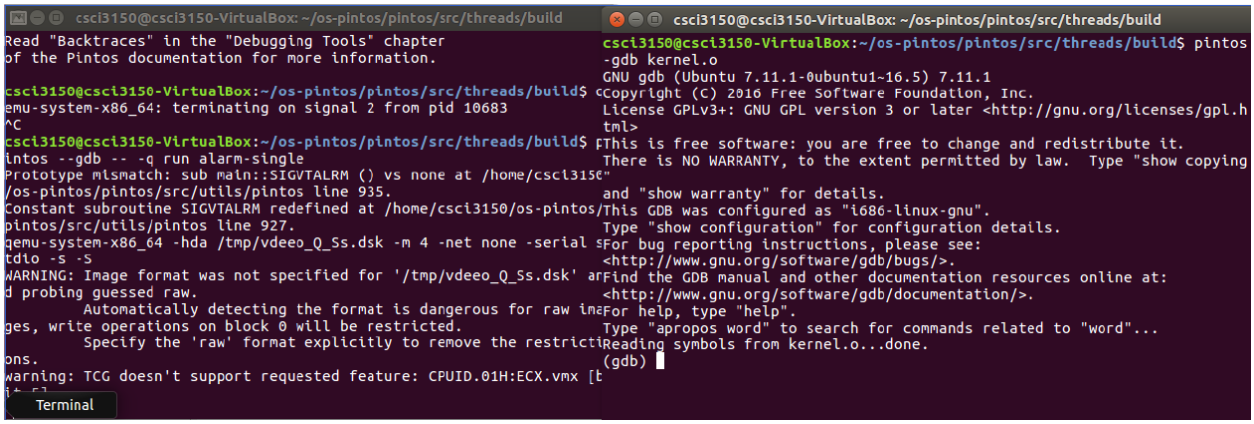
```
~$ pintos --gdb -- -q  run alarm-single
```

Right now the program is suspended, you should see:

```
csci3150@csci3150-VirtualBox:~/os-pintos/pintos/src/threads/build$ pintos --gdb
 -- -q run alarm-single
Prototype mismatch: sub main::SIGVTALRM () vs none at /home/csci3150/os-pintos/
pintos/src/utils/pintos line 935.
Constant subroutine SIGVTALRM redefined at /home/csci3150/os-pintos/pintos/src/
utils/pintos line 927.
qemu-system-x86_64 -hda /tmp/vdeeo_Q_Ss.dsk -m 4 -net none -serial stdio -s -S
WARNING: Image format was not specified for '/tmp/vdeeo_Q_Ss.dsk' and probing g
uessed raw.
         Automatically detecting the format is dangerous for raw images, write
operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
```

Open a separate terminal, navigate to the same directory as where you ran Pintos and run:

```
~$ pintos-gdb kernel.o
```

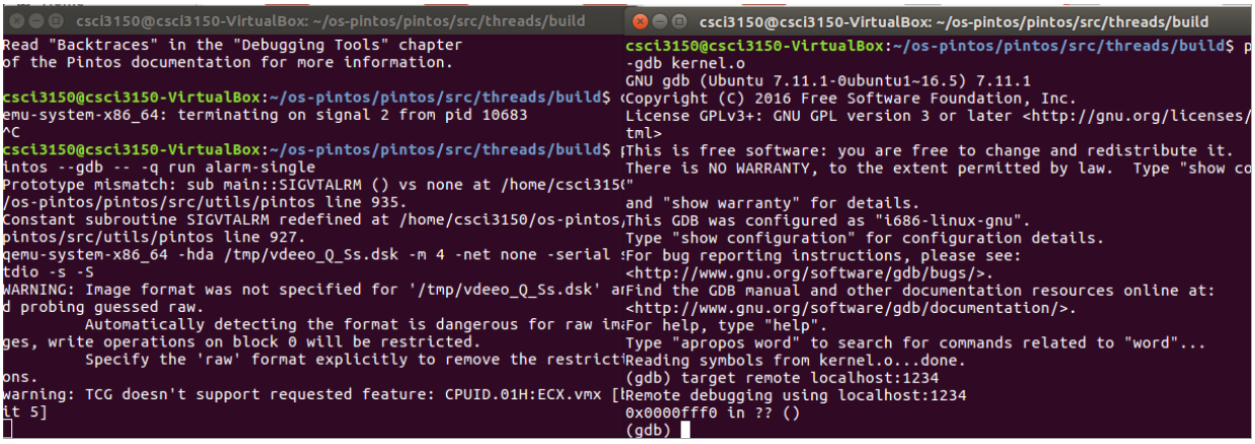


Now in the same terminal running the GDB kernel shell, type the following to connect to the Pintos instance you ran:

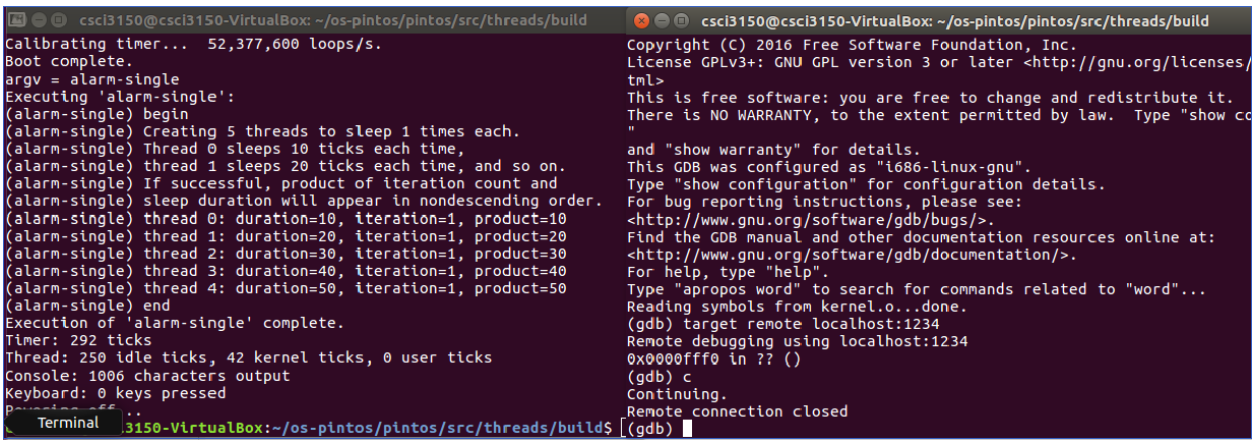
```
~$ target remote localhost:1234
```

Or you can type the following to connect, `debugpintos` is the macro of `target remote localhost:1234`, they have the same effect.

```
~$ debugpintos
```



Now the gdb has connected to the pintos instance in the first terminal. In the second terminal running the GDB shell, you can now call the usual GDB commands.



If you have not used gdb before, you can use Google to search related tutorials.

1.5.3 Common Debug Messages

Here is a common message you will see when running Pintos:

```
~$ Kernel PANIC at ...
```

A kernel panic is effectively the same as a `Segmentation fault` in *C++*; meaning you're trying to access something that you have no access to.

Kernel panics are often caused by assertion errors:

```
~$ Kernel PANIC at ../../threads/thread.c:350 in thread_unblock() : assertion
   t->status == THREAD_BLOCKED failed
```

Assertion statements are effectively a boolean check in which if the check is a `false`, Pintos will crash right away with a kernel panic. Consider the following line:

```
ASSERT (t->status == THREAD_BLOCKED);
```

This means the boolean statement must return a true when this line is executed, otherwise resulting in a kernel panic.

1.6 Grading

We will grade your assignments based on test results and design quality. Your submission should include the source code and design document for the project. Please note that we reserve the right to ask your questions. The questions will help us better understand your solution of the project and test your understanding of Pintos. Next are some requirements for the Pintos projects in this course:

1) Project 1 and project 2 are compulsory, each of which accounts for 20% of the course assessment(i.e. your final grade of this course). You should complete these two projects by yourself.

2) Project 3 and project 4 are bonuses, each of which accounts for 20%! You can choose to do it by yourself or join a team. The number of people in each team cannot exceed three. The members belonging to the same team will get the same score. You need to fill in the information(i.e. name and email) of every group member in the design document. Every group needs to assign one person to submit your group's code and document(No duplicate submission from the different members in the same group). Please make sure that you do enough work in your group, otherwise you will not get the bonus.

3) Maximum score of each project is 100. They will be converted to the corresponding score according to the proportion(e.g. project 1 is 20%) and added to your final score of this course.

4) Project 1 has 27 tests, each of which carries 3.704 mark.

5) Project 2 has 76 tests, each of which carries 1.316 mark;

6) Project 3 has 109 tests. A lot of these tests are exactly the same as the tests you've seen in project 2. The score distribution is uneven. The tests from project 2(75 tests out of 76 tests in original project 2) accounts for 10 mark and each test carries 0.133 mark. The tests that are new in project 3(34 tests in total) account for 90 mark and each test carries 2.647 mark. Unlike Stanford's official documentation, implementing sharing among processes has no extra bonus but we encourage you to implement it.

7) You can build project 4 on top of project 2 or project 3. We encourage you to build on top of project 3 although it does not has extra bonus like Stanford's official documentation. The score discribution is also uneven. The tests from other projects(2 or 3) account for 10 mark. The tests that are new in project 4(46 tests in total) account for 60 mark and each test carries 1.304 mark. Please note that implementing buffer cache is one of 3 significant parts of this project, so buffer cache accounts for 30 mark.

8) The deadline of each project is as follows.

a. Project 1: Monday, December 3, 2018, 10:30AM

b. Project 2: Monday, December 17, 2018, 10:30AM

c. Project 3 and 4: Saturday, December 29, 2018, 10:30AM

9) **Please submit your projects before the deadline, no late submission is allowed**.

10) An incomplete, evasive, or non-responsive design document or one that strays from the template without good reason may be penalized.

11) The submission with no design document will get 0 mark.

1.6.1 Design Document

We will judge your design based on the design document and the source code that you submit. Your entire design document and much of your source code will be checked. We will try to match up your description of the design with the code submitted. Important discrepancies between the

description and the actual code will be penalized.

Before you submit your assignment, please name the design document "DESIGNDOC" with no extention and put it under the directory of that project. To avoid confusion, please follow the instruction below.

a. Put the design document of the project 1 under 'threads' directory(i.e. '/home/csci3150/os-pintos/pintos/src/threads/').

b. Project 2's design document => 'src/userprog' direcotry

c. Project 3's design document => 'src/vm' direcotry

d. Project 4's design document => 'src/filesys' direcotry

It's better to read the design document first before rushing into code writing. You can get some hints from the design document. We provide the design document template for each project. You can download the design document template for each project on the piazza(Project section of resources).

1.6.2 Submission

We will use Github Classroom to collect your code and design document of every porject. As the Pintos has been installed on your virtual machine, we do not use Github Classroom to distribute starter package, so the project repo is empty initially when you clone it to your local directory .

When you want to sumbit a project, please log in your GitHub account first and then go to the corresponding following link to get the new repo under your account for that project.

a. Project 1: https://classroom.github.com/a/Y-ABySmF

b. Project 2: https://classroom.github.com/a/J0qJSSpP

c. Project 3: https://classroom.github.com/a/TMXuaEHo

d. Project 4: https://classroom.github.com/a/nor-0qSJ

If you do not know how to use Github Classroom, please refer to section 2 of warm-up assignment 1. After you clone the project repo to your local directory on your computer, please copy and paste the **'src'** directory(i.e. '/home/csci3150/os-pintos/pintos/src') into your local directory and submit that project to your project repo using git command. Here are a few things to note.

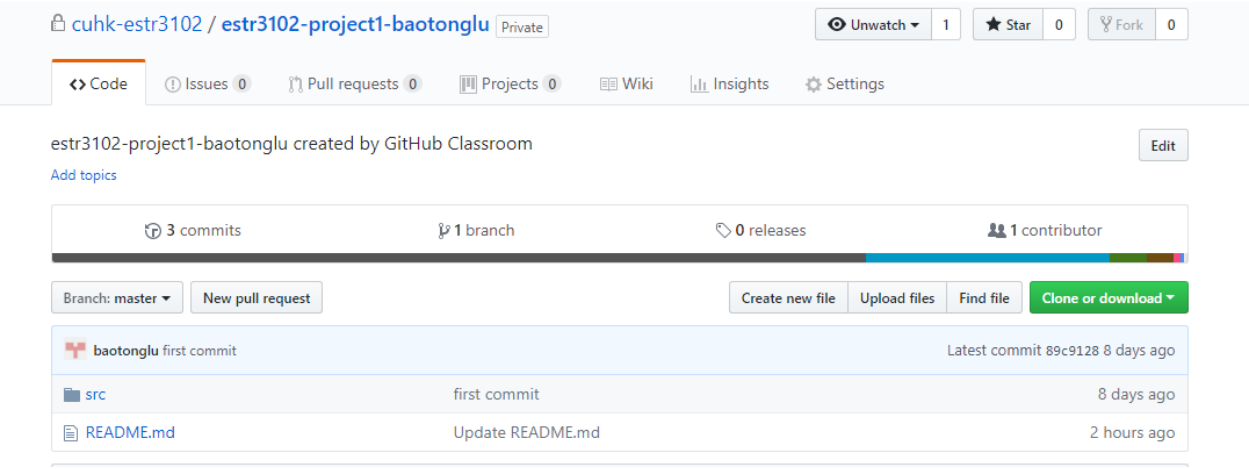
1) Do not just copy and paste the files under 'src' direcotry. The 'src' directory itself should also be copied. Using command 'cp -r' can finish this work in one step.

2) Make sure that the tests of that project can run normally as you expected. You'd better re-run the tests before you submit that project. Do not expect the TA to adjust the settings of your

Pintos(e.g. kernel path[2]) when testing your submission. We will replace the 'src' directory(same installation path as yours, is also '/home/csci3150/os-pintos/pintos/src') in our own VM with your 'src' directory when testing it.

3) Make sure your design document for each project is placed in the correct directory (refer to 'Design Doucment' section of this guide), and named "DESIGNDOC" with no extensions.

After you successfully submit the project 1, your project repo on GitHub should be look like this:



[2]The kernel path of the Pintos in our provided VM points to the directory of project 1, so do not worry when you submit the first project as long as you do not touch the settings of Pintos. We will tell you how to modify the kernel path from project 2.

2 Utilities

This section will take you through the utilities you will find useful throughout the entire Pintos project.

As you read through this section, you will notice that each of these utilities needs to be instantiated and initialized with an `_init` function. As you are doing your project you will notice that there is no clear `main` function for you to instantiate one of these objects, and finding a place to initialize it is even trickier. As a general rule of thumb, you can do the following:

- You can declare global objects for you to use. If you declare them globally, these objects are immediately instantiated once Pintos starts running.
- Call the `_init` functions in other `_init` functions. For example, if you want your thread struct to contain a list, call `list_init` in the `thread_init` function. These `_init` functions are called once and only once for Pintos to initialize necessary procedures for all future instances of this object, thus the best place to call an `_init` function is in other `_init` functions.

2.1 List

List is the linked list implementation in Pintos. You can find the header and implementation file in the `src/lib/kernel` directory. You include the list library with `#include<list.h>`.

Each list element is a struct containing a previous and next pointer:

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

You will notice that there is no data member to store the actual contents of an element. We'll get into this later.

Each list contains a head and a tail pointer:

```
struct list
{
    struct list_elem head;      /* List head. */
    struct list_elem tail;      /* List tail. */
};
```

An empty list will contain a head and tail `list_elem` by default. Instead of detecting a `NULL` pointer for the bounds of the list, this list library uses the head and tail `list_elem`. As far as you're concerned, you don't need to touch the head and tail `list_elem`.

2.1.1 Instantiating and Manipulating `lists`

As mentioned, the `list_elem` does not contain a member to store the contents. Instead, `list_elem` is used as a member of another struct. Every `list` and `list_elem` exists globally due to no class implementation in *C*, thus you cannot use the same `list_elem` for multiple `lists`; for each `list` you want your struct to be stored in, you need to create a unique `list_elem` to use for that `list` and that `list` alone.

In order to use `list`, you'll need two essential steps:

1. Instantiate an instance of `list` and initialize it with `list_init`.
2. Have a struct with a `list_elem` as a member.

Here's a simple example:

```
//For list and list_elem
#include <list.h>
//For malloc
#include "threads/malloc.h"

struct thread
{
    /*    Members    */
    struct list_elem all_elem;
    struct list_elem ready_elem;
};

int main(void)
{
    //For demonstration, assume the ready_thread is ready
    //and the blocked_thread is blocked upon creation
    struct thread* ready_thread = malloc(sizeof(struct thread));
    struct thread* blocked_thread = malloc(sizeof(struct thread));

    //Instantiate
    struct list all_threads_list;
    struct list ready_threads_list;

    //Initialize: You MUST do this step
    list_init(&all_threads_list);
    list_init(&ready_threads_list);

    //Use the lists and list_elems
    //Prototype: void list_push_front (struct list *, struct list_elem *)
    list_push_front(&all_threads_list, &(ready_thread->all_elem));
    list_push_front(&all_threads_list, &(blocked_thread->all_elem));
    list_push_front(&ready_threads_list, &(ready_thread->ready_elem));

    return 0;
}
```

Notice that the `ready_thread` has to exist in both `all_threads_list` and `ready_threads_list`, and thus two `list_elems` were required. If you do not make a unique `list_elem` for each `list`, you will get an assertion error saying that this `list_elem` already exists in another list.

