pintOS is a computer software, a simple instructional operating system framework for the x86 instruction set architecture. It supports kernel threads, loading and running user programs and file systems, but implements them all in a very simple way.

In order to understand the operating system, it is important to implement each concept(User program, Thread, Virtual memory, Filesystem) directly. Commercial operating systems such as Linux take at least an hour to compile, but pintOS is more simple, easy to understand and fast to compile.

**User program**

- Pintos can run normal C programs, as long as they fit into memory and use only the system calls you implement. Notably, malloc() cannot be implemented because none of the system calls required for this project allow for memory allocation. Pintos also can't run programs that use floating point operations, since the kernel doesn't save and restore the processor's floating-point unit when switching threads.

The src/examples directory contains a few sample user programs. The Makefile in this directory compiles the provided examples, and you can edit it compile your own programs as well.

**Source files**

"process.c"

"process.h"

Loads ELF binaries and starts processes.

"pagedir.c"

"pagedir.h"

A simple manager for 80*x*86 hardware page tables.

"syscall.c"

"syscall.h"

Whenever a user process wants to access some kernel functionality, it invokes a system call. This is a skeleton system call handler.

"exception.c"

"exception.h"

When a user process performs a privileged or prohibited operation, it traps into the kernel as an "exception" or "fault."

"gdt.c"

"gdt.h"

The 80*x*86 is a segmented architecture. The Global Descriptor Table (GDT) is a table that describes the segments in use. These files set up the GDT.

"tss.c"

"tss.h"

The Task-State Segment (TSS) is used for 80*x*86 architectural task switching. Pintos uses the TSS only for switching stacks when a user process enters an interrupt handler, as does Linux.

**Using the file system**

**How User programs work**

Pintos can run normal C programs, as long as they fit into memory and use only the system calls you implement.

Pintos can load ELF executables with the loader provided for you in userprog/process.c. ELF is a file format used by Linux, Solaris, and many other operating systems for object files, shared libraries, and executables. You can actually use any compiler and linker that output 80x86 ELF executables to produce programs for Pintos.

**Virtual Memory Layout**

Kernel virtual memory is global. It is always mapped the same way, regardless of what user process or kernel thread is running. In Pintos, kernel virtual memory is mapped one-to-one to physical memory, starting at PHYS\_BASE. That is, virtual address PHYS\_BASE accesses physical address 0, virtual address PHYS\_BASE + 0x1234 accesses physical address 0x1234, and so on up to the size of the machine's physical memory.

A user program can only access its own user virtual memory. An attempt to access kernel virtual memory causes a page fault, handled by page\_fault() in userprog/exception.c, and the process will be terminated. Kernel threads can access both kernel virtual memory and, if a user process is running, the user virtual memory of the running process. However, even in the kernel, an attempt to access memory at an unmapped user virtual address will cause a page fault.

**Accessing User Memory**

As part of a system call, the kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above PHYS\_BASE). All of these types of invalid pointers must be rejected without harm to the kernel or other running processes, by terminating the offending process and freeing its resources.

**System call**

Implement the system call handler in userprog/syscall.c. The skeleton implementation we provide "handles" system calls by terminating the process. Instead, it will need to retrieve the system call number, then any system call arguments, and carry out appropriate actions.

- void **halt** (void)

- Terminates Pintos by calling shutdown\_power\_off() (declared in devices/shutdown.h). This should be seldom used, because you lose some information about possible deadlock situations, etc.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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

- int **filesize**(int fd)

Returns the size, in bytes, of the file open as fd.

- 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().

- 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.

- 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 will 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.

- 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.

- 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.