**Detailed Notes from Reading Pintos Manual**

**Suggested Order of Implementation:**

1. Argument passing (see Section 3.3.3 [Argument Passing], page 29). Every user program will page fault immediately until argument passing is implemented.

For now, you may simply wish to change

\*esp = PHYS\_BASE;

To

\*esp = PHYS\_BASE – 12;

In setup\_stack(). That will work for any test program that doesn’t examine arguments, although its name will be printed as (null).

Until you implement argument passing, you should only run programs without passing command-line arguments. Attempting to pass arguments to a program will include those arguments in the name of the program, which will probably fail.

1. User memory access (see Section 3.1.5 [Accessing User Memory], page 27). All system calls need to read user memory. Few system calls need to write to user memory.
2. System call infrastructure (see Section 3.3.4 [System Calls], page 29). Implement enough code to read the system call number from the user stack and dispatch a handler based on it.
3. The exit system call. Every user program that finishes in the normal way calls exit. Even a program that returns from main() calls exit indirectly (see \_start() in ‘lib/user/entry.c’).
4. The write system call for writing to fd 1, the system console. All of our test programs write to the console (the user process version of printf() is implemented this way), so they will all malfunction until write is available.
5. For now, change process\_wait() to an infinite loop (one that waits forever). The provided implementation returns immediately, so Pintos will power off before any processes actually get to run. You will eventually need to provide a correct implementation.

After the above are implemented, user process should work minimally. At the very least, they can write to the console and exit correctly. You can then refine your implementation so that some of the tests start to pass.

**KIND OF A TO-DO LIST:**

* Some, but not all, solutions to project 2 require modifying page\_fault() in exception.h.
* You will want to look over ‘filesys.h’ and ‘file.h’ interfaces to understand how to use the file system, and especially its many limitations.
* You need to be able to create a simulated disk with a file system partition. Description of how to do so is provided on page 24.
* You might want to create a clean reference file system disk and copy that over whenever you trash your ‘filesys.dsk’ beyond a useful state, which may happen occasionally while debugging.
* Regarding argument passing, if you are completely lost (which is reasonable) look at strtok\_r(), prototyped in ‘lib/string.h’ and implemented with thorough comments in ‘lib/string.c’. You can find more about it by looking at the man page (run man strtok at the prompt)
* Implement the following system calls: (only after providing ways to read and write data in user virtual address space)
  + Halt
  + Exit
  + Exec
  + Wait
  + Create
  + Remove
  + Open
  + Filesize
  + Read
  + Write
  + Seek
  + Tell
  + Close
* You must synchronize system calls so that any number of user processes can make them at once. In particular, it is not safe to call into the file system code provided in the ‘filesys’ directory from multiple threads at once. Your system call implementation must treat the file system code as a critical section. Don’t forget that process\_execute() also accesses files.
* Add code to deny writes to files in use as executables. You can use file\_deny\_write() to prevent writes to an open file. Calling file\_allow\_write() on the file will re-enable them (unless the file is denied writes by another opener). Closing a file also re-enables writes. Thus, to deny writes to a process’s executable, you must keep it open as long as the process is still running.

**KIND OF A DO-NOT LIST:**

* Modify the file system code for this project
* Use ‘malloc’