For removal and other functions, you can take a look at `src/lib/kernel/list.h`. Remember that there are no classes, therefore every function is called globally. You'll need to specify which `list` and which `list_elem` you want the function to act on.

2.1.2 Accessing Contents of `lists`

The reason why `list_elems` do not have a member that contains the contents is because templates are not supported by *C*, therefore we don't know how much memory to allocate each `list_elem`. However we still want the functionality of a templated list, which is why we have a struct containing a `list_elem` member. This way when we have a `list_elem`, we can specify the size of the struct wrapping the `list_elem` and acquire the struct containing the data we want.

To do this, we need to use the `list_entry` macro. The definition of this macro is super confusing:

```
#define list_entry(LIST_ELEM, STRUCT, MEMBER)           \
        ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next     \
                     - offsetof (STRUCT, MEMBER.next)))
```

Instead, here's the prototype equivalent of `list_entry` in *C++*:

```
//T is always a struct
template <class T>
T* list_entry(struct list_elem*, struct T, list_elem_name);
```

Following the example in the previous section:

```
//For list and list_elem
#include <list.h>
//For malloc
#include "threads/malloc.h"

struct thread
{
    /*    Members    */
    struct list_elem all_elem;
    struct list_elem ready_elem;
};

int main(void)
{
    /* Same setup as the above section's main function */
    //list_entry returns a pointer to the struct, so we need a pointer to
       capture the return value
    //Gets the front entry of all_theads_list
    struct thread* entry = list_entry(list_front(&all_threads_list), struct
       thread, all_elem);
    //Gets the front entry of ready_threads_list
```

```
    struct thread* next_ready_thread = list_entry(list_front(&
       ready_threads_list), struct thread, ready_elem);

    return 0;
}
```

Notice that if I'm using the `list_elem` from `all_threads_list`, then I pass in the specific member name of the `list_elem` used for that list as the third argument (`all_elem`, not `ready_elem`).

2.1.3 Looping Through Lists

Often times you'll want to loop through a list. This is very similar to what you usually do for linked lists in *C++*; however the syntax can get quite messy. Here's an example you can use as reference:

For Loop

```
//Effectively: for(iterator* iter = list_begin(); iter != list_end(); iter++)
for(struct list_elem* iter = list_begin(&my_list);
     iter != list_end(&my_list);
     iter = list_next(iter))
     {
         //do stuff with iter
         struct list_contents* = list_entry(iter, struct list_contents,
            list_elem);
     }
```

While Loop

```
//Effectively: while(iter != list_end()) { ...; iter++}
struct list_elem* iter = list_begin(&my_list);
while(iter != list_end(&my_list)
{
    //do stuff with iter
    struct list_contents* = list_entry(iter, struct list_contents, list_elem);
    iter = list_next(iter);
}
```

If you are to loop through a list to remove `list_elems` or to free memory, take care that once you remove a `list_elem` from a list, the `list_next` function will give you an assertion error saying that this `list_elem` is no longer an interior element of a list (basically a seg fault). You'll want to be careful with how you construct your loops in this case.

2.2 Function Pointers

You may already familiar with function pointers & passing function as arguments, however this section will help you in the two main ways Pintos uses function pointers.

2.2.1 The `thread_foreach` Function

The `thread_foreach` function will run a function once for each thread in existence (except for threads in the terminating state). The function you pass in has to follow the following prototype:

```
static void my_function(struct thread* t, void* aux);
//or if no auxiliary data needed
static void my_function(struct thread* t, void* aux UNUSED);
```

When you call the `thread_foreach` function, it will loop through all the threads, and pass in each thread as an argument into your function:

```
//From src/lib/kernel/thread.c
for (e = list_begin (&all_list); e != list_end (&all_list);
     e = list_next (e))
{
    struct thread *t = list_entry (e, struct thread, allelem);
    //func is the function you pass into thread_foreach
    func (t, aux);
}
```

Here's an example of using the `thread_foreach` function:

```
//For printf
#include <stdio.h>
//For thread_foreach
#include <threads/thread.h>
static void my_function(struct thread* t, void* aux)
{
    printf("Thread␣%d␣in␣<%s>␣function", t->tid, aux);
}
void foo(void)
{
    /* In this function threads 1 and 3 exits and dies */
    thread_foreach(my_function, "foo");
}
int main(void)
{
    /* Instantiates and initializes 3 threads */
    thread_foreach(my_function, "main");
    foo();
    return 0;
}
```

Output:

```
~$ Thread 1 in <main> function
~$ Thread 2 in <main> function
~$ Thread 3 in <main> function
~$ Thread 2 in <foo> function
```

2.2.2 Comparators & Sorting

Being able to sort stuff is good for you. Besides, you practically spend the entirety of your **CSCI 104** sorting stuff. It wouldn't be surprising if you'll need to sort stuff for Pintos.

Sorting in Pintos requires three steps:

1. Define the prototype of the comparator in the header file
2. Implement the comparator in the implementation file
3. Use the comparator when needed

Defining the Prototype

Mostly you'll be comparing between `list_elems` to sort lists or the likes. Define your prototype in the relevant header file. Your prototype will be defined similar to:

```
bool list_less_comp(const struct list_elem* a,
                    const struct list_elem* b, void* aux UNUSED);
```

If you are not comparing `list_elems`, then feel free to change the prototype as necessary.

Implementing the Comparator

In the *.c* file, you'll want to implement the comparator. Be sure to know whether or not this should be a `less_than` or a `greater_than` comparator. If you're using an existing library they will specify which one they want (for `list.h` they specify a `less_than` comparator). Here's an outline of such:

```
bool list_less_comp(const struct list_elem* a,
                    const struct list_elem* b, void* aux UNUSED)
{
    const int a_member = (list_entry(a, struct thread, elem)->member;
    const int a_member = (list_entry(b, struct thread, elem)->member;
    return a_member < b_member;
}
```

It is important to note that the `elem` you pass in as the third argument of `list_entry` must match which `list` you'll be using this comparator for.

Using the Comparator

Nothing too fancy here, except to make sure you pass the address of the function in order to pass the pointer of the function:

```
//list_sort
#include <list.h>
int main(void)
{
    /* Instantiate, initialize, and populate my_list */
```

```
    //Prototype for sorting a list: void list_sort (struct list *,
       list_less_func *, void *aux);
    list_sort(&my_list, &list_less_comp, NULL);
    return 0;
}
```

Note that if the member you're comparing is a struct itself, then you'll need to write a comparator to use nested in the general comparator for the `list_elem`.

2.3 Synchronization Constructs

Pintos provides you with three synchronization constructs to use: locks, semaphores, and condition variables. You can find the header file to these synchronization constructs in `src/threads/synch.h` and their implementation in `synch.c` file in the same directory. You'll need to `#include "threads/synch.h"` in order to use these constructs in your file.

Reading the header file, you will notice that most of the synchronization functions take a pointer to themselves as an argument:

```
void lock_acquire (struct lock *);
void lock_release (struct lock *);
void sema_down (struct semaphore *);
void sema_up (struct semaphore *);
void cond_wait (struct condition *, struct lock *);
void cond_signal (struct condition *, struct lock *);
void cond_broadcast (struct condition *, struct lock *);
```

Similar to `lists`, since there are no classes, you'll have to call these function from a global scope and specify which synchronization construct you want to use these operations on.

Once you instantiate a synchronization construct, make sure you always call the `_init` function before you use any of these functions.

2.4 Interrupts

Like all operating systems, Pintos will have interrupts in which it must finish a subroutine before resuming its original work. There are some sections of code in which you do not want interrupts to happen. Consider the `sema_down` function:

```
//From src/threads/synch.c
void
sema_down (struct semaphore *sema)
{
    //intr_level stores the interrupt level
    enum intr_level old_level;

    ASSERT (sema != NULL);
```

```
    //makes sure that this currently is not an external interrupt
    ASSERT (!intr_context ());

    //saves the old interrupt level & disables interrupts at the same time
    old_level = intr_disable ();
    while (sema->value == 0)
    {
        list_push_back (&sema->waiters, &thread_current ()->elem);
        thread_block ();
    }
    sema->value--;

    //return the interrupt level back to its original state
    intr_set_level (old_level);
}
```

Note the order of process of the commented lines. These four lines allows you to disable interrupts and resume the interrupt level to what it was before.

Also keep in mind that you want to minimize the amount of time you keep interrupts disabled. There are critical interrupts that must happen for Pintos to run properly. If your project has a section of code that disables interrupts for too long, you will get very unexpected behavior.

2.5 `memset` and `memcpy`

For more information, visit cplusplus' website on `memset` and `memcpy`.

If you want to write data to the destination of a pointer, you'll want to use memset and memcpy. These two functions come in handy for part 1 of project 2.

2.5.1 `memset`

Say you want to write a single piece information into the destination of a pointer. You will want to use `memset`. For example, let's say we want to write the character `a`, you will do the following:

```
#include <stdio.h>
int main(void)
{
    char my_string[] = "XSCI350";
    printf("Original:␣%s\n", my_string);
    //ptr points to the start of the string
    char* ptr = my_string;

    //memset(void pointer to data to modify, write data, size of data in bytes
       )
    memset(static_cast<void*>(ptr), 'C', sizeof(char));

    printf("After:␣␣␣␣%s\n", my_string);
    return 0;
```

```
}
```

Output:

```
Original: XSCI350
After:    CSCI350
```

2.5.2 memcpy

Instead of writing a single piece of data, you want to write a string of data. You will want to use `memcpy`. For example, let's say we want to write the string `CSCI350`, you will do the following:

```
#include <stdio.h>
int main(void)
{
    char my_string[] = "ABCD123";
    printf("Original: %s\n", my_string);
    //ptr points to the start of the string
    char* ptr = my_string;

    //memcpy(void pointer to data to modify, write data, size of data in bytes
       )
    memset(static_cast<void*>(ptr), 'CSCI350', sizeof(char) * 7);

    printf("After:    %s\n", my_string);
    return 0;
}
```

Output:

```
Original: ABCD123
After:    CSCI350
```

2.6 Hex Dump

Hex dumps is immensely useful for part 1 of project 2; it allows you to print out the addresses and the contents of the addresses of a specified stack. The prototype of using a hex dump is:

```
static void hex_dump((uintptr_t)**, void**, int, bool);
```

The following is an example of a hex dump:

```
bfffffd0                                        00 00 00 00 |            ....|
bfffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
bfffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 00 00 62 69 |..............bi|
bffffff0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
```

It looks very daunting, so let's break it down shall we?

Firstly, you'll want to know how to call hex dump in Pintos. You'll call hex dump from the `setup_stack` function in `src/userprog/process.c`. Note the declaration of `setup_stack`[3]:

```
static bool setup_stack (void **esp);
```

The `void** esp` is the stack pointer. This is a double pointer because you will be doing pointer manipulation, and since you want these modifications to be global and not just within this function's scope, you are given a pointer to the stack pointer (pass by pointer for a pointer). Meaning to write things to the stack you will want to dereference `void** esp`.

Initially, `void** esp` is initialized to `PHYS_BASE`, which is basically the bottom of the stack (`0xbfffffff`).

```
*esp = PHYS_BASE;
```

From here you are free to start writing to the stack.

Note that because you are writing to a stack, you will be writing everything backwards. Say you want to write the word "`CSCI350`". You will write in the order of "`053ICSC`". Let's say we write this to the stack:

```
static bool setup_stack(void** esp)
{
    //Method 1
    //Notice that you have to manually move the esp pointer every time before
       you write to the stack (move pointer then write).
    *esp -= sizeof(char);
    memset(*esp, '\0', sizeof(char));
    *esp -= sizeof(char);
    memset(*esp, '0', sizeof(char));
    *esp -= sizeof(char);
    memset(*esp, '5', sizeof(char));
    *esp -= sizeof(char);
    memset(*esp, '3', sizeof(char));
    *esp -= sizeof(char);
    memset(*esp, 'I', sizeof(char));
    *esp -= sizeof(char);
    memset(*esp, 'C', sizeof(char));
    *esp -= sizeof(char);
    memset(*esp, 'S', sizeof(char));
    *esp -= sizeof(char);
}
```

```
static bool setup_stack(void** esp)
{
    //Method 2
    //Using memcpy to copy an array of data into the stack. Make sure you move
        the stack pointer the appropriate amount of bytes.
    char my_string[8] = "CSCI350\0";
    *esp -= sizeof(char) * 8;
    memcpy(*esp, my_string, sizeof(char) * 8);
}
```

[3]You are free to change function declarations and pass in any other member data you see fit

And now if we call hex dump:

```
static bool setup_stack(void** esp)
{
    //Write CSCI350\0 to the stack
    //The boolean argument toggles whether you want to show the contents of
       the stack or not. I highly suggest you set it to true to double check
       if the contents are correct.
    hex_dump((uintptr_t)*esp, *esp, sizeof(char) * 8, true);
}
```

Output of the hex dump:

```
bffffff0  00 00 00 00 00 00 00 00-43 53 43 49 33 35 30 00 |........CSCI350.|
```

The right side shows the contents of the stack (this will not show if you pass in a false for the fourth argument of hex dump). You're able to see the characters you wrote. Note that this will only show readable characters, numbers and letters; if it's not a readable character, then it will only show up as a period.

On the left side you can see the actual hex values. You can double check that the pointers you want to write are correct.

Once you have finished the code for setup stack, you'll want to double check your work.

2.7 Hash Tables

The Stanford website has documentation on how to use hash tables in Pintos. It goes without saying that Stanford's documentation is more comprehensive than mine.

The hash file can be found in `src/lib/kernel/hash.h`. Much like `list.h`, you can find all the functions in that file. As you will see, the hash is a struct; thus you can define a hash table in the following manner:

```
#include <hash.h>
struct thread
{
    /* other members */
    struct hash hash_table;

    unsigned magic;
}
```

2.7.1 Initializing the Hash Table

Hash tables are especially useful in project 3 to keep track of pages. Since every single page is going to be unique, and the virtual addresses each page is identified with is conveniently very byte-hashable, a hash table is a fantastic choice as the data structure to store pages.

Now unlike other data structures you've used in Pintos, the hash table can only be initialize if and only if the thread containing that hash table is **running**. If you try to initialize the hash table when the thread is not running, you will get the following kernel panic:

```
Kernel panic in run: PANIC at ../../threads/thread.c:350 in thread_current():
    assertion 't->status == THREAD_RUNNING' failed.
```

This means where you usually initialize your data structures, in `init_thread`, will not work because `init_thread` is being run by the parent thread, not the newly created thread. A good place to initialize the hash table is in the `start_process` function in `process.c`, since this is the very first function every single new thread will run.

The initialization function of the hash table requires four parameters:

```
bool hash_init (struct hash *, hash_hash_func *, hash_less_func *, void *aux);
```

The first parameter is a pointer to the hash struct you wish to initialize. The second is a pointer to the hash function used for the hash table, the third is a pointer to the less-than comparator function for the hash function, and the last is just auxiliary data. An example of a hash table storing pages is as follows:

```
#include <hash.h>
struct thread
{
    /* other members */
    struct hash hash_table;     /* hash table to store pages */
    unsigned magic;             /* Detects stack overflow */
}

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

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

/* 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->vaddr < b->vaddr;
}
```

```
/* First function every new thread runs */
static void start_process(void* file_name_)
{
    (...)
    // initializes the hash table
    hash_init(&thread_current()->spage_table, page_hash, page_less, NULL);
    (...)
}
```

2.7.2 Using the Hash Table

Once initialized, you can insert elements and remove elements as you would expect:

```
int main()
{
    struct page* p = malloc(sizeof(struct page));
    hash_insert (&pages, &p->hash_elem);   /* inserting */
    hash_delete (&pages, &p->hash_elem);   /* removing */

    return 0;
}
```

Iterating through a hash table can be a bit tricky, but fortunately the Stanford website provides an example:

```
/* 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;
}
```

2.7.3 Freeing Memory in a Hash Table

When you want to destroy your hash table, if the entries of the hash table has dynamically allocated memory, you will need to free these memory. You will notice that the `hash_clear` and `hash_destroy` function takes in a function pointer:

```
void hash_clear (struct hash *, hash_action_func *);
void hash_destroy (struct hash *, hash_action_func *);
```

The `hash_action_func` is a function that frees memory from each hash table entry. An example is as follows:

```
struct page
{
     struct my_struct* my_ptr;    /* dynamically allocated object */
     struct hash_elem hash_elem
}


void
page_free (struct hash_elem* elem, void* aux UNUSED)
{
    struct page *entry = hash_entry(elem, struct page, hash_elem);

    //free the memory
    free(entry->my_ptr);
}
```

2.8 Bitmaps

Bitmaps are essentially an array of booleans. This is very useful in project 3 to keep track of which sectors on disk, organized contiguously from sector 0 to sector n, are free and which sectors are not. By iterating through the bits and finding the first free bit, it allows for an easy "first fit" policy for cache eviction (due to every page being 4KB large in Pintos).

2.8.1 Initializing a Bitmap

Unlike lists, when you initialize a bitmap you'll need to indicate how many bits you wish to have for your bitmap. In the context of swap and sectors, an easy way to do so is as follows:

```
#include <bitmap.h>
struct bitmap* bitmap;
struct block* block;

int main()
{
    /* Initializes the block for swapping */
    block = block_get_role(BLOCK_SWAP);

    /* Initializes the bitmap with the appropriate number of bits */
    bitmap = bitmap_create(block_size(block));

    return 0;
}
```

Upon creation, all bits in the bitmap are set to 0, or false.

