Project2. User Programs

Score

Total: 35 points

Requirements

1. User Process (10 points)

A. Argument Passing (5 points)

- Implement "argument passing" by extending ***process_execute()*** so that it divides the command line into words at spaces, instead of simply taking a program file name as its argument.
- The first word in command line is the program name, the second word is the first argument, and so on.
- Within a command line, each word is separated by one or more spaces.

  Example) args-dbl-space two   spaces

  In the example above, program name is args-dbl-space, the first argument is two, and the second argument is spaces.

B. Process Termination Messages (5 points)

- Whenever a user process terminates, print the process's name and exit code, formatted as if printed by ***printf ("%s: exit(%d)₩n", ...);***.
- The printed name should be the full name passed to *process_execute()*, omitting command line arguments.
- Do not print the messages when a kernel thread that is not a user process terminates, or when the *halt()* system call is invoked.

2. System Calls (20 points)

- Implement the following system calls.
- The prototypes listed are those seen by a user program that includes '*lib/user/syscall.h*'
- System call numbers for each system call are defined in '*lib/syscall-nr.h*'
- Modify the source code in **syscall.c** and **syscall.h** in **userprog** directory.
- Source codes which are related to the file system are located in **filesys** directory. In this project, use those codes without modification.
- You can use functions declared in **filesys.h** and **file.h**.

A. User Process Manipulation (10 points)

- void **halt** (void)

  : Terminates Pintos by calling *shutdown_power_off()* (declared in '*devices/shutdown.h*').

- void **exit** (int *status*)
  : Terminates the current user program, returning *status* to the kernel.

- 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, if the program cannot load or run for any reason.

- int **wait** (pid_t *pid*)
  : Waits for a process which has the same Program ID as *pid,* and returns the target process's exit status. If the target process did not call *exit()*, but was terminated by the kernel (e.g. killed due to an exception), *wait* must return **-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. *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 process B and B spawns child process C, then A cannot wait for C, even if B is dead.
  - The process that calls *wait* has already called *wait* on *pid*.

B. File Manipulation (10 points)

- 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.

- bool **remove** (const char **file*)
  : Deletes the file called *file*. Returns *true* if successful, *false* otherwise.

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

- 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. The expected behavior is to write as many bytes as possible up to end-of-file and return the actual number written (0 at end of file). *fd* 1 writes to the console using *putbuf()*.

- 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.)

- 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.

3. Denying writes to executables (5 points)

- 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.
- Closing a file will also re-enable writes.

Supplements

1. Using the File System

You will need to interface to the file system code for this project, because user programs are loaded from the file system and many of the system calls you must implement deal with the file system. Therefore, a simple but complete file system is provided in the 'filesys' directory. Functions are declared in 'filesys.h' and 'file.h'. You can use it without modification. But there are some limitations you should look over.

- No internal synchronization. Concurrent accesses will interfere with one another. You should use synchronization to ensure that only one process at a time is executing file system code.
- File size is fixed at creation time. The root directory is represented as a file, so the number of files that may be created is also limited.
- File data is allocated as a single extent, that is, data in a single file must occupy a contiguous range of sectors on disk.
- No subdirectories.
- File names are limited to 14 characters.
- A system crash mid-operation may corrupt the disk in a way that cannot be repaired automatically.
- If a file is open when it is removed by filesys_remove(), its blocks are not deallocated and it may still be accessed by any threads that have it open, until the last one closes it.

You can create a simulated disk with a file system partition by **mkdisk** program. From the '**userprog/build**' directory, execute **pintos-mkdisk filesys.dsk --filesys-size=2**. This command creates a simulated disk named '**filesys.dsk**' that contains a 2 MB Pintos file system partition. Then format the file system partition by passing '**-f -q**' on the kernel's command line: **pintos -f -q**. The '**-f**' option causes the file system to be formatted, and '**-q**' causes Pintos to exit as soon as the format is done.

The pintos '**-p**' ("put") and '**-g**' ("get") options can copy files in and out of the simulated file system. To copy 'file' into the Pintos file system, use the command '**pintos -p file -- -q**'. (The '--' is needed because '-p' is for the pintos script, not for the simulated kernel.) To copy it to the Pintos file system under the name 'newname', add '**-a newname**': '**pintos -p file -a newname -- -q**'. The commands for copying files out of a VM are similar, but substitute '**-g**' for '-p'. Incidentally, these commands work by passing special commands extract and append on the kernel's command line and copying to and from a special simulated "scratch" partition. If you're very curious, you can look at the pintos script as well as '**filesys/fsutil.c**' to learn the implementation details.

Here's 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. (Argument passing won't work until you implemented it.) It assumes that you've already built the examples in 'examples' and that the current directory is 'userprog/build':

```
pintos-mkdisk filesys.dsk --filesys-size=2
pintos -f -q
pintos -p ../../examples/echo -a echo -- -q
pintos -q run 'echo x'
```

You can delete a file from Pintos file system using **rm *file*** kernel action, e.g. **pintos –q rm *file***. Also, **ls** lists the files in the file system and **cat *file*** prints a file's content to the display.

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. Some of the example programs will only work once projects 3 or 4 have been implemented.

2. System Call

The first project already dealt with one way that the operating system can regain control from a user program: interrupts from timers and I/O devices. These are "external" interrupts. The operating system also deals with software exceptions, which are events that occur in program code. These can be errors such as a page fault or division by