**Further Details**

* Global Descriptor Table (GDT) is a tale that describes the segments in use.
* Task-State Segment (TSS) is used for 80x86 architectural task switching.
* ELF is a file format used by Linux, Solaris, and many other operating systems for object files, shared libraries, and executables.
* Until you copy a test program to the simulated file system, Pintos will be unable to do useful work. You won’t be able to do interesting things until you copy a variety of programs to the file system.
* Virtual memory in Pintos is divided into two regions:
  + User virtual memory
  + Kernel virtual memory
* User virtual memory ranges from virtual address 0 up to PHYS\_BASE, which is defined in ‘threads/vaddr.h’ and defaults to 0xc0000000 (3 GB).
* Kernel virtual memory occupies the rest of the virtual address space, from PHYS\_BASE up to 4GB.
* User virtual memory is per-process whereas kernel virtual memory is global.
* A user program can only access its own user virtual memory.
* An attempt to access virtual memory causes a **page fault**.
* Code segment in Pintos starts at user virtual address 0x08048000
* Linker sets the layout of a user program in memory
* In argument passing, first word is process and following words are arguments
* System Call Handler will need to retrieve:
  + System call number
  + Any system call arguments
  + Carry out appropriate actions
* The sole way a user program should be able to cause the OS to halt is by invoking the halt system call.
* If a system call is passed an invalid argument, acceptable options include returning an error value (for those calls that return a value), returning an undefined value, or terminating the process.
* What is the difference between tid\_t and pid\_t?
  + A tid\_t identifies a kernel thread, which may have a user process running in it (if created with process\_execute()) or not (if created with thread\_create()). It is a data type used only in the kernel.
  + A pid\_t identifies a user process. It is used by user processes and the kernel in the exec and wait system calls.
  + You can choose whatever suitable types you like for tid\_t and pid\_t. By default, they’re both int. You can make them a one-to-one mapping, so that the same values in both identify the same process, or you can use a more complex mapping, it’s up to you.
* Consider how to handle arguments for the following example command: ‘/bin/ls –l foo bar’. First, break the command into words:
  + ‘bin/ls’
  + ‘-l’
  + ‘foo’
  + ‘bar’

Place the words at the top of the stack. Order doesn’t matter, because they will be referenced through pointers.

Then push the address of each string plus a null pointer sentinel, on the stack, in right-to-left order. These are the elements of argv. The null pointer sentinel ensures that argv[argc] is a null pointer, as required by the C standard. The order ensures that argv[0] is at the lowest virtual address.

Then push argv (the address of argv[0]) and argc, in that order.

Finally, push a fake “return address”: although the entry function will never return, its stack frame must have the same structure as any other.

* Word-aligned accesses are faster than unaligned accesses, so for best performance round the stack pointer down to a multiple of 4 before the first push.
* In the 80x86 architecture, the ‘int’ instruction is the most commonly used means for invoking system calls.
* In pintos, user programs invoke ‘int $0x30’ to make a system call.
* When the system call handler syscall\_handler() gets control, the system call number is in the 32-bit word at the caller’s stack pointer, the first argument is in the 32-bit word at the next higher address, and so on.
* You should try to avoid writing large amounts of repetitive code for implementing system calls. Each system call argument, whether an integer or pointer, takes up 4 bytes on the stack. You should be able to take advantage of this to avoid writing much near-identical code for retrieving each system call’s argument from the stack.

**Fucking Around W/ The File System:**

**How to create a simulated disk with a file system partition:**

Inside the ‘userprog/build’ directory

**pintos-mkdisk filesys.dsk --filesys-size=2**

This creates a ‘filesys.dsk’ that contains a 2MB Pinto file system partition

**pintos -f** option causes the file system to be formatted

**pintos -q** option causes Pintos to exit as soon as the format is done

**pintos -p** option is for “put”

**pintos -g** option is for “get”

To copy ‘file’ into the Pintos file system, use the command

**pintos -p *file* -a *newname* -- -q**

To view the layout of a particular executable, run objdump (80x86) or i386-elf-objdump (SPARC) with the ‘-p’ option.

Here is a summary of how to create a disk with a file system partition, format the file system, copy the echo program into the new disk, and then run echo, passing argument x.

**pintos-msdisk filesys.dsk –filesys-size=2**

**pintos -p ../../examples/echo -a echo -- -f -q run ‘echo x’**

How to delete a file from the Pintos file system using the rm *file* kernel action

**pintos –q rm *file***

**Issues to Fix:**

**ISSUE 1:**

The kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so considering that invalid pointers can be passed.