2.8.2 Using a Bitmap

Since one sector is 512 bytes, and one page is 4KB, it means it takes 8 sectors to store one page. In the bitmap, we want to find 8 consecutive bits, corresponding to 8 consecutive sectors in disk, to use as the index to the disk. The following functions will come in handy to do so:

```
/* sets the boolean at index <idx> to the value <bool> */
void bitmap_set (struct bitmap *, size_t idx, bool);

/* sets the boolean at index <idx> to the true */
void bitmap_mark (struct bitmap *, size_t idx);

/* sets the boolean at index <idx> to the false */
void bitmap_reset (struct bitmap *, size_t idx);

/* beginning from index <start>, finds the first instance of <cnt> consecutive
    <bool> bits and returns the index */
size_t bitmap_scan (const struct bitmap *, size_t start, size_t cnt, bool);

/* same as bitmap_scan, but in addition also flips the bits */
size_t bitmap_scan_and_flip (struct bitmap *, size_t start, size_t cnt, bool);
```

Note that bitmaps does not have elems or any structs associated with each entry.

2.8.3 Destroying a Bitmap

Since a bitmap is only an array of bits, there can be no dynamically allocated memory associated with each entry of the bitmap. Therefore destroying a bitmap consists of simply calling the destroy function:

```
void bitmap_destroy (struct bitmap *);
```

3 Project 1: Threads

Since this will be your first Pintos Project, this section will begin with showing you how to use some essential aspects in Pintos.

You can find the official documentation for project 1 on Stanford's website.

Remember that you should also read Stanford's official documentation before you start writing the code on it.

3.1 The Thread Struct

You can find the definition of `struct thread` in `src/threads/thread.h`:

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list.
       */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
};
```

The members that are defined by default are pretty easy for you to read around the header file to figure out. For project 1, feel free to add more members to this class as you see fit. Make sure that you **do not** add members below the declaration of `unsigned magic` as this will alter the placement of the member in the struct on the stack and give you undefined behavior.

It is important to know that the kernel of Pintos effectively is run as a thread. Meaning the kernel is subject to all the functions you find in `src/threads/thread.h`.

Also note that Pintos is a single-processor operating system, meaning one and only one thread can be run by the processor at any given point in time.

3.1.1 Noteworthy Functions

Here are a list of functions to pay close attention to in `src/threads/thread.h`

Function	Usage
`thread_init`	Called once and only once to initialize global constructs all threads use such as locks and semaphores.
`thread_create`	Spawns a new thread and places it in the *ready* state. This new thread will run the function passed in by the argument `thread_func`.
`thread_block`	Blocks the current running thread.
`thread_unblock`	Takes the thread passed in the argument and unblocks it.
`thread_current`	Returns a pointer to the current running thread.
`thread_foreach`	Takes the function passed in the argument and runs it with every thread.
`thread_get_priority`	Returns the priority of the currently running thread.
`thread_set_priority`	Takes the int passed in the argument and sets it as the priority of the current running thread.

3.1.2 Thread Scheduler

You will notice that there are no scheduling functions at all in `thread.h`. Scheduling should be done automatically and not be manually invoked outside of `thread.c`.

If you look at `thread.c`, you will see a `schedule` function.

```
static void
schedule (void)
{
    struct thread *cur = running_thread ();
    struct thread *next = next_thread_to_run ();
    struct thread *prev = NULL;

    ASSERT (intr_get_level () == INTR_OFF);
    ASSERT (cur->status != THREAD_RUNNING);
    ASSERT (is_thread (next));

    if (cur != next)
        prev = switch_threads (cur, next);
    thread_schedule_tail (prev);
}
```

If you trace this function, you will find that this `schedule` function is called on the last line of `thread_block`, `thread_exit`, and `thread_yield`; this ensures that whenever a thread is taken off the processor, the scheduler is run and another thread (not necessarily a different thread than the one just taken off the processor) will be placed on the processor to run.

Note that the actual "scheduling", or in other words the deciding of which thread to run next, is done in the `next_thread_to_run` function. The `schedule` runs this function as well as takes care of tying up loose ends in the thread switching process.

3.2 Part 1: Alarm

3.2.1 Pintos Timer

The timing functionality of Pintos can be found in `src/devices/timer.h`. You should take note of the following two functions in particular:

```
int64_t timer_interrupt (struct intr_frame *args UNUSED);
int64_t timer_elapsed (int64_t);
```

`timer_interrupt` is effectively the clock of Pintos. Pintos will only progress in clock ticks when `timer_interrupt` is called.

`timer_elapsed` provides you with the number of ticks that has passed since Pintos started running. This will be very helpful in this part of the project.

3.2.2 The `timer_sleep` Function

Pintos provides a `timer_sleep` function for the current thread to be pulled off the processor for the given number of ticks. If we take a look at the implementation of `timer_sleep`:

```
void
timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();

    ASSERT (intr_get_level () == INTR_ON);
    while (timer_elapsed (start) < ticks)
        thread_yield ();
}
```

You will notice that all the default implementation does is to check if the current thread is supposed to be sleeping; if so, yield this thread and schedule another one.

Since `thread_yield` puts the current running thread into the *ready* state, this thread scheduled back onto the processor immediately. There is nothing guaranteeing sleeping threads from being scheduled onto the processor before their waking time; thus wasting resources by having the processor constantly switching in and out sleeping threads.

Now instead of using `thread_yield`, you'll want to use `thread_block`. Blocking a thread will effectively banish it from the *ready* state until someone calls `thread_unblock` on that thread:

```
void
timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();

    ASSERT (intr_get_level () == INTR_ON);
    /* No more busy-waiting
    while (timer_elapsed (start) < ticks)
```

```
        thread_yield ();
    */
    //Use some kind of thread_block() functionality instead
}
```

You'll need to properly use `thread_block` and `thread_unblock` in order to implement `timer_sleep` properly.

3.3 Part 2: Priority Donation

3.3.1 Overview

For this part of the project, you will need to implement a donation functionality for threads to donate their priority to the threads they are waiting on. Note that you do not want to donate if the target thread has a higher priority than what you want to donate.

Thread	**Priority**	**Lock Remarks**
Case 1: Donation		
Thread 1	10	Holding *Lock A*
Thread 2	100	Waiting for *Lock A*
Case 2: No Donation		
Thread 1	100	Holding *Lock A*
Thread 2	10	Waiting for *Lock A*

In addition to priority donation, you want to modify the thread scheduler to always run the highest priority thread first. Refer to section 3.1.2 for more information on the scheduler.

Below describes a subset of the trickier donation tests you will need to pass.

3.3.2 Multiple Donation

In this scenario, there are multiple threads waiting on the same lock held by another thread. The thread holding the lock must have the priority of the highest donation unless all donated priorities are lower than its original priority.

Thread	**Priority**	**Lock Remarks**
Thread 1	10	Holding *Lock A*
Thread 2	100	Waiting for *Lock A*
Thread 3	50	Waiting for *Lock A*
Donation Result		
Thread 1	**100**	Holding *Lock A*
Thread 2	100	Waiting for *Lock A*
Thread 3	50	Waiting for *Lock A*

Once *Thread 1* release *Lock A*, it must relinquish the priorities donated from *Thread 2* and *Thread 3*. *Thread 2* acquires *Lock A* next due to its higher priority compared to *Thread 3*:

Thread	**Priority**	**Lock Remarks**
Thread 1	**10**	None
Thread 2	100	Holding *Lock A*
Thread 3	50	Waiting for *Lock A*
No Donation Occurs		

3.3.3 Nested Donation

In this test case, a thread's donation must be donated to not only its recipient, but also the thread its recipient is waiting on.

Thread	**Priority**	**Lock Remarks**
Thread 1	10	Holding *Lock A*
Thread 2	50	Holding *Lock B* Waiting for *Lock A*
Thread 3	100	Waiting for *Lock B*
Donation Results		
Thread 1	**100**	Holding *Lock A*
Thread 2	**100**	Holding *Lock B* Waiting for *Lock A*
Thread 3	100	Waiting for *Lock B*

3.3.4 Donation Chain

This is similar to *nested donation*, however the donation must propagate through an arbitrary number of nested donations. The Pintos test case `pintos -v -k -T 60 -bochs - -q run priority-donate-chain` will be testing for a depth layer of seven.

3.3.5 Design Considerations

When designing priority donation, you will want to consider the following questions:

- Should a thread keep track of its received donations?
- Should a thread keep track of which other threads it has donated to?
- Should a thread change the value of its `priority` data member?
- Should a thread keep track of a secondary priority data member?
- How should a thread update its priority?
- How should a thread keep track of multiple locks?

- When do I need to use synchronization constructs to prevent race conditions between donations?
- What processes should happen when I call `thread_set_priority`?
- What processes should happen when I call `thread_get_priority`?

3.4 Part3: Advanced Scheduler

In this part, you need to implement a multilevel feedback queue scheduler. To finish this part, you need to refer to the corresponding content in Stanford's official documentation: 2.2.4 Advanced Scheduler and B. 4.4BSD Scheduler.

3.5 Design Document

Remember to complete the design document `doc/threads.tmpl` and save it as `src/threads/DESIGNDOC`. The following are a subset of the questions in the design document for your convenience. Remember, these questions might give you hints and tips on how to approach the project.

- Briefly describe what happens in a call to `timer_sleep`, including the effects of the timer interrupt handler.
- What steps are taken to minimize the amount of time spent in the timer interrupt handler?
- How are race conditions avoided when multiple threads call `timer_sleep` simultaneously?
- How are race conditions avoided when a timer interrupt occurs during a call to `timer_sleep`?
- How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first?
- Describe the sequence of events when a call to `lock_acquire` causes a priority donation. How is nested donation handled?
- Describe the sequence of events when `lock_release` is called on a lock that a higher-priority thread is waiting for.
- Describe a potential race in `thread_set_priority` and explain how your implementation avoids it. Can you use a lock to avoid this race?

4 Project 2: User Programs

You can find the official documentation for project 2 on Stanford's website

The list of tests to pass for project 2 can be found in here.

If you're wondering why you aren't passing any tests, you can find the section about how to start passing tests here.

4.1 Project Setup

Before you start this project, please make sure that you have submitted your code and design document of project 1 to the repo of project 1 on Github Classroom.

4.1.1 Keep Alarm Code

You will need alarm working for project 2, therefore you will need to setup using one of the two following ways:

1. Make sure that you have submitted the project 1 to the GitHub repo. Then directly remove all the priority donation code from project 1 on your own computer(you can ignore the code about advanced scheduler because we will not run Pintos using `-mlfqs` in the future).
2. You need to get the initial version of Pintos code first. You can find it on your own virtual machine through the path `/home/csci3150/initial-version/pintos-base/` or you can download it from the GitHub by execute the command `git clone https://github.com/cuhk-estr3102/pintos-base.git`. Copy & paste your project 1 code for alarm into `pintos-base` directory. Afterwards replace the `src` directory under `/home/csci3150/os-pintos/pintos/` with the new `src` directory under `pintos-base`.

Once you have completed the above step, run `make check` from `src/threads/build` to see if you pass all the alarm tests (except alarm-priority since that is a priority-donation test).

4.1.2 Modify Kernel Path Information

Because you will begin to work under the directory `/home/csci3150/os-pintos/pintos/src/userprog/` for this new project, you need to update some path information in Pintos:

(1) Firstly, in the line 259 of the file `/home/csci3150/os-pintos/pintos/src/utils/pintos`, replace the

```
my $name = find_file('/home/csci3150/os-pintos/pintos/src/threads/build/kernel
    .bin');
```

with

```
my $name = find_file('/home/csci3150/os-pintos/pintos/src/userprog/build/
    kernel.bin');
```

(2) Secondly, in the line 362 of the file "/home/csci3150/os-pintos/pintos/src/utils/Pintos.pm", replace the

```
$name = find_file('/home/csci3150/os-pintos/pintos/src/threads/build/kernel.
    bin') if !defined $name ;
```

with

```
$name = find_file('/home/csci3150/os-pintos/pintos/src/userprog/build/kernel.
    bin') if !defined $name ;
```

(3) At last, run `make` command under the `userprog` directory to recompile the Pintos kernel. (Warning: run `make clean` first if you compiled the kernel in `userprog` directory before)

4.2 The Process File

In Pintos, every user program is run by a process. In a modern operating system, a single thread can run multiple processes; however in Pintos, every thread will only run **one and only one** process.

If you look in `src/userprog/process.h`, you will see a small handful of functions. These are the functions for user programs. The following is a short and incomplete description of each:

Function	**Description**
`process_execute`	Executes the user program from the designated file in the argument
`process_wait`	Waits for the child process with designated tid to finish before continuing execution
`process_exit`	Terminates user program currently running
`process_activate`	Sets up CPU to run user program in current thread

You will be working heavily in this file for both project 2 and 3.

4.3 Part 1: Setup Stack

In this part, you will take in a filename for a user program(a command line argument), parse it, and setup the stack for this user program.

You will be mainly working in `src/userprog/process.c`.

4.3.1 Where a User Program Starts

If you read the description for `process_execute`, you will see that this is the function responsible to start a user program. Also, the argument passed into this function is a filename.

This filename is "raw" in the sense that it contains both the executable name and the arguments, for example:

```
user_program arg1 arg2 arg3 arg4
```

Which means when a thread is created with `thread_create` in this function to run the user program, you will notice that the thread is named the raw filename:

```
tid = thread_create(file_name, PRI_DEFAULT, start_process, fn_copy);
```

You do not want the thread to have the raw filename. Instead you want the thread's name to be the executable name. You will need to extract the executable name from `file_name` and pass that in instead.

```
tid = thread_create{exec_name, PRI_DEFAULT, start_process, fn_copy);
```

Also notice `fn_copy`. This is a copy of the raw filename and passed in as an auxiliary parameter. This will come in handy. The function this thread will run is `start_process`, which takes in an argument `void *file_name_`. `fn_copy` is passed in as this argument, allowing you access to a copy of the full raw filename in this function. This will come in handy.

If you look at the `start_process` function, you will see a `load` function; this function is where the user program gets loaded with all its data. In this `load` function, Pintos will try to load the executable (a file) with `filesys_open(file_name)`. Once again, this filename should not be the raw filename but instead just the executable name. You will decide when to extract the executable name and pass in the correct string.

In the `load` function you will also find a function called `setup_stack`. This is the function in which you will setup the stack for each user program.

4.3.2 Emulate `process_wait()`

Looking at the file `process.c`, when a new process is created in `process_execute`, you will notice that the function `thread_create` is created. When a child process is created, the parent process will wait for this process to finish loading properly, then proceed to continue with its execution.

If you take a look at `process_wait()`, you will notice that right now all this function does is return -1.

```
int
process_wait(tid_t child_tid)
{
    return -1;
}
```

This means when a parent process waits, it's going to return immediately. This is a problem because with the default code, when a parent process spawns a child process (in the test cases, when the test case process spawns child processes to test), the parent will not wait at all for the child to finish loaded, terminating before the child process gets a chance to do anything. This becomes especially problematic when the main test process spawns a child process to test, and instead exits immediately, finishing the test before anything can happen.

What you want to do first is to emulate a simple process wait function. The easiest way is to simply continually yielding:

```
int
process_wait(tid_t child_tid)
{
    while(true)
    {
        thread_yield();
    }
}
```

This will allow the child process to finish all its setup, allowing you to check the stack setup of the child process; this is a good quick start for you to start setup stack implementation and debug your stack setup with hex dump or print statements. The problem here is that the parent thread will be forever stuck in the ready state, never finishing the test.

The second quick fix is to use a semaphore with an initial acquire of 0:

```
int
process_wait(tid_t child_tid)
{
    //this semaphore is initialized to an acquire of 0
    //allowing the parent to become immediately block upon waiting
    sema_down(&thread_current()->some_semaphore);
}
```

Which means when a child process finishes its execution, you'll want to up this semaphore

```
void
thread_exit(void)
{
    sema_up(&thread_current()->parent->some_semaphore);
}
```

This way you have emulated a very simple process wait functionality, allowing you to get started with setup stack.

IMPORTANT: Note that this is only a very quick and simple implementation to get started with setup stack. You WILL be modifying these implementations. It is highly suggested you move where you up and down the semaphores to better positions in the code. For example once you start implementing syscalls, it will be a much better idea to up and down the semaphore in the syscalls rather than the process functions.

4.3.3 Setup Stack

First notice that `setup_stack` only takes in a `void** esp` (the stack pointer). Feel free to add more arguments and pass in any other information you want.

Don't modify what's already there, you'll need to add on to the function before you return success. Remember `void** esp` will be initialized with `PHYS_BASE`, which is the top of the stack. You'll need to incrementally move the pointer and write data to the pointer. Also remember that since it's a stack, you'll want to write everything in reverse order, and you want to decrement the pointer instead.

Here are the general steps to setting up the stack. The code snippets will not actually work, but they are to give you a gist of how it's supposed to work.

1. Parse the filename deliminating by white spaces. Notice that `setup_stack` doesn't have access to the filename. You'll need to find a way to pass that information into this function. Parsing the filename can easily be done with `strtok_r`.
2. Write each argument (including the executable name) in reverse order, as well as in reverse for each string, to the stack. Remember to write a `\0` for each argument. `memcpy` will come in handy here.

```
char argument[] = "arg1\0"
*esp -= strlen(argument);
memcpy(*esp, argument, strlen(argument));
```

3. Write the necessary number of `0`s to word-align to 4 bytes. A little modulus math and `memset` will get the job done.

