**Pintos Project 2 Guideline**

1. **Score**

**Design Report: 10%**

Design report must include:

* Analysis on current Pintos system(50%)
* Process execution procedure
* System call procedure
* File system
* Solutions for each requirements (50%)
* Describe data structures and functions to be added or modified.

**Quiz: 20%**

We will ask simple questions on stack, system call, pintos kernel and any other topics related to implementing user program. You won’t have any problem if you have implemented Pintos User Program successfully.

**Demo: 10%**

We will ask questions to check if you have implemented *user program* properly. You should be expected to answer questions on how you had implemented each requirements for this project.

**Final Report: 10%**

Final report should include:

* + - How to achieve each requirements. (90%)
* Explain overall scheme, added/modified data structures/functions, and algorithm briefly.
  + - Discussion (10%)
* What have been changed with respect to the design report, what you have learned, etc.

**Code: 50%**

The most important part of Pintos project: running code that does not fail. DO NOT try to share or copy other’s work. And make sure your code does compiles, or else you will get no points.

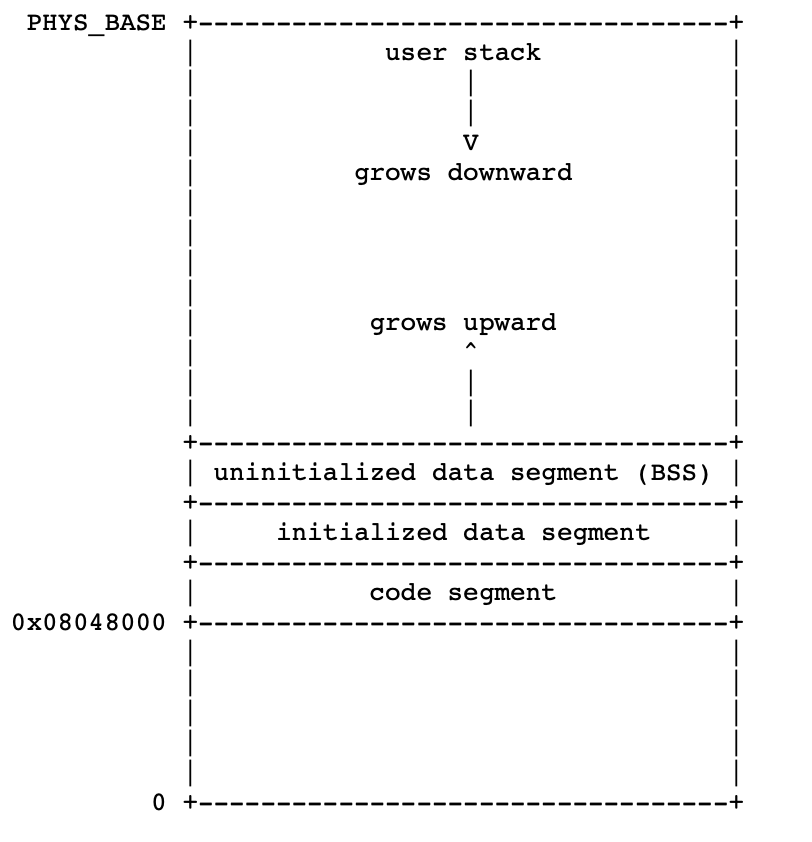
* Process Termination Messages: 5%
* Argument Passing: 15%
* System Call: 20%
* Denying Writes to Executables: 10%

1. **Backgrounds**

System Calls:

Special functions supported by the OS kernel which user programs can call to do special jobs that requires the authorization of the kernel. This includes things such as opening and writing a file, creating a memory space and etc.

Virtual Memory:

Virtual memory is a memory management technique that provides an “idealized abstraction of storage resources that are actually available on a given machine”. The pintos memory layout is as below. 

File system:

File system controls how data is stored and retrieved in the OS.

File Descriptor:

Is an abstract indicator or handler used to access a file or any other input/output resources. A file descriptor is a non-negative integer and normally 0, 1, 2 are reserved for standard input, standard output, and standard error, respectively.

1. **Code configuration**

Because user programs are loaded from the file system, you will have to modify the file system code for this project. And in order to use the file system, you need to create a simulated disk with a file system partition.

From the “userprog/build” directory, execute *pintos-mkdirk filesys.dsk --filesys-size=2*

This command creates a simulated disk named “filesys.dsk” with 2MB pintos file system partition. Then you need to format the file system partition by passing “-f -q” on the kernel’s command line: *pintos -f -q.* “-f”option formats the file system and “-q” causes Pintos to exit as soon as the format is done.