Invalid Pointers:

* A null pointer
* A pointer to unmapped virtual memory
* A pointer to kernel virtual address space (above PHYS\_BASE)

All of these pointers must be rejected without harm to the kernel or other running processes, by terminating the offending process and freeing its resources.

Method 1 (Simpler):

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.

Method 2 (Faster):

Check that only a user pointer points below PHYS\_BASE, then derefence 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). However, it’s more difficult to handle if an invalid pointer causes a page fault, because there’s no way to return an error code from a memory access. Code to compensate for this issue is available on page 27.

In any case, you need to make sure not to “leak” resources.

**ISSUE 2:**

Whenever a user process terminates, because it called **exit** or for any other reason, print the process’s name and exit code, formatted as if printed by printf (“%s: exit (%d)\n”,…);. The name printed should be the full name passed to process\_execture(), omitting command-line arguments.

The issue here is how does one get the information for what the passed process is, but will look back at this later, still trying to get the scope of this whole project here.

In any case, do not print these messages when a kernel thread that is not a user process terminates, or when they halt system call is invoked.

^^^ Shitty pseudocode that doesn’t work but gives you the idea

if(! Kernel thread that is not a user process || halt system call is invoked){

Printf(“%s: exit (%d)\n”,..);

}

**Regarding 80x86 Calling Convention (I am still figuring this out)**

The calling convention works like this:

1. The caller pushes each of the function’s arguments on the stack one by one, normally using the PUSH assembly language instruction. Arguments are pushed in right-to-left order.

The stack grows **downward**: each push decrements the stack pointer, then stores into the location it now points to, like C expression ‘\*--sp = value’.

1. The caller pushes the address of its next instruction (the return address) on the stack and jumps to the first instruction of the callee. A single 80x86 instruction, CALL, does both.
2. The callee executes. When it takes control, the stack pointer points to the return address, the first argument is just above it, second argument is just above the first argument, and so on.
3. If the callee has a return value, it stores it into register **EAX**.
4. The callee returns by popping the return address from the stack and jumping to the location it specifies, using the **RET** instruction.
5. The caller pops the arguments off the stack.

Consider a function f() that takes three int arguments. This diagram shows a sample stack frame as seen by the callee at the beginning of step 3 above, supposing that f() is invoked as f(1, 2, 3). The initial stack address is arbitrary:

+----------------+

0xbffffe7c | 3 |

0xbffffe78 | 2 |

0xbffffe74 | 1 |

Stack pointer 🡪0xbffffe70 | return addr |

+----------------+

**Regarding the state of the stack and the relevant registers right before the beginning of the user program, assuming PHYS\_BASE is 0xc0000000:**

Address Name Data Type

0xbffffffc argv[3][…] ‘bar\0’ char[4]

0xbffffff8 argv[2][…] ‘foo\0’ char[4]

0xbffffff5 argv[1][…] ‘-l\0’ char[3]

0xbfffffed argv[0][…] ‘/bin/ls\0’ char[8]

0xbfffffec word-align 0 uint8\_t

0xbfffffe8 argv[4] 0 char \*

0xbfffffe4 argv[3] 0xbffffffc char \*

0xbfffffe0 argv[2] 0xbffffff8 char \*

0xbfffffdc argv[1] 0xbffffff5 char \*

0xbfffffd8 argv[0] 0xbffffffd char \*

0xbfffffd4 argv 0xbfffffd8 char \*\*

0xbfffffd0 argc 4 int

0xbfffffcc return address 0 void (\*) ()

In this example, the stack pointer would be initialized to 0xbfffffcc.

As shown above, your code should start the stack at the very top of the user virtual address space, in the page just below virtual address PHYS\_BASE (defined in ‘threads/vaddr.h’)

You may find the nonstandard **hex\_dump()** function, declared in ‘<stdio.h>’, useful for debugging your argument passing code.