```
int word_align = 0, 1, 2, or 3
*esp -= word_align;
memset(*esp, 0, word_align);
```

4. Write the last argument, consisting of four bytes of `0`'s.
5. Write the addresses pointing to each of the arguments. You'll need to figure out how to reference the addresses after writing all the arguments. These are `char*`s.

```
*esp -= sizeof(char*);
memcpy(*esp, address, sizeof(char*));
```

6. Write the address of `argv[0]`. This will be a `char**`.

```
*esp -= sizeof(char**);
memcpy(*esp, address of argv[0], sizeof(char**));
```

7. Write the number of arguments (`argc`). Make sure that this spans over 4 bytes.
8. Write a NULL pointer as the return address. This will be a `void*`.

It will be a good idea to use `hex_dump` to check the correctness of your stack after each and every step in order to avoid a debugging mess.

Here are the first three `hex_dump` results for you to cross check.

Hex Dump for args-none

```
bffffe0  00 00 00 00 01 00 00 00-ec ff ff bf f6 ff ff bf |................|
bfffff0  00 00 00 00 00 00 61 72-67 73 2d 6e 6f 6e 65 00 |......args-none.|
```

Hex Dump for args-single

```
bffffd0              00 00 00 00-02 00 00 00 e0 ff ff bf |    ............|
bffffe0  ed ff ff bf f9 ff ff bf-00 00 00 00 00 61 72 67 |.............arg|
bfffff0  73 2d 73 69 6e 67 6c 65-00 6f 6e 65 61 72 67 00 |s-single.onearg.|
```

Hex Dump for args-multiple

```
bffffb0              00 00 00 00-05 00 00 00 c0 ff ff bf |    ............|
bffffc0  da ff ff bf e8 ff ff bf-ed ff ff bf f7 ff ff bf |................|
bffffd0  fb ff ff bf 00 00 00 00-00 00 61 72 67 73 2d 6d |..........args-m|
bffffe0  75 6c 74 69 70 6c 65 00-73 6f 6d 65 00 61 72 67 |ultiple.some.arg|
bfffff0  75 6d 65 6e 74 73 00 66-6f 72 00 79 6f 75 21 00 |uments.for.you!.|
```

4.4 Syscall Handler

Part 2 of the project will require you to implement the support for user programs to request system call functions (kernel functions). You will mainly be working in `src/userprog/syscall.c` and `src/userprog/process.c`. This section is written with reference to Colin Cammarano's guide from Spring of 2016.

4.4.1 What is a System Call?

A system call, or syscall, is a function that allows a user program to perform a kernel level task. System calls include tasks such as file I/O, opening a user program, exiting a user program, and waiting for a user program. Since the behaviors defined here change the status of the OS itself, the kernel (the core of the OS) should provide exclusive access to these functions. By restricting system calls to a small syscall interface, the OS can provide programs with great flexibility while also keeping the kernel secure. Something to keep in mind here is that only user processes and threads can use the syscall interface–kernel threads can (and in Pintos, must) directly call the underlying kernel code for each syscall.

4.4.2 The Syscall Process

When a user program calls one of the functions defined in lib/user/syscall.h, it causes a software interrupt and creates an interrupt frame. This suspends the currently running thread. The syscall function that the user program calls takes anywhere between 1 and 3 parameters. The pointers to each parameter are pushed onto the stack from the end of the beginning, then an integer value (the syscall code) is pushed last. This frame is then dispatched to the `void`

`syscall_handler(struct intr_frame* f);` function. This function should look at the first element of the frame's stack pointer (`f->esp`), and determine the type of syscall to execute. You can get the syscall code by doing the following:

```
int sys_code = *(int*)f->esp;
```

From here, you should write functions that actually execute each system call at the kernel level. The list of syscalls you need to implement can be found on Stanford's Website. You will need to parse the data in `f->esp` and send it to these functions. When the `syscall_handler` finishes, the current thread resumes.

4.4.3 Layout, Parsing, and Validation

Layout

Now, let's talk about how `f->esp` is laid out. Conveniently, the data in `f->esp` is stored as a uint and a series of pointers to data stored in memory. For example, here is an example of `f->esp` with three arguments below:

```
f->esp   ----------------------------
        |           SYS_CODE         |
         ----------------------------
        |  arg[0] ptr -OR- int_value |
         ----------------------------
        |  arg[1] ptr -OR- int_value |
         ----------------------------
        |  arg[2] ptr -OR- int_value |
         ----------------------------
```

Notice that the `SYS_CODE` is preset in Pintos. You can find the full list of sycall codes in `src/lib/syscall-nr.h`.

```
/* System call numbers. */
enum
{
    /* Projects 2 and later. */
    SYS_HALT,                   /* Halt the operating system. */
    SYS_EXIT,                   /* Terminate this process. */
    SYS_EXEC,                   /* Start another process. */
    SYS_WAIT,                   /* Wait for a child process to die. */
    SYS_CREATE,                 /* Create a file. */
    SYS_REMOVE,                 /* Delete a file. */
    SYS_OPEN,                   /* Open a file. */
    SYS_FILESIZE,               /* Obtain a file's size. */
    SYS_READ,                   /* Read from a file. */
    SYS_WRITE,                  /* Write to a file. */
    SYS_SEEK,                   /* Change position in a file. */
    SYS_TELL,                   /* Report current position in a file. */
    SYS_CLOSE,                  /* Close a file. */
};
```

Parsing

Because the first element is an integer, and each of the following arguments are either pointers or integers, we can move through the stack in 4 byte intervals. Regardless of whether the value is an integer or pointer, we can store it as a pointer (`void*`). When we want to parse these elements, then cast it back to an integer if needed. We can do this because in C (and by extension, low level C++), pointers are merely integer values that represent memory addresses. Thus, the conversion from `void*` to `int` via `int(ptr)` is valid. Remember that syscalls can take either 1, 2, or 3 arguments.

As a quick example, parsing the `SYS_CODE` will look like the following:

```
static void
syscasll_handler(struct intr_frame* f)
{
    //first check if f->esp is a valid pointer)
    if (f->esp is a bad pointer)
    {
        exit(-1);
    }

    //cast f->esp into an int*, then dereference it for the SYS_CODE
    switch(*(int*)f->esp)
    {
        case SYS_HALT:
        {
            //Implement syscall HALT
            break;
        }
        case SYS_EXIT:
        {
            //Implement syscall EXIT
            break;
        }
        (...)
    }
}
```

After parsing the `SYS_CODE`, you will want to parse out the argument for the corresponding syscall. For example, the write syscall will have three arguments:

```
int write(int fd, const void* buffer, unsigned size);
```

And so extracting these three arguments will look like the following:

```
static void
syscall_handler(struct intr_frame* f)
{
    switch(*(int*)f->esp)
    {
        case SYS_WRITE:
        {
            int fd = *((int*)f->esp + 1);
            void* buffer = (void*)(*((int*)f->esp + 2));
            unsigned size = *((unsigned*)f->esp + 3);
```

```
            //run the syscall, a function of your own making
            //since this syscall returns a value, the return value should be
               stored in f->eax
            f->eax = write(fd, buffer, size);
        }
    }
}
```

You might notice that parsing the `void* buffer` is really odd. If we break it down into steps:

1. First, `f->esp` can only be incremented with +1, +2, +3 if and only if it is of an `int*` type. This is because +1, +2, +3 will only have the correct integer math when we perform these additions on an integer pointer.

   ```
   (*int)f->esp + 2;
   ```

2. This integer pointer is now pointing to the contents of the actual buffer we need. If we directly cast this `int*` into a `void*`, you will be getting the address of the buffer, not the buffer itself, therefor we need to dereference the `int*` in order to get the contents, then cast it into a `void*`

   ```
   int* ptr = (int*)f->esp + 2;
   void* buffer = (void*)*ptr;
   ```

3. The reason for all the parenthesis on that one line is to make sure that my + operations and dereferencing operations are being done in the correct order.

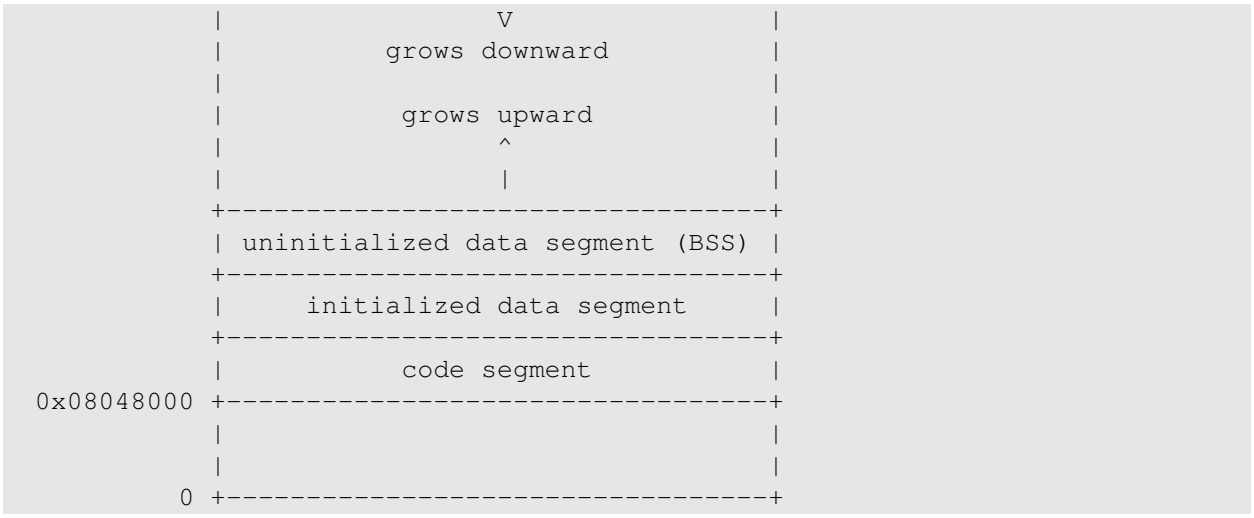
Validation

After parsing the stack and acquiring the arguments you need, you will want validate the pointers. You can do this either in the `syscall_handler` or in the actual syscall functions; it's up to you. There are two main ways to approach validating the addresses:

1. The first method is to verify the validity of a user-provided pointer, then dereference it. If you choose this route, you'll want to look at the functions in `userprog/pagedir.c` and in `threads/vaddr.h`. This is the simplest way to handle user memory access.
2. The second method is to check only that a user pointer points below `PHYS_BASE`, then dereference it. An invalid user pointer will cause a "page fault" that you can handle by modifying the code for `page_fault()` in `userprog/exception.c`. This technique is normally faster because it takes advantage of the processor's MMU, so it tends to be used in real kernels (including Linux).

What it means by a "valid" pointer is that the pointer requested by the user program, in other words the pointer you get when you extract it from the interrupt frame, is within the correct user program's memory space. In Pintos, a typical user virtual memory is laid out in the following fashion:

```
   PHYS_BASE +----------------------------------+
             |            user stack            |
             |                 |                |
```



The code segment of Pintos (where the user program code is stored) starts at the virtual address `0x08048000`, approximately 128MB from the bottom of the address space. This is number is chosen arbitrarily.

For project 2, the stack is fixed in size, therefore the user program should not be able to cannot access any memory that is less than `0x08048000` or greater than `PHYS_BASE`.

4.4.4 System Call Implementation

There are 13 system calls for you to implement. All 13 system calls and their descriptions (found on the Stanford Website) are replicated below:

1. `void halt(void)`
   Terminates Pintos by calling `shutdown_power_off()` (declared in `threads/init.h`). This should be seldom used, because you lose some information about possible deadlock situations, etc.
2. `void exit(int status)`
   Terminates the current user program, returning status to the kernel. If the process's parent waits for it (see below), this is the status that will be returned. Conventionally, a status of 0 indicates success and nonzero values indicate errors.
3. `pid_t exec(const char* cmd_line)`
   Runs the executable whose name is given in `cmd_line`, passing any given arguments, and returns the new process's program ID (`pid`). Must return `pid -1`, which otherwise should not be a valid `pid`, if the program cannot load or run for any reason. Thus, the parent process cannot return from the exec until it knows whether the child process successfully loaded its executable. You must use appropriate synchronization to ensure this.
4. `int wait(pid_t pid)`
   Waits for a child process `pid` and retrieves the child's exit status.

   If `pid` is still alive, waits until it terminates. Then, returns the status that `pid` passed to exit. If `pid` did not call `exit()`, but was terminated by the kernel (e.g. killed due to an exception),

`wait(pid)` must return `-1`. It is perfectly legal for a parent process to wait for child processes that have already terminated by the time the parent calls wait, but the kernel must still allow the parent to retrieve its child's exit status, or learn that the child was terminated by the kernel.

`wait` must fail and return `-1` immediately if any of the following conditions is true:

- pid does not refer to a direct child of the calling process. pid is a direct child of the calling process if and only if the calling process received pid as a return value from a successful call to exec. Note that children are not inherited: if A spawns child B and B spawns child process C, then A cannot wait for C, even if B is dead. A call to wait(C) by process A must fail. Similarly, orphaned processes are not assigned to a new parent if their parent process exits before they do.
- The process that calls wait has already called wait on pid. That is, a process may wait for any given child at most once.

Processes may spawn any number of children, wait for them in any order, and may even exit without having waited for some or all of their children. Your design should consider all the ways in which waits can occur. All of a process's resources, including its struct thread, must be freed whether its parent ever waits for it or not, and regardless of whether the child exits before or after its parent.

You must ensure that Pintos does not terminate until the initial process exits. The supplied Pintos code tries to do this by calling `process_wait()` (in `userprog/process.c`) from `main()` (in `threads/init.c`). We suggest that you implement `process_wait()` according to the comment at the top of the function and then implement the wait system call in terms of `process_wait()`.

Implementing this system call requires considerably more work than any of the rest.

5. `bool create(const char* file, unsigned initial_size)`
   Creates a new file called file initially `initial_size` bytes in size. Returns `true` if successful, `false` otherwise. Creating a new file does not open it: opening the new file is a separate operation which would require a open system call.

6. `bool remove(const char* file)`
   Deletes the file called `file`. Returns `true` if successful, `false` otherwise. A file may be removed regardless of whether it is open or closed, and removing an open file does not close it. See Removing an Open File, for details.

7. `int open(const char* file)`
   Opens the file called `file`. Returns a nonnegative integer handle called a "file descriptor" (`fd`), or `-1` if the file could not be opened.

   File descriptors numbered `0` and `1` are reserved for the console: `fd 0` (`STDIN_FILENO`) is standard input, `fd 1` (`STDOUT_FILENO`) is standard output. The open system call will never return either of these file descriptors, which are valid as system call arguments only as explicitly described below.

   Each process has an independent set of file descriptors. File descriptors are not inherited by child processes.

When a single file is opened more than once, whether by a single process or different processes, each open returns a new file descriptor. Different file descriptors for a single file are closed independently in separate calls to close and they do not share a file position.

8. `int filesize(int fd)`
   Returns the size, in bytes, of the file open as `fd`.

9. `int read(int fd, void* buffer, unsigned size)`
   Reads size bytes from the file open as `fd` into buffer. Returns the number of bytes actually read (0 at end of file), or `-1` if the file could not be read (due to a condition other than end of file). Fd 0 reads from the keyboard using `input_getc()`.

10. `int write(int fd, const void* buffer, unsigned size)`
    Writes size bytes from buffer to the open file `fd`. Returns the number of bytes actually written, which may be less than size if some bytes could not be written.

    Writing past end-of-file would normally extend the file, but file growth is not implemented by the basic file system. The expected behavior is to write as many bytes as possible up to end-of-file and return the actual number written, or `0` if no bytes could be written at all.

    Fd 1 writes to the console. Your code to write to the console should write all of buffer in one call to `putbuf()`, at least as long as size is not bigger than a few hundred bytes. (It is reasonable to break up larger buffers.) Otherwise, lines of text output by different processes may end up interleaved on the console, confusing both human readers and our grading scripts.

11. `void seek(int fd, unsigned position)`
    Changes the next byte to be read or written in open file `fd` to `position`, expressed in bytes from the beginning of the file. (Thus, a position of 0 is the file's start.)

    A seek past the current end of a file is not an error. A later read obtains 0 bytes, indicating end of file. A later write extends the file, filling any unwritten gap with zeros. (However, in Pintos files have a fixed length until project 4 is complete, so writes past end of file will return an error.) These semantics are implemented in the file system and do not require any special effort in system call implementation.

12. `unsigned tell(int fd)`
    Returns the position of the next byte to be read or written in open file `fd`, expressed in bytes from the beginning of the file.

13. `void close(int fd)`
    Closes file descriptor `fd`. Exiting or terminating a process implicitly closes all its open file descriptors, as if by calling this function for each one.

4.4.5 Helpful Files and Functions

File Syscalls

For syscalls dealing with files, the functions in `src/filesys/file.h` and `src/filesys/filesys.h` will be immensely helpful. All the syscalls you will need to implement that deals with files will be using at least one of the functions found in these two files. You will need to match the user program's arguments into the arguments taken for the file-related functions. For example, the pseudocode for syscall `create` will be:

