Phase 0

1. Attached
2. Below

Kernel command line: run halt

Pintos booting with 4,088 kB RAM...

382 pages available in kernel pool.

382 pages available in user pool.

Calibrating timer... 3,352,166,400 loops/s.

hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"

hda1: 167 sectors (83 kB), Pintos OS kernel (20)

hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"

hdb1: 4,096 sectors (2 MB), Pintos file system (21)

filesys: using hdb1

Boot complete.

Executing 'halt':

Timer: 55 ticks

Thread: 0 idle ticks, 0 kernel ticks, 55 user ticks

hdb1 (filesys): 33 reads, 0 writes

Console: 574 characters output

Keyboard: 0 keys pressed

Exception: 0 page faults

Powering off...

1. The very first thing that runs is the BIOS, which primes hardware for the loader. Then the bootloader (loader.S) runs, drawing out the Pintos kernel from memory and loading it into the system. Pintos’ main procedure starts up various subsystems so that it is ready to run its first process. The first user-space program is ran as a process. init.c in src/threads runs various functions that initialize subsystems. Beyond that, a scheduler will allocate time based on priority to various processes that the user or system creates.
2. The bootloader is in assembly because the system is not yet set up to handle C yet. The processor can only understand assembly before Pintos and GCC are loaded into memory.
3. These procedures use assembly code because it allows us to specifically direct which registers to use for each command. Using GCC limits the amount of control the user has, especially when multiple arguments have to be handled. Using the default registers GCC assigns when it compiles C code may make complex system calls difficult or buggy.
4. When Pintos boots, “syscall\_init” is one of the very first functions called. In it, “intr\_register\_int” is called with parameters “0x30” (interrupt number) and “syscall\_handler.” This registers the number with the system call handler so that when a program invokes 0x30, the processor executes “syscall\_handler” after turning on privileged mode. In this way, other interrupt handlers can be registered to an interrupt number by using “intr\_register\_int.”
5. The privilege bit starts turned off during general user-space program use. When a user-space program invokes a halt system call, syscall0 (parameter-less) from syscall.c takes in a system call identifier number (listed in syscall-nr.h). In this case, we’re using SYS\_HALT (confusingly enough, these are not actually numbers). syscall0, using assembly code, pushes (push1) SYS\_HALT onto the stack and then invokes 0x30 (int $0x30), which transfers control to syscall\_handler. The privilege bit is turned on at this point. The syscall\_handler then pops the system call identifier off the stack (SYS\_HALT) and executes it. At this point, SYS\_HALT halts the computer.

#include <stdio.h>

// pass in addresses, initialize as pointers to be able to reassign variables

void swap(int\* pa, int\* pb) {

int hold = \*pa;

\*pa = \*pb;

\*pb = hold;

}

int main(int argc, char \*argv[]) {

int a = 1, b = 2;

swap(&a, &b);

printf("a was 1, it is now %d\n", a);

printf("b was 2, it is now %d\n", b);

// print out command-line arguments

int i;

for (i = 0; i < argc; i++) {

printf("%s\n", argv[i]);

}

return 0;

}

// https://stackoverflow.com/questions/8403447/swapping-pointers-in-c-char-int

// The top answer in this stackoverflow post highlighted the problem I was having in

// "swap2," which was not identifying the type of arguments I was sending to swap.

// I only had "pa" and "pb" initially, but changed it to "int\* pa" and "int\* pb"

// to clearly show that it was a pointer being passed in. 04 SEP 18, West Point, NY.

// The CSE citation is in the references section of the write-up.

References

Swapping pointers in C (char, int). Stack Overflow. 2011 Dec 6 [accessed 2018 Sep 4]. <https://stackoverflow.com/questions/8403447/swapping-pointers-in-c-char-int>