Once the partition is created, you need to copy files in and out of the simulated file system.

Pintos “-p” and “-g” are put and get options that you will use. To copy file into the file system, use the command “pintos -p [file] -- -q” Or you can add the file with a new name by “pintos -p [file] -a [new name] -- -q”

Below is an example of creating 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-mkdisk filesys.dsk --filesys-size=2*

*pintos -f -q*

*pintos --p ../../examples/echo -a echo -- -q*

*pintos -q run ‘echo x’*

If you don't want to keep the file system disk around for later use, you can run above four steps into a single command.

*pintos --filesys-size=2 -p ../../examples/echo -a echo -- -f -q run ‘echo x’*

1. **Requirements**
2. **Process Termination Messages**

Print the process’s name and exit code by print(“%s: exit(%d)\n”, process\_name, exit\_code). Do not print these messages when a kernel thread terminates or the halt system call is invoked. Don’t print any other additional messages

1. **Argument Passing**

Add codes to support argument passing for the function process\_execute(). Instead of taking a program file name as its only argument, process\_execute() should split words using spaces and identify program name and its arguments separately. For instance, process\_execute(“grep foo bar”) shoud run grep with two arguments foo and bar.

1. **System Call**

Implement the system call handler in “userprog/syscall.c”. Current handler will terminate the process if system call is called. It should retrieve the system call number and system call arguments after the system call is handled.

Implement the following system calls:

1. void halt(void)

Terminates Pintos by calling shutdown\_power\_off()

1. void exit(int status)

Terminates the current user program and return status to the kernel. Ther kernel passes the status to the parent process.

1. pid\_t exec(const char \*cmd\_line)

Runs the program whose name is given in cmd\_line and returns the new process’s pid. If its fails to execute the new program, it should return -1 as pid. Synchronization should be ensured for this system call.

1. int wait (pid\_t pid)

Waits for the child process given as pid to terminate and retrieves the exit status. It is possible for the parent process to wait for a child process that has already been terminated. The kernel should retrieve the child’s exit status and pass it to the parent anyway. If the child process has been terminated by the kernel, return status must be -1.

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. That is, if A spawns child B and B spawns child C, A cannot wait for C.
    - The process already called wait for the pid in the past. That is, the process can wait for a pid only once.

Processes may spawn any number of children. Your design should consider all the possible situation that can happen between parent and the child process.

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

1. bool create(const char \*file, unsigned initial\_size)

Creates a new file with the name file and initialize its size with initial\_size. Return true if successful, false otherwise.

1. bool remove(const char \*file)

Deletes the file called file. Returns true if successful, false otherwise. A file may be removed whether it is open or closed. However, removing an open file does not close it.

1. int open(const char \*file)

open the file with name file. Returns a nonnegative integer number for the file descriptor, or -1 if unsuccessful.

File descriptors number 0 and 1 are reserved for the console, STDIN\_FILENO, STDOU”T\_FILENO, respectively. These numbers should not be used.

Each process has an independent set of file descriptors and file descriptors are not inherited to child processes.

1. int filesize(int fd)

Returns the size of opened file with fd.

1. int read(int fd, void \*buffer, unsigned size)

Reads size bytes from the opend file and save the contents into buffer. Returns the number of bytes that are acutally read. -1 should be returned if the system fails to read. If 0 is given as fd, it should read from the keyboard using input\_getc()

1. 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.

Since the basic file system for this project does not support file growth, you should not write past the end-of-file.

If 1 is given as the fd, it should write to the console. You should use putbuf() to write things in the buffer to the console.

1. void seek (int fd, unsigned position)

Changes the next bytes to be read or written in open file fd to position.

1. unsigned tell (int fd)

Returns the position of the next byte to be read or written in open file fd.

1. void close (int fd)

Closes the file with fd. You should also close all of its open file descriptors as well.

1. **Denying Writes to Executables**

Add code to deny writes to files in use as executable. You can use file\_deny\_write() to prevent writes to an open file. Calling file\_allow\_write() or closing a file will re-enable writes to the file.

**4. Supplements**

**Tips:**

You must synchronize system calls so that any number of user processes can make them 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.

Once you implement system calls, there should be no circumstances where a user program can cause OS to crash, panic, fail an assertion, or otherwise it’s a malfunction.

**Suggested order of implementation:**

1. Argument passing
2. User memory access
3. Process terminate messages
4. System call
5. Denying writes to executables

**5. References**

Ben Pfaff, et al. “Pintos”. 2009