```
bool create (const char* file, unsigned initial_size)
{
    check to see if valid file pointer
    using synchronization constructs:
        //bool filesys_create (const char *name, off_t initial_size);
        bool = filesys_create(file pointer, initial size);
    return bool
}
```

All file-related syscalls are rather straightforward since you can just use existing functions.

Denying Writes to Executables You must implement a safeguard in order for a user program to not change the contents of a file if that file is another executable. You may find `void file_deny_write` helpful found in `src/filesys/file.h`.

You will also want to look at the `load` function in `src/userprog/process.c`. If an executable is successfully loaded in this function, you want to proceed to deny writing to this executable. You will need to find a way to keep track of which files are executables and which files are not.

`pagedir_get_page` For the bad-ptr tests (create-bad-ptr, open-bad-ptr, read-bad-ptr, write-bad-ptr, and exec-bad-ptr), you will need to check for two conditions:

1. The pointer in the syscall handler is within `PHYS_BASE` and `0x08048000`.
2. The pointer requested belongs to a page in the virtual memory of this thread.

The second condition is a bit trickier to check. First note that in the threads struct there is a definition:

```
#ifdef USERPROG
    uint32_t* pagedir
#endif
```

This means for project 2 and onwards every thread will also have a pointer to its page directory. You will learn much more about virtual memory soon, so in simple terms this pointer basically points to the list of pages (memory) this thread has access to. All these pages are valid user program pointers, meaning within `PHYS_BASE` and 0x08048000, thus just checking for these two pointers is not sufficient.

You will need to check whether a pointer has been allocated within a thread's pages, you will need to use `pagedir_get_page`. Simply pass in the pointer to the page directory (`thread_current()->pagedir`) as well as the pointer in question, and this function will either 1) a pointer to the page the pointer passed into exists in or 2) `NULL` if this pointer has not been allocated memory for yet. You will need to check for the second case to pass some of the bad-ptr tests. A short example is as follows:

```
#include "threads/thread.h"
#include "userprog/pagedir.h"

int main(void)
{
    char* bad_ptr = 0x20101234;
```

```
    if (pagedir_get_page(thread_current()->pagedir, bad_ptr) == NULL)
    {
        printf("This␣pointer␣is␣bad");
    }
    else
    {
        printf("This␣pointer␣is␣good");
    }
    return 0;
}
```

The Wait Syscall This is by far the trickiest syscall to implement. To help you along, here are some tips and questions for you to ponder:

- Remember, in Pintos, each process is run by its own exclusive thread. I will use the term "process" and "thread" interchangeably for this reason.
- Should every thread keep track of which thread is its parent? What if there isn't a parent?
- Should every thread keep track of which other threads are its child threads?
- If a thread is going to wait on its child thread, how can I ensure this parent thread does NOT get scheduled onto the processor?
- If a thread is going to wait on its child thread, and the child thread exists and is running, when should I resume the parent thread? How can I communicate the child thread's exit status to the parent thread?
- If a thread is going to wait on its child thread, and the child thread doesn't exist:
  - This child thread has never existed before, therefore return -1
  - This child thread has existed before and has already exited. How can I access the exist status of this already-exited child thread?
- How can I create / remove child threads from a parent thread?

The Exit Syscall For an odd reason, Pintos requires a very specific print statement for the exit syscall. When a process exits, you need to print the following with `printf`:

```
<thread_current()->name>: exit(<exit status>)
```

For example:

```
Main Thread: exit(1)
```

4.5 How to Start Passing Tests

There are five things you need to do in order to start passing tests:

- Pass in the executable name instead of the raw filename for `thread_create` and `filesys_open`.

- Setup the stack properly
- Implement a simple form of `process_wait`
- Implement the write syscall for `STDOUT_FILENO` with `putbuf`
- Implement the exit syscall.

This will allow you to start passing the first few tests for userprog.

4.6 Design Document

Remember to complete the design document in `docs/userprog.tmpl` and save it as `src/userprog/DESIGNDOC`. The following are a subset of questions in the design document for your convenience. Remember, these questions might give you hints and tips on how to approach the project.

- In Pintos, the kernel separates commands into a executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.
- Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?
- Any access to user program memory at a user-specified address can fail due to a bad pointer value. Such accesses must cause the process to be terminated. System calls are fraught with such accesses, e.g. a "write" system call requires reading the system call number from the user stack, then each of the call's three arguments, then an arbitrary amount of user memory, and any of these can fail at any point. This poses a design and error-handling problem: how do you best avoid obscuring the primary function of code in a morass of error-handling? Furthermore, when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed? In a few paragraphs, describe the strategy or strategies you adopted for managing these issues. Give an example.
- The "exec" system call returns -1 if loading the new executable fails, so it cannot return before the new executable has completed loading. How does your code ensure this? How is the load success/failure status passed back to the thread that calls "exec"?
- Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait(C) before C exits? After C exits? How do you ensure that all resources are freed in each case? How about when P terminates without waiting, before C exits? After C exits? Are there any special cases?
- What advantages or disadvantages can you see to your design for file descriptors?
- The default `tid_t` to `pid_t` mapping is the identity mapping. If you changed it, what advantages are there to your approach?

5 Project 3: Virtual Memory

You can find the official documentation for project 3 on Stanford's website

The list of tests to pass for project 3 can be found in here.

This is written with reference to Colin Cammarano's guide from Spring of 2016.

5.1 Project Setup

Project 3 will require a working version of Project 2. You need to work on top of the project 2. Please note that you have submitted the code and design document of project 2 to the repo on Github Classroom before you work on this project.

5.1.1 Modify Kernel Path Information

Because you will begin to work under the directory `/home/csci3150/os-pintos/pintos/src/vm/` for this new project, you need to update some path information in Pintos:

(1) Firstly, in the line 259 of the file `/home/csci3150/os-pintos/pintos/src/utils/pintos`, replace the

```
my $name = find_file('/home/csci3150/os-pintos/pintos/src/userprog/build/
    kernel.bin');
```

with

```
my $name = find_file('/home/csci3150/os-pintos/pintos/src/vm/build/kernel.bin'
    );
```

(2) Secondly, in the line 362 of the file "/home/csci3150/os-pintos/pintos/src/utils/Pintos.pm", replace the

```
$name = find_file('/home/csci3150/os-pintos/pintos/src/userprog/build/kernel.
    bin') if !defined $name ;
```

with

```
$name = find_file('/home/csci3150/os-pintos/pintos/src/vm/build/kernel.bin')
    if !defined $name ;
```

(3) At last, Run a quick `make` and `make check` in the `src/vm` directory and `src/vm/build` directory respectively. You should be passing quite a few tests right away if you have a working version of project 2. A lot of these tests are exactly the same as the tests you've seen in project 2.

5.1.2 Keeping the Makefile Updated

You will be very likely adding your own header and source files for project 3. Whenever you create a new source file to use, make sure to update the makefile accordingly.

If you look under `src/`, you will find a file named `Makefile.build`. This is the file that keeps track of all the pathing to all the source files being used for the `vm` tests. If you look around line 64 of this file, you will come across these two lines:

```
# No virtual memory code yet.
#vm_SRC = vm/file.c             # Some file.
```

This is the place you want to add the path to your source code. For example:

```
# Virtual memory code.
vm_SRC = vm/new_file1.c
vm_SRC += vm/new_file2.c
vm_SRC += vm/new_file3.c
```

Note that the first file you add should just use an `"="`, all subsequent files should use `"+="`.

5.2 Before We Begin...

Project 3, more so than the other two projects you've done, is very open ended. A lot of this project is design–you will need to come up with designs that adequately fulfill the test cases. You will need to implement features such as page allocation on page faults, swap, eviction, memory mapped files, stack growth, and page reclamation. This all being said, this section will go over virtual memory and paging; we will also talk about the requirements of project 3 and suggest **possible** designs and data structures for each.

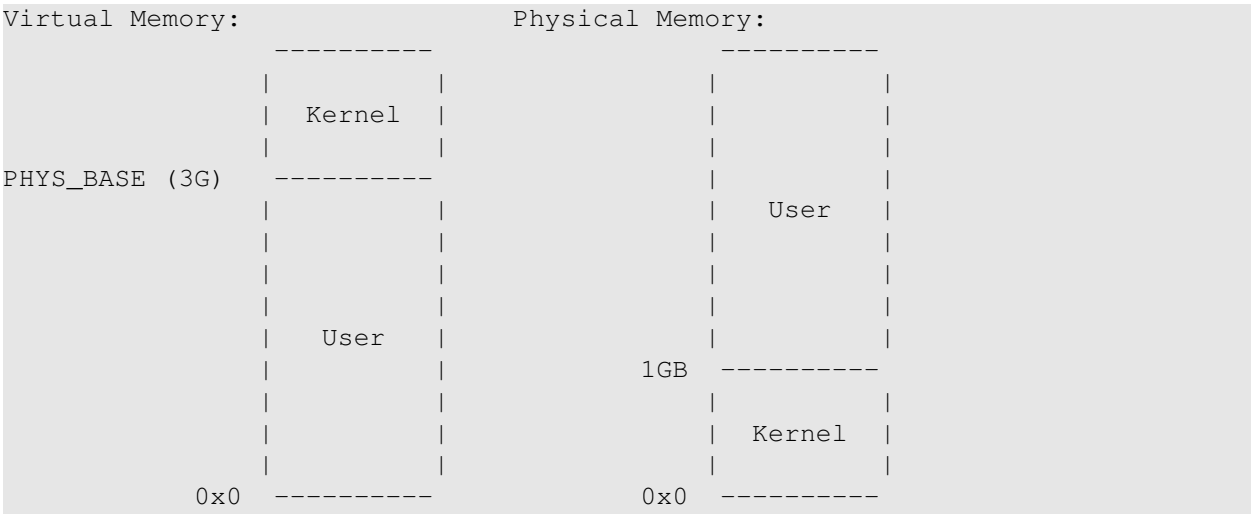
5.3 Introduction

Project 3 is a large project not just in terms of implementation, but in terms of its content and material as well. This section is not meant to give you direction on the project, but serves to go over a lot of the foundational knowledge you need all throughout this project. Yes reading this will be boring for most of you, yes you're reading this guide to know how to do the project and this is not what was promised, but we **highly** suggest you to read it nonetheless in order to reduce confusion.

5.3.1 Virtual Memory

Virtual memory is a memory management technique used by an OS to manage access to a computer's memory. The OS maps the computer's physical memory addresses to a series of unique software memory addresses, called virtual addresses. The mapping scheme varies between operating systems, but in Pintos, **this mapping is always 1-to-1 for kernel memory**, that is, each virtual kernel address maps exactly to a physical address.

In Pintos, virtual memory is separated into two pools: user and kernel memory. The user memory pool extends from `0x0` to `PHYS_BASE`, and the kernel memory pool extends from `PHYS_BASE` to the end of memory.



The mapping of kernel addresses to physical addresses is simple. The kernel physical address is always the kernel virtual address - `PHYS_BASE`. However this mapping does not necessarily hold true for user memory.

5.3.2 Paged Memory

Pages are fixed-size regions of contiguous virtual memory. Pages in a system will always be the same size; that is, if a system uses 4KB pages, then all pages in the system will also be 4KB. The number of pages in a system and their sizes can be determined by subdividing a virtual address into two parts–the page number and the offset.

For a 32 bit system, we say that memory addresses are 32-bit addressable, that is, a memory address is comprised of 32 bits. In the simplest paged virtual memory system, we split this address into two parts–the page number and the offset. The page number serves two purposes: it serves as a key for a lookup table (which I will explain later) and it determines the number of pages in a system. The offset also serves two purposes: it determines what byte of memory to address in a page, and it also determines the size of the pages.

The equations used to determine the number of pages and their sizes are as follows:

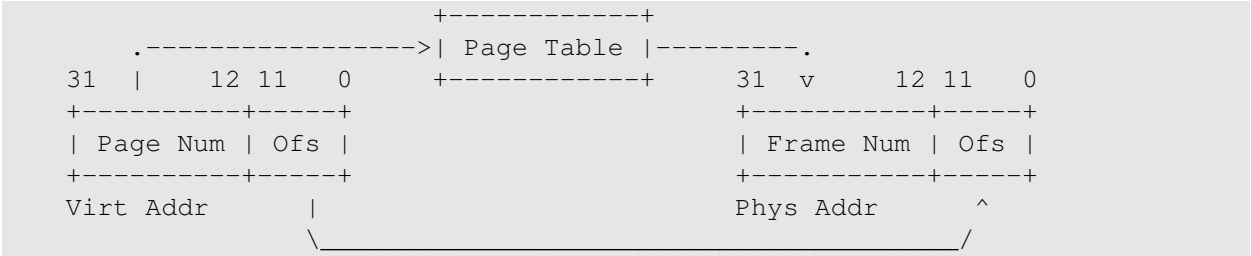
$$page\_size = 2^{offset} = \frac{2^{address\_size}}{2^{number\_pages}}$$

$$number\_of\_pages = 2^{number\_pages} = \frac{2^{address\_size}}{2^{offset}}$$

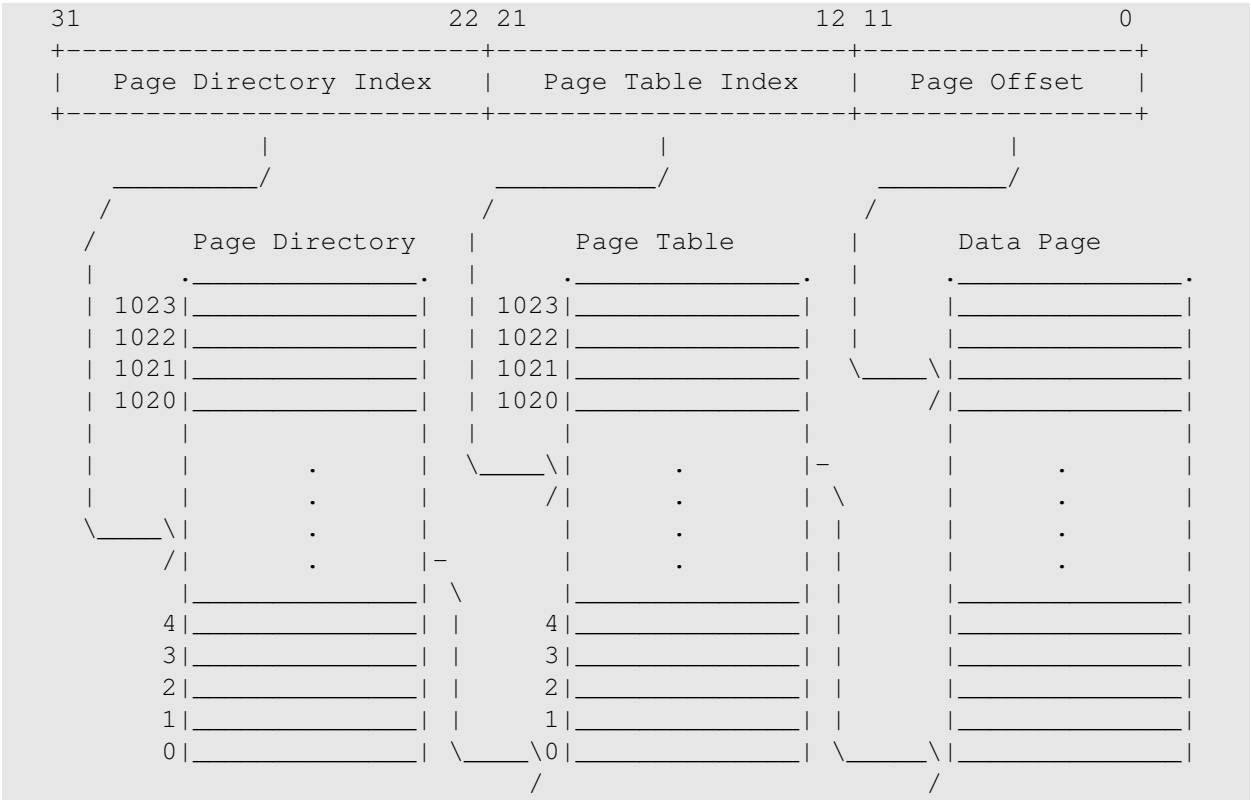
When a process wishes to access memory, it will use a virtual address. This virtual address is split into two parts by the OS: the first segment is the page number, the second is the offset. In a system like Pintos, where pages are 4KB in size, the page number will be the first *twenty bits* and the offset will be the last *twelve bits*. In the simplest case, the page number is sent to a data

structure called the page table, which translates the page number into a new, equal sized set of bits, called the frame number.

This number represents the first twenty most significant bits of a physical address. By concatenating the offset to this address, we end up with a complete physical memory address!



The lookup process for user pages in Pintos is a little more involved. Pintos has a global page table, called the page directory. When a lookup is performed, a virtual address is separated into two sections. The twenty most significant bits form the page number, and the last twelve bits form the offset, like before; however, in Pintos, the page number is further subdivided into two 10 bit numbers. The 10 most significant bits are called the *page directory index*, and the next 10 bits are the page table index. First, a lookup is performed in the page directory, using the page directory index as a key. The result of this lookup, assuming the address is mapped, is a page table. A lookup is then performed in this page table, using the page table index as the key. The result of this lookup is the frame address. Concatenating this with the offset yields the physical address. Below is an image from the documentation describing this process:



If any of these lookups fail, we get what's called a **page fault**. By default, this will terminate the process. We want to change this. We will be getting to this in a bit!

In Pintos, each process has its own page table, which manages a process's active (or mapped) pages. Every process has a page table that's allocated for it when the process loads; in fact, a process cannot succeed in loading unless a page table has been created for it. When a process calls `palloc_get_page`, a new page is allocated from a global pool of memory (assuming there's memory left to allocate), and a pointer to this page is stored in the process's page table. By default, when a page table is created, a single page is allocated from the global memory pool and mapped to this page table; and the page table contains mappings for kernel memory. As you saw in project 2, you can test to see if an address exists in the page directory by calling `pagedir_get_page`, which returns the physical address corresponding to the virtual address on success or a null pointer on failure.

5.3.3 Frames and Physical Memory

When dealing with pages, we are working wholly with virtual memory. When we describe a page in the context of physical memory, we call it a **frame**. Thus, virtual addresses correspond to addresses within pages and physical addresses correspond to addresses within frames.

As mentioned previously, these physical addresses are computed by first separating the virtual address into multiple sections: the page number (and its subdivisions) and the offset. A lookup is performed in the page table, using the page number as a key. The result of this lookup is a sequence of bits equal in length to the page number called the **frame number**. The physical address is then computed by concatenating the offset to this frame number.

Since Pintos maps kernel virtual memory 1-to-1 with physical memory, the addresses within a kernel page correspond exactly to the addresses in a kernel frame. Likewise, kernel pages 1 through n will be mapped exactly to kernel frames 1 through n. Remember that a kernel physical address is a kernel virtual address - `PHYS_BASE`.

5.3.4 Page Allocation and Management

Pintos already supports the allocation of pages via the use of `palloc`, but there is no readily accessible data structure that stores information about each allocated page. Behind the scenes, all available pages in the OS are represented as a bitmap in `palloc.c`. Furthermore, these pages are separated into two pools–kernel pages and user pages.

When a thread calls `palloc_get_page(FLAG)`, a pointer to the next free page is returned if there is a free page, otherwise, the function returns a null pointer. The `FLAG` argument is one of (or a bitwise or of) three constants: `PAL_USER` (a user page), `PAL_ZERO` (a completely emptied page), or `PAL_ASSERT`. You've probably seen the allocation function called as follows:

```
palloc_get_page (PAL_USER | PAL_ZERO);
```

This means that both flags will be applied. For a user process, we install this page into the thread's page table using the `install_page` function. This was already done for you in project 2 during the `setup_stack` function. This `install_page` function calls `pagedir_set_page`, which will map a page acquired by `palloc_get_page` (called the kernel page in the context of this function–even if it was allocated from the user pool) to the page table held by the current thread.

When a page is allocated by `palloc_get_page`, the global page table (the bitmap in `palloc.c`)

flags that area of memory as mapped (the bits are set to true). When a page is freed (for example, when `palloc_free_page` is called) those bits are set to false, which means that the page is now unmapped. You will want to look into these functions when implementing your new allocator functions.

5.3.5 Page Faults, and What to Do With Them

As mentioned earlier, a page fault occurs when a process attempts to access unmapped memory OR when a process attempts to access mapped memory outside of its own pages. For project 3, we will mainly be focusing on the first case. As I mentioned earlier, we get a page fault when a page table lookup fails. The page fault exception is, in fact, a software interrupt. The page fault handler, then, is an interrupt handler with with following signature and implementation (located in `src/userprog/exceptions.c`:

```
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->eip).
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
     (#PF)". */
  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
```

```
}
```

The behavior of this interrupt is very similar to the syscall software interrupt–by default, it pauses the current thread, executes some kernel code, then terminates the offending thread. In this situation, we want the page fault handler to allocate and install a new page for the process AND allow the process to continue as normal (rather than killing the thread) after the page fault handler finishes. The implementation details of this are up to you–there are a variety of ways of tackling this problem. As mentioned in the comment in the above snippet of code, you will want to handle your allocation code here in the page fault interrupt context. Thus, `exception.c` is a good place to start writing your new page fault memory allocation code.

The steps for allocating pages from the page fault handler are enumerated below, courtesy of the Pintos documentation:

1. Locate the page that faulted in the supplemental page table. If the memory reference is valid, use the supplemental page table entry to locate the data that goes in the page, which might be in the file system, or in a swap slot, or it might simply be an all-zero page. If you implement sharing, the page's data might even already be in a page frame, but not in the page table.

   If the supplemental page table indicates that the user process should not expect any data at the address it was trying to access, or if the page lies within kernel virtual memory, or if the access is an attempt to write to a read-only page, then the access is invalid. Any invalid access terminates the process and thereby frees all of its resources.
2. Obtain a frame to store the page. See section 4.1.5 Managing the Frame Table, for details.
3. Fetch the data into the frame, by reading it from the file system or swap, zeroing it, etc.
4. Point the page table entry for the faulting virtual address to the physical page. You can use the functions in `userprog/pagedir.c`.

5.4 Part 1: Growing the Stack

If a process wants to have more memory, you as the operating system should give more memory to the user. This is done through two steps:

1. The user will try to access a virtual memory that is not paged in its current page directory. This will cause a page fault.
2. In the page fault handler, you will determine if the address requested by the process is valid to allocate a new page for the process.

5.4.1 Validating the Address

There are two places you can validate a pointer. One is in the syscall hander (`userprog/syscall.c`) or in the page fault handler(`userprog/exception.c`). Remember, if a user process wants to access memory, they need to do this via a syscall, therefore you can detect a good number of invalid address requests in the syscall handler. You may also decide to detect every invalid pointer in the

page fault handler, that's ok too. Or you may decide to detect some in the syscall handler and some on the page fault handler; it's really up to you.

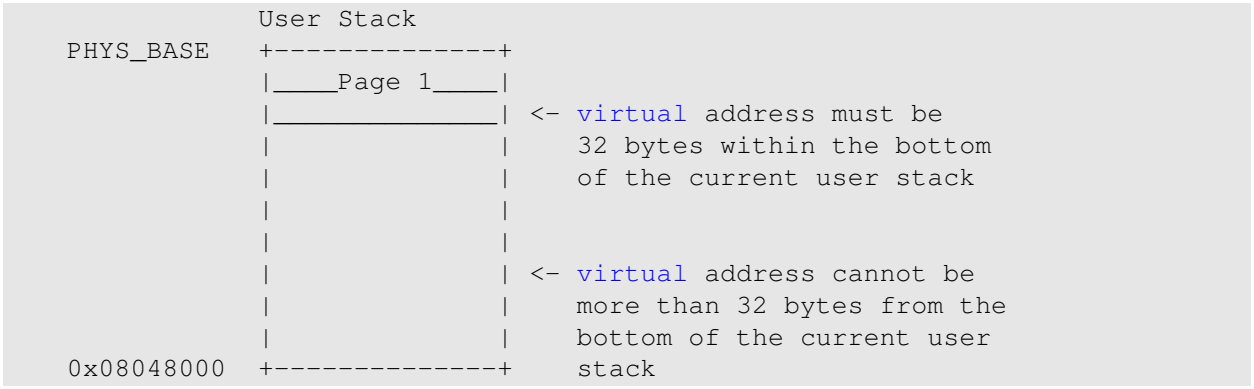
One thing to note is that if there is any kind of bad address request by the user program, a page fault will be called. If in the page fault handler you decide this address is actually not invalid and the process can proceed, make sure you do **not** run the last two lines of the page fault hander in `userprog/exception.c`:

```
printf ("Page␣fault␣at␣%p:␣%s␣error␣%s␣page␣in␣%s␣context.\n",
    fault_addr,
    not_present ? "not␣present" : "rights␣violation",
    write ? "writing" : "reading",
    user ? "user" : "kernel");
kill (f);
```

A good idea would to have some kind of success boolean to determine if this page fault should kill this process or proceed as normal.

Now, which virtual addresses are actually invalid?

- A `NULL` pointer
- A kernel address (the virtual address is larger or equal to the `PHYS_BASE`)
- An address below the user stack (virtual address is less than `0x08048000`)
- An address that does not appear to be a stack access; this means if the virtual address requested does not to appear to be the next contiguous memory page address of the stack (within 32 bytes of the stack pointer).



Any invalid virtual memory access should exit with a -1. Also remember that if the page fault is not caused by a user program, meaning the page fault is caused by the kernel, there is nothing you can do and must kill the process.

Finally, and perhaps most importantly, in the page fault handler, the virtual address that caused the page fault, in other words the virtual address you want to do all the validation and installing with, is the `void* fault_addr` as defined at the beginning of the function:

```
static void
page_fault (struct intr_frame* f)
```

```
{
    bool not_present;
    bool write;
    bool user;
    void* fault_addr;

    (...)

    kill(f);
}
```

Now these addresses can be anything, however for purposes of virtual memory, you want to see all the virtual memory that are in the same 4KB page as the same virtual address (the same page). Therefore you can round this address down to the nearest page boundary with `pg_round_down` found in `src/threads/vaddr.h`.

5.4.2 Allocating a New Page

Once you have confirmed that the virtual address the process wants is a valid pointer, you want to acquire a new frame from the physical memory (cache), and install that frame to be a page for the process.

You can acquire a new frame by:

```
//kpage stands for kernel page, this is the terminology
//used by Pintos
uint8_t* kpage = palloc_get_page(PAL_USER | PAL_ZERO);
```

Once you have the frame, you can install this frame as a page in the current thread's page directory by:

```
void* upage = pg_round_down(fault_addr);
void* kpage = (void*)palloc_get_page(PAL_USER | PAL_ZERO);
bool writable = true;
bool success = pagedir_set_page(thread_current()->pagedir, upage, kpage,
    writable);
```

This is a bit unncessary because in `userprog/process.c`, there is already a function named `install_page` which does the exact same thing AND validates whether or not if there is space in the page directory to install a page:

```
//in userprog/process.c
static bool
install_page (void* upage, void* kpage, bool writable)
{
    struct thread* t = thread_current();

    /* Verify that there's not already a page at the virtual address, then map
        our page there */
    return (pagedir_get_page(t->pagedir, upage) == NULL
            && pagedir_set_page(t->pagedir, upage, kpage, writable));
}
```

Feel free to use this function to install the new frame to a process' page directory. Once you have installed a new page, any virtual memory requested by the user that's within this new page will not cause a page fault.

5.4.3 Bookeeping for Frames and Pages

As of right now, you might now see the point of bookkeeping all the frames and pages you have allocated / installed. However as the project progresses, and becomes more and more complicated, these bookkeeping steps will prove to be absolutely useful.

For now, you can start with some really simple structs to store information on the pages and frames.

5.5 Frame Table

The frame table is a data structure that contains an entry for each frame that contains a user page; in other words, the frame table contains an entry for every page of memory allocated by the `palloc_get_page` function. The goal of this table is to make the process of eviction much more efficient by keeping information about frames both readily accessible and organized.

In its simplest incarnation, the frame table is a list or table of nodes which contain information about a frame and the process that owns it. For example, you could model the nodes in this table like so:

```
//a list of frame_table_entry as the page table
struct list frame_table;

struct frame_table_entry {
    uint32_t* frame;
    struct thread* owner;
    struct sup_page_entry* aux;
    // Maybe store information for memory mapped files here too?
};
```

If you decide to use this data structure, you will likely want to add frames to the table as they are allocated. The easiest way to do this would be to make a function that creates a frame table entry, calls `palloc_get_page`, stores this frame and the frame's owner in the entry, stores this entry in the frame table, then returns the newly allocated frame. You will likely need to add additional functions and data to this table as you work through project 3.

It will make organization much easier to create a separate `frame_table` header and source file in your `vm` directory. Remember to add the source file path to your makefile.

5.6 Supplementary Page Table

The supplementary page table is a data structure that stores additional information about pages not found in Pintos's default page table implementation. This table serves two main purposes: first, if there is a page fault, the supplementary page table allows Pintos to look up the faulting address and determine the type of data that should be there (i.e., if a new page should be allocated or if data should be pulled from the swap); the other purpose of this table is resource management–when a process exits, we can use this table to determine what needs to be freed. You should consider storing the page's virtual address and other page-specific data here, like so:

```
//a list or hashtable of sup_page_table_entry as your supplementary page table
//remember, each thread should have its own sup_page_table, so create a new
   list or hashtable member in thread.h

struct sup_page_table_entry {
    uint32_t* user_vaddr;
    /*
    Consider storing the time at which this page was accessed if you want to
       implement LRU!
    Use the timer_ticks () function to get this value!
    */
    uint64_t access_time;

    // You can use the provided PTE functions instead. I've posted links to
       the documentation below
    bool dirty;
    bool accessed;
}
```

5.7 Part 2: Swap, Eviction, and Reclamation

5.7.1 Overview of Swap, Eviction, and Reclamation

Sometimes, we need access to more memory than is available in our computer's primary memory. Virtual memory gives us a unique solution to this problem. That solution is called the **swap**. Swap space is space in secondary memory (such as an HDD or SDD) that is treated as primary memory by the operating system. In the most general case, we can treat this swap space as additional virtual memory. Why would we want to use secondary storage as additional primary memory? Well, I have an example below.

Let's say that we have a computer with 4GB of memory (RAM) and 1TB of secondary storage (HDD). Now, let's say we open a bunch of applications, like Photoshop, Chrome (complete with dozens of tabs), our media player of choice (maybe VLC), and our Pintos development VM. There's a very real possibility that the total amount of memory necessary to run these programs (and our OS of choice) is greater than 4GB; yet the OS doesn't force any of these programs to close. Behind the scenes, allocated pages of memory not currently in use are being copied to the HDD and freed to make room for any additional memory requests from our running programs. In this situation, you will likely notice that the programs run very slow–this is because secondary memory is far slower than primary memory. This is seen as an acceptable trade off–we trade read/write speed for the ability to run lots of programs concurrently.

The process of moving data from primary memory to swap space is called **eviction**. In its simplest form, eviction occurs when an OS runs out of available frames. If a page is requested and the OS cannot fulfill the request, it will choose a frame to remove (this choice is based upon our eviction algorithm), write the contents of the frame to the disk, free that frame in primary memory, then allocate it to the requesting process. Eviction algorithms can take a variety of forms, but the most common algorithm for eviction is LRU, or least recently used.

If now a process requests data from a page in which the frame the page's data was written in was evicted, we need to reclaim that frame back into primary memory, re-install the frame with the page, so that the process can get access to the data it rightfully owns. This process is called **reclamation**. This will be very similar to eviction, except rather than re-allocating a new frame to use, we bring a frame from swap to primary memory.

5.7.2 What is a Swap?

Pintos's test cases create a temporary virtual hard disk–this is where test files are copied to when the tests are run. We will be expanding Pintos's virtual memory capability to treat this disk as swap space. Hard drives have their storage media split into sectors, or equal sized regions of contiguous storage space. This holds true in Pintos, as the virtual disk is split into 512 byte sectors. The term for a page sized region of swap space is called a **swap slot**; in Pintos, we would say that a swap spot is represented by eight consecutive sectors on the virtual hard disk. Pintos already includes a handful of functions and data structures that allow us to manipulate data on the virtual hard disk.

The block structure, in particular, is very useful to us. We can use this block device to send data to our temporary hard disk. The block struct, defined in `src/devices/block.h`, is detailed below:

```
struct block
{
    struct list_elem list_elem;         /* Element in all_blocks. */

    char name[16];                      /* Block device name. */
    enum block_type type;                /* Type of block device. */
    block_sector_t size;                 /* Size in sectors. */

    const struct block_operations *ops;  /* Driver operations. */
    void *aux;                          /* Extra data owned by driver. */

    unsigned long long read_cnt;        /* Number of sectors read. */
    unsigned long long write_cnt;       /* Number of sectors written. */
};
```

5.7.3 Using Swap

Firstly, you can get a pointer to this block struct by calling `struct block *block_get_role(enum block_type)`:

```
//Make the swap block global
static struct block* global_swap_block;
```

```
//Get the block device when we initialize our swap code
void swap_init()
{
    global_swap_block = block_get_role(BLOCK_SWAP);
}
```

The `BLOCK_SWAP` flab ensures that we get a block device appropriately initialized for swap operations. We can perform read and write operations on the block by calling the following functions:

```
void block_read(struct block*, block_sector_t, void*);
void block_write(struct block*, block_sector_t, const void*);
```

The first parameter is the pointer to the block device we just initialized above, the second parameter is the sector (index) within the block we wish to read / write from, and the last parameter is the pointer to a buffer to read data into or write from.

Now remember that each block device is 512 bytes (defined as `BLOCK_SECTOR_SIZE` in `block.h`, and each page is 4096 bytes (defined as `PGSIZE` in `vaddr.h`, meaning it will take 8 blocks to hold the information of one page. This means if we want to read or write a page into disk using a block, you'll need to read or write 8 consecutive blocks (`PGSIZE / BLOCK_SECTOR_SIZE`):

```
void
read_write_from_block(uint8_t* frame, int index){
{
    //the frame is the frame I want to read into / write from
    //the index is the starting index of the block that is free
    for(int i = 0; i < 8; ++i)
    {
        //each read/write will rea/write 512 bytes, therefore we need to read/
           write 8 times, each at 512 increments of the frame
        block_read/write(block*, index + i, frame + (i * BLOCK_SECTOR_SIZE);
    }
}
```

It will be a very good idea to create a dedicated header and source file just for swap operations. Make sure to include the path to the source of this file in your makefile.

5.7.4 Eviction

When you want to grow the stack, you used `palloc_get_page` to get an available frame from cache. However when the cache is full, this function will return a `NULL`:

```
uint8_t* frame = palloc_get_page(PAL_USER | PAL_ZERO);
if (frame != NULL)
    //the cache wasn't full and you got a new frame. Go ahead and grow the
       stack
else
    //palloc get page returned a NULL, meaning you have to evict something
       from the cache.
```

Now at this point you know you want to evict a frame, but you have to decide which frame to evict. Optimally you'll want to use the Least Recently Used policy, so you'll need to be comparing the timestamps of when each frame was last accessed.

Once you have decided which frame you want to evict, it's time to evict! But before you get too excited, you need to manage some bookkeeping. If you decided to use both a frame table and a supplementary page table, you'll need to do some interesting bookkeeping:

- You will be evicting the frame, therefore you the page associated with the frame you have selected needs to be unlinked. Then you want to remove this frame from your frame table after you have freed the frame with `pagedir_clear_page`
- You do **not** want to delete the supplementary page table entry associated with the selected frame. The process that was using the frame should still have the illusion that they still have this page allocated to them. If you delete this page table entry, you will not be able to reclaim the data from disk when needed.
- Find a free block to write your data to. Since the blocks are just numbered contiguously, you just need an index that is free. Now this index is going to be needed to reclaim the data of the page, therefore it would be best to keep this index of where the data is in some member variable in the supplemental page table entry
- You'll also want to keep track of which pages are evicted and which are not for quick checking

Once you have the index of a free block, the frame you wish to evict and the corresponding page associated with it, you want to write the data of the frame to the block with `block_write`, store the index in the supplementary page table entry, update any other members necessary, free the frame with `pagedir_clear_page`, and remove the frame from the frame table.

After doing all this, you have successfully evicted a frame, meaning the next time you call `palloc_get_page` will return you a pointer to a free frame instead of giving you a NULL pointer.

5.7.5 Reclamation

Reclamation is going to work a lot similarly to growing the stack and evicting the stack.

Firstly you'll need to detect whether or not the virtual address that caused a page fault is from a page that has been evicted or not. If it is from a previous frame that has been evicted, you'll want to proceed with reclamation. If not, then it's just normal stack growing.

You'll want to evict a frame from the cache to make room to read from the disk into the cache. So you'll want to evict a frame just like you did in eviction. Once you have evicted a frame, you can now get a free frame to use. You'll want to re-link this new frame with the corresponding supplement page table entry in order to point the page to the right frame that has the data. You do **not** want to create a new supplement page table entry; you already have one from before, you want to reuse this page table entry.

Now comes the tricky part. You'll need to use `block_read` in order to read the contents of the block into the frame. Once finished, you don't need the data in the block anymore, so you can set the index of the block you just read from as free.

As always, you'll need to update all bookkeeping items as necessary.

5.8 Part 3: Memory Mapping

Memory mapping requires you to implement two new syscalls:

```
mapid_t mmap(int fd, void* addr);
void muunmap(mapid_t mapping);
```

`mapid_t` works a lot like the `fd`s, a unique identifier for each memory mapping you do for this process. A running `mapid_t` counter in each thread will be sufficient here. There are no reserved `mapid_t`.

5.8.1 `mmap` Syscall

You'll want to take the file associated with the `fd` passed in as the argument, and map that file into contiguous memory.

This is actually **a lot** of fun. There are some pretty neat address math and indexing you have to do. You will find the `load_segment` function in `process.c` very helpful to figure out the math. Here are some helpful tips:

- You get the length of a file with:

  ```
  //file_length function in file.h
  uint32_t length = file_length(file);
  ```

- This file may have been opened before, so instead of opening a fresh file, you can use `file_reopen` to reopen a file to refresh.
- For each page you allocate for this file to map, you'll want to add a new frame table entry as well as a new supplementary page table entry to go along with it.

5.8.2 `munmap` Syscall

This syscall is pretty simple. All you want to do is unmap all the pages you mapped for the designated `mapid_t`. Removing all the supplementary page table entries and clearing the frames allocated (if allocated) is all you have to do.

5.9 Design Document

Remember to complete the design document in `docs/vm.tmpl` and save it as `src/vm/DESIGNDOC`. The following are a subset of questions in the design document for your convenience. Remember, these questions might give you hints and tips on how to approach the project.

- How does your code coordinate accessed and dirty bits between » kernel and user virtual addresses that alias a single frame, or » alternatively how do you avoid the issue?
- When two user processes both need a new frame at the same time, » how are races avoided?
- Why did you choose the data structure(s) that you did for » representing virtual-to-physical mappings?
- When a frame is required but none is free, some frame must be » evicted. Describe your code for choosing a frame to evict.
- When a process P obtains a frame that was previously used by a » process Q, how do you adjust the page table (and any other data » structures) to reflect the frame Q no longer has?
- Explain your heuristic for deciding whether a page fault for an » invalid virtual address should cause the stack to be extended into » the page that faulted.
- Explain the basics of your VM synchronization design. In » particular, explain how it prevents deadlock. (Refer to the » textbook for an explanation of the necessary conditions for » deadlock.)
- A page fault in process P can cause another process Q's frame » to be evicted. How do you ensure that Q cannot access or modify » the page during the eviction process? How do you avoid a race » between P evicting Q's frame and Q faulting the page back in?
- Suppose a page fault in process P causes a page to be read from » the file system or swap. How do you ensure that a second process Q » cannot interfere by e.g. attempting to evict the frame while it is » still being read in?
- Explain how you handle access to paged-out pages that occur » during system calls. Do you use page faults to bring in pages (as » in user programs), or do you have a mechanism for "locking" frames » into physical memory, or do you use some other design? How do you » gracefully handle attempted accesses to invalid virtual addresses?
- A single lock for the whole VM system would make » synchronization easy, but limit parallelism. On the other hand, » using many locks complicates synchronization and raises the » possibility for deadlock but allows for high parallelism. Explain » where your design falls along this continuum and why you chose to » design it this way.
- Describe how memory mapped files integrate into your virtual » memory subsystem. Explain how the page fault and eviction » processes differ between swap pages and other pages.
- Explain how you determine whether a new file mapping overlaps » any existing segment.
- Mappings created with "mmap" have similar semantics to those of » data demand-paged from executables, except that "mmap" mappings are » written back to their original files, not to swap. This implies » that much of their implementation can be shared. Explain why your » implementation either does or does not share much of the code for » the two situations.

6 Project 4: File System

6.1 Project Setup

You have 2 choices for your starting point for project 4:

1. Build project 4 on top of project 2
2. Build project 4 on top of project 3

In both cases, you must have all the functionality of project 2 working.

If you decide to build on top of project 3, you will need to modify the file `filesys/Make.vars` to enable VM functionality (uncomment the bottom four lines):

```
#Uncomment the lines below to enable VM.
kernel.bin: DEFINES += -DVM
KERNEL_SUBDIRS += vm
TEST_SUBDIRS += tests/vm
GRADING_FILE = $(SRCDIR)/tests/filesys/Grading.with-vm
```

Finally, make sure update the kernel path information in `pintos` and `Pintos.pm` to make them point to the the `filesys` directory. Please refer to section **Modify Kernel Path Information** in previous project to know how to update it.

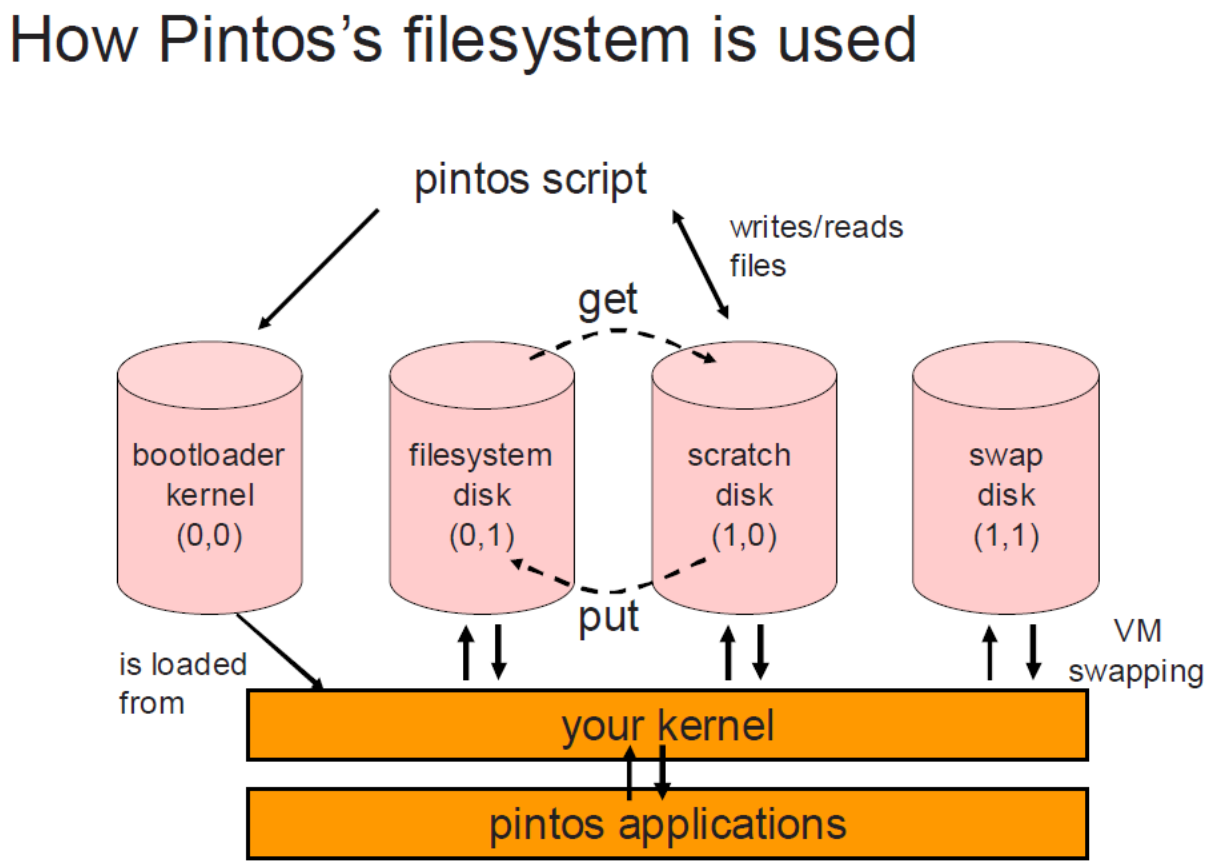
6.2 Introduction

Let's begin with what a file is in terms of operating systems. A file is a named collection of related information that is recorded on secondary storage (magnetic disks, magnetic tapes, optical disks etc.).The system that manages all files is the file system(filesys for shorthand), and the filesys provides persistent storage. A basic filesys has already been implemented in Pintos; however, this basic filesys only support fixed-sized files and is stored within consecutive sectors. Project 4 will support the following:

- Buffer Cache
- Extensible Files
- Subdirectories
- *(hidden mission of file system synchronization)

6.3 File System Usage

6.3.1 How Pintos' Filesys is Used



6.3.2 Sector and Disk

We refer to a section or a block of a physical storage disk as a **sector**. You can think of a disk as an array (it's not actually, for for simplicity's sake we can), and a sector is one entry is the array. To read/write data from/to a sector, we need a sector number or ID, which is just like the array's indices.

6.3.3 `free_map`

```
static struct bitmap *free_map;         /* Free map, one bit per sector. */
```

As we did in project 3 for swap block, Pintos also uses a bitmap to track which sectors in the disk are available. To keep this `free_map` persistent across Pintos, `free_map` is also written to the disk at sector 0 by default.

6.3.4 `inode_disk`

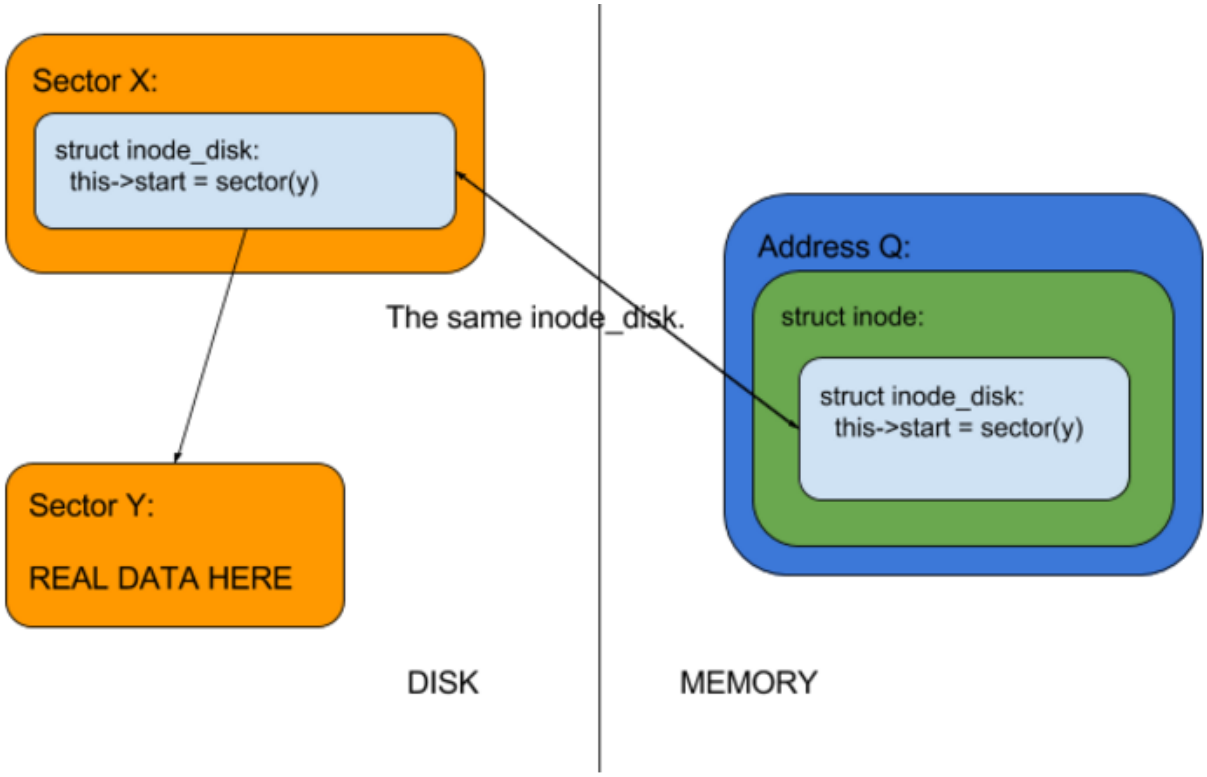
```
struct inode_disk
{
    block_sector_t start;       /* First data sector. */
    off_t length;               /* File size in bytes. */
    unsigned magic;             /* Magic number. */
    uint32_t unused[125];       /* Not used. */
};
```

`inode_disk` has the exact size of `BLOCK_SECTOR_SIZE`, which is 512 bytes. You can think of it as the metadata of a file. `block_sector_t` is the number of which the first real file is stored, and `off_t` is the total length of the file. Since files are stored in consecutive sectors, we could retrieve every single byte of a file easily with these two members. `inode_disk` is supposed to be persistent so it is written (serialized) to the disk. When we need to access a file that is stored in the disk, we find where its `inode_disk` is located through directories (explained later) and "deserialize" it from the disk.

6.3.5 `inode` (memory inode)

```
/* In-memory inode. */
struct inode
{
    struct list_elem elem;      /* Element in inode list. */
    block_sector_t sector;      /* Sector number of disk location. */
    int open_cnt;               /* Number of openers. */
    bool removed;               /* True if deleted, false otherwise. */
    int deny_write_cnt;         /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;     /* Inode content. */
};
```

To distinguish this data structure from `inode_disk`, we will refer the inode as **memory inode** since it is stored in memory. Memory inode is effectively a wrapper for `inode_disk`, and it has a `struct inode_disk data`, which is the underlying "metadata" that tells which sector stores that file. We will explain how memory inode and `inode_disk` works in relation with each other and how they support the basic file operations later. The figure on the next page outlines the organization of `inode_disk` and `inode`.



6.3.6 `dis` and `dir_entry`

```
/* A directory */
struct dir
{
    struct inode *inode;        /* Backing store. */
    off_t pos;                  /* Current position. */
};

/* A single directory entry. */
struct dir_entry
{
    block_sector_t inode_sector;        /* Sector number of header. */
    char name[NAME_MAX + 1];            /* Null terminated file name. */
    bool in_use;                        /* In use or free? */
};
```

As we learned in class, a directory is a file as well. Therefore, dir has a `struct inode *inode` that associate with its underlying block sector. `dir_entry` is used for retrieving the metadata of a file by its filename. One `dir_entry` is associated with one `inode_disk` (just like one `inode_disk` is associated with one file). For now, one `dir` can have at most 16 `dir_entry`.

Example

Root directory's `inode_disk` is stored at sector 1 by default, and to fetch files under the root directory, we need to:

1. Fetch `inode_disk` at sector 1.
2. Based on the start variable of `inode_disk`, we read data from sector (`inode_disk->start`) to a buffer, which stores a series of `struct dir_entry`.
3. For `dir_entry` in buffer:
   - Fetch `inode_disk` from sector (`dir_entry->inode_sector`).
   - Read file data from sector (`inode_disk->start`).
4. In case we need to fetch files under root directory in the future, create a `struct dir` for the root directory:

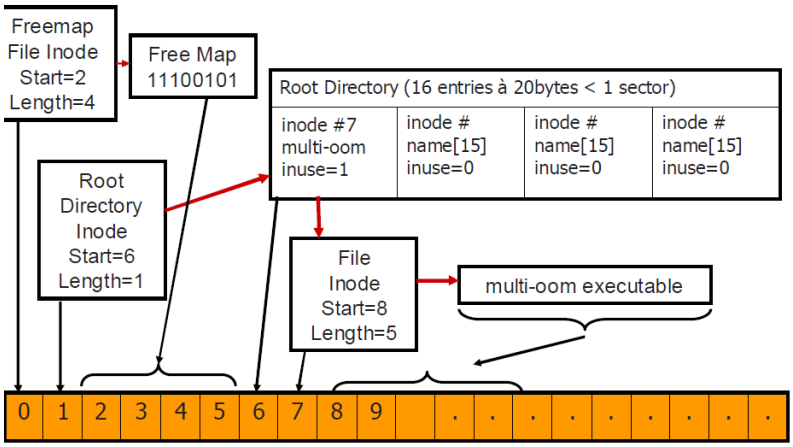
```
block_sector_t sector = 1;
/* Allocate memory. */
struct inode = malloc (sizeof(struct inode));

/* Initialize. */
inode->sector = sector;
inode->open_cnt = 1;
inode->deny_write_cnt = 0;
inode->removed = false;
block_read(physical_disk, inode->sector, &inode->data);

/* Create our root_directory dir*/
struct dir = malloc(sizeof(struct dir));
dir->inode = inoode
```

6.3.7 File System Disk Sector Layout

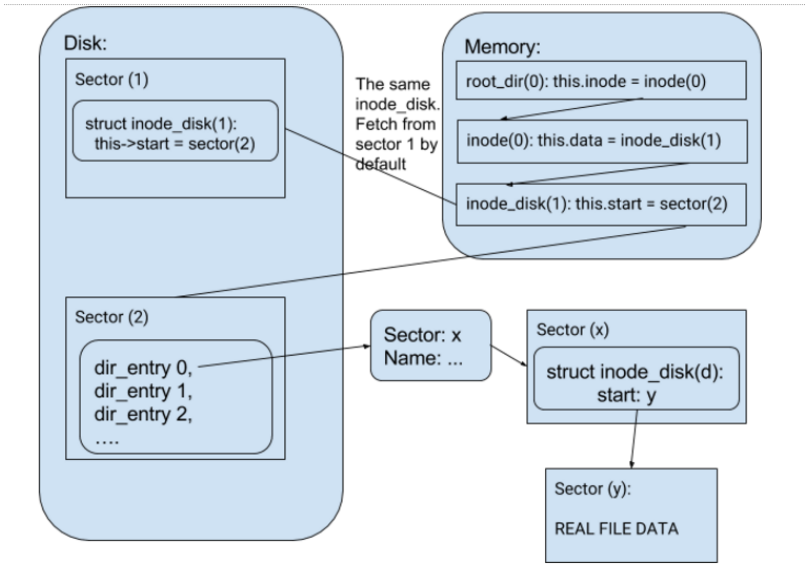
The figure below outlines the relationship between disk sector layout, inodes, and files for the above example.



Disk Sectors

6.3.8 Memory and Disk Sector

The figure below outlines the relationship between the disk sectors and memory.



6.4 Before We Continue

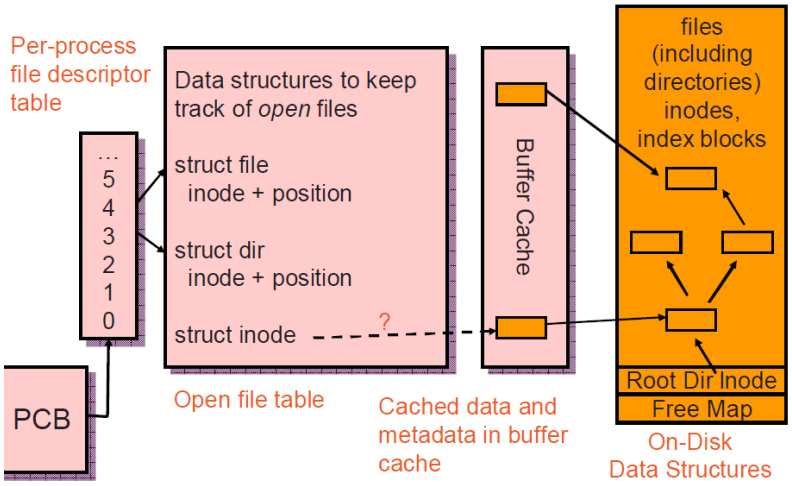
Since project 4 is really open-ended and at this point you have a lot of experience with Pintos (threads, user programs, and virtual memory), the following section will give you a broad introduction to the problems and a brief discussion on the possible solutions. You are more than welcome to come up with your own design.

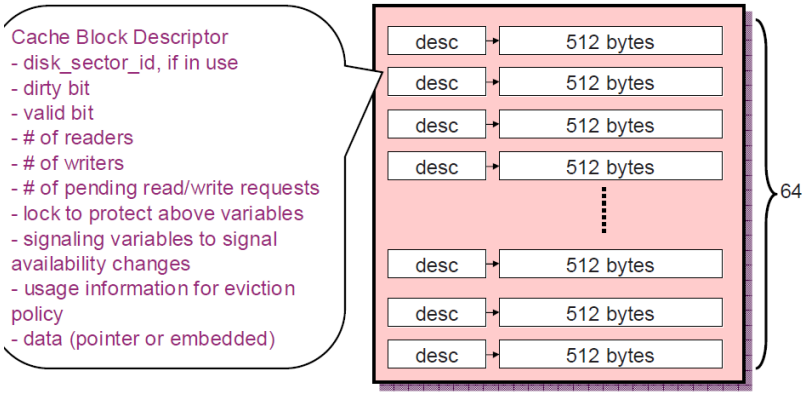
The Stanford Pintos guide has a recommended implementation order (Buffer Cache -> Extensible Files -> Subdirectories). Please follow this order to avoid some unnecessary trouble.

6.5 Part 1: Buffer Cache

You have already done virtual memory for project 3, so you should be an expert for this section!

Modify the file system to keep a cache of file blocks. When a request is made to read or write to a block, check to see if it is in the cache, and if so, use the cache data without going to disk. Otherwise, fetch the block from disk into cache, evicting an older entry if necessary. You are limited to a cache no greater than 64 sectors in size. In other words, buffer cache is the only bridge between memory and disk.





6.5.1 `cache.h/.c`

You need to add two new files, `cache.h` and `cache.c`, and declare your own cache interface. For the cache buffer itself, declare a static array of 64 blocks in `cache.c` would work. Using structure hiding would make things easier (private members should be declared in the `.c` file).

For eviction, use LRU or any other policy you'd like.

For synchronization, per-block locking is a good choice. Synchronization per-condition will be better but harder to implement.

6.5.2 Write-behind and Read-ahead

Your cache should be write-behind, that is, keep dirty blocks in the cache, instead of immediately writing modified data to disk. Write dirty blocks whenever they are evicted. Because write-behind makes your file system more fragile in the face of crashes, in addition you should periodically write all dirty, cached blocks back to disk. The cache should also be written back to disk in `filesys_done()`, so that halting Pintos flushes the cache.

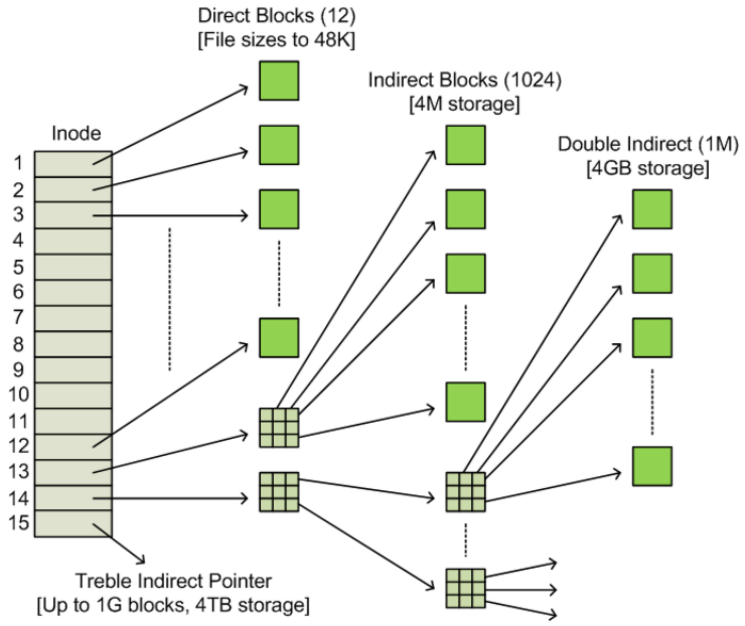
You should also implement read-ahead, that is, automatically fetch the next block of a file into the cache when one block of a file is read, in case that block is about to be read. Read-ahead is only really useful when done asynchronously. That means, if a process requests disk block 1 from the file, it should block until block 1 is read in, but once that read is complete, control should return to the process immediately. The read-ahead request for disk block 2 should be handled asynchronously, in the background.

For periodical write-behind and read-ahead, it is important that they are asynchronous. All you need to do is create two new threads, each of which takes care of one of the two jobs. You can use `thread_create` to create a new thread and pass in your own `thread_func` function.

6.6 Part 2: Indexed and Extensible Files

The basic file system allocates files as a single extent, making it vulnerable to external fragmentation, that is, it is possible that an n-block file cannot be allocated even though n blocks of non-contiguous memory are free. You can eliminate this problem by modifying the on-disk `inode` structure. In practice, this means using an indexing structure with direct, indirect, and double-indirect blocks. You are more than welcome to choose a different design.

You can assume that the file system partition will not be larger than 8 MB. Supporting 8 MB files will require you to implement, at the very minimum, double-indirect blocks. You'll need to replace `block_sector_t start` in `inode_disk` with any array structure that stores direct, indirect, and double-indirect blocks.



The figure above outlines the idea of the indexed file structure, but we definitely do not need a block array of 15 entries. A total of 12 entries with 10 direct blocks, 1 indirect block, and 1 double-indirect block will be enough for Pintos. One way to implement such a structure is to declare an array of sector numbers:

```
block_sector_t blocks[BLOCK_NUMBER];
```

6.6.1 `inode_create` Function

Since the `inode_disk` structure is changed, you'll need to modify how this structure is created by updating the `inode_create()` function. You'll want to initialize the block array carefully as well as any other members you'll need to initialize.

6.6.2 `inode_read_at` and `inode_write_at` Functions

The user programs are allowed to seek beyond the current end-of-file (EOF). The seek itself does not extend the file. Writing at a position past EOF extends the file to the position beign written, and any gap between the previous EOF and the start of the write must be filled with 0s.

A read starting from a position past EOF returnes no bytes.

Writing far beyond EOF can cause many blocks to be entirely zeroed. Some file systems allocate and write real data blocks for these implicitly zeroed blocks. Other file systems do not allocate these blocks at all until they are explicitly written (kind of like memory mapping). The latter file systems are described to support "sparse files". You may adopt either of these allocation strategies in your file system.

No matter which allocation strategy you choose, you must support file extension for the writing function. Since you'll need to extend the file in `inode_create()` as well, you might want to modulate your extension functionality.

6.6.3 `inode_delete` Function

When a file is deleted, its associated `inode_disk` should be deleted as well. Make sure to free all resources allocated to the `inode_disk` and update the `free_map` accordingly. If you are using some kind of nested array for multi-level indices, you will want to free every allocated entry in the array.

6.7 Part 3: Subdirectories

In this section you'll need to implement a hierarchical name space. In the basic file system, all files live in the root directory. Modify this to allow directory entries to point to files or other nested directories. This requires you to track the type of file (directory or plain file) in the `inode_disk` structure. Note that Operations on files should not be done on directories.

Make sure that directories can be expanded beyond their original size just as any other file can. This should be done automatically once part 1 is done. The basic file system has a 14-character limit on filenames. You may retain this limit for individual filenames, or you may extend it. Make sure, at minimum, you allow full path names to be longer than 14 characters.

6.7.1 Directory Lookup

For directory lookup, you'll want to maintain a separate current directory for each process. At startup, set the root as the initial process' current directory. When one process starts another with the exec system call, the child process inherits its parent's current directory. After that, the two processes' current directories are independent, so that changing one will have no effect on the other.

Filenames are used everywhere in file systems. You will need to support both absolute filenames and relative filenames. This means you might want to implement some parsing function to deal with filenames. The directory separator character is a forward slash ("/"). There are also other special characters you must support: "`.`", "`./`", "`..`", and "`../`". There are many solutions for this requirement. Two suggestions are:

1. In your parsing function, remove the special characters and perform their effects on their filename, and return the final absolute filename. Don't forget to make a copy of the filename since `strtok_r()` will mutate the string you pass into the function.
2. Create two `dir_entries` (`.`) and (`..`) on disk when a directory is created. Treat (`.`) and (`..`) as two directory entries that tells you where they lead to.

6.7.2 Update Existing Syscalls

All syscalls should support absolute and relative filenames. Parse their arguments before processing system calls.

- Syscall `open`: Update the open syscall so that it can also open directories.
- Syscall `close`: Update the close syscall so that it can also close directories.
- Syscall `remove`: Update the remove syscall so that it can delete empty directories (other than the root) in addition to regular files. Directories may only be deleted if they do not contain any files or subdirectores (other than `.` and `..`). You may decide whether to allow deletions for a directory that is opened by another process or is in use as another process' currect working directory. If it is allowed, then attempts to open files (including `.` and `..`) or create new files in a deleted directory must be disallowed.

6.7.3 Adding New Syscalls

You will need to write new syscalls for project 4:

1. `bool chdir(const char *dir)`

   Change the current working directory of the process to `dir`, which may be relative or absolute. Returns `true` if successful, `false` on failure.

2. `book mkdir(const char *dir)`

   Creates the directory named `dir`, which may be relative or absolute. Returns `true` if successful, `false` on failure. Fails if `dir` already exists or if any directory name in `dir`, besides the last, does not already exist. That is, `mkdir("a/b/c")` succeeds only if `/a/b` already exists and `/a/b/c` does not.

3. `bool readdir(int fd, char *name)`

   Reads a directory entry from file descriptor `fd`, which must represent a directory. If successful, stores the null-terminated filename in name, which must have room for `READDIR_MAX_LEN + 1` bytes, and returns `true`. If no entries are left in the directory, returns

`false`. Besides, `(.)` and `(..)` should not be returned by `readdir`. If the directory changes while it is open, then it is acceptable for some entries not to be read at all or to be read multiple times. Otherwise, each directory entry should be read once, in any order. `READDIR_MAX_LEN` is defined in `lib/user/syscall.h`. If your file system supports longer filenames than the basic file system, you should increase this value from the default of 14.

4. `bool isdir(int fd)`

   Returns `true` if `fd` represents a directory, `false` if it represents an ordinary file.

5. `int inumber(int fd)`

   Returns the inode number of the inode associated with `fd`, which may represent an ordinary file or a directory.

*An inode number persistently identifies a file or directory. It is unique during the file's existence. In Pintos, the sector number of the inode is suitable for use as an inode number.

Synchronization

Since files are shared across different processes, you need to pay attention to the synchronization of files/directories. The most important part is the synchronization of open and close files/directories. One simple but very inefficient way is to add a lock for the entire file system. Alternatives could be adding locks on different files/directories or on related file operations.

6.8 Design Document

Remember to complete the design document in `docs/filesys.tmpl` and save it as `src/filesys/DESIGNDOC`. The following are a subset of the questions in the design document for your convenience. Remember, these questions might give you hints and tips on how to approach the project.

- What is the maximum size of a file supported by your inode structure? Show your work.
- Explain how your code avoids a race if two processes attempt to extend a file at the same time.
- Suppose processes A and B both have file F open, both positioned at end-of-file. If A reads and B writes F at the same time, A may read all, part, or none of what B writes. however, A may not read data other than what B writes, e.g. if B writes nonzero data, A is not allowed to see all zeros. Explain how your ode avoids this race.
- Explain how your synchronization design provides "fairness". File access is "fair" if readers cannot indefinitely block writers or vice versa. That is, many processes reading from a file cannot prevent forever another process from writing the file, and many processes writing to a file cannot prevent another process forever from reading the file.
- Is your inode structure a multilevel index? If so, why did you choose this particular combination of direct, indirect, and doubly indirect blocks? If not, why did you choose an alternative inode structure, and what advantages and disadvantages does your structure have, compared to a multilevel index?
- Describe your code for traversing a user-specified path. How do traversals of absolute and relative paths differ?

- How do you prevent races on directory entries? For example, only one of two simultaneous attempts to remove a single file should succeed, as should only one of two simultaneous attempts to create a file with the same name, and so on.
- Does your implementation allow a directory to be removed if it is open by a process or if it is in use as a process' current working directory? If so, what happens to that process' future file system operations? If not, how do you prevent it?
- Explain why you chose to represent the current directory of a process the way you did.
- Describe how your cache replacement algorithm chooses a cache block to evict.
- Describe your implementation of write-behind.
- Describe your implementation of read-ahead.
- When one process is actively reading or writing data in a buffer cache block, how are other processes prevented from evicting that block?
- During the eviction of a block from the cache, how are other processes prevented from attempted to access the block?
- Describe a file workload likely to benefit from buffer caching, and workloads likely to benefit from read-ahead and write-behind.

7 Miscellaneous

Here is a collection of other facts that you might want to keep in mind:

- The number after the `-T` flag indicates the number of seconds that test will be run before being terminated and treated as a failed test. For example `pintos -v -k -T 60 -bochs --q run alarm-wait` will run for 60 seconds before terminating.

8 Test Cases

You can find the test cases for each project in the following links(The test cases of project 1 in the following link is incomplete):

Project 1 Project 2 Project 3 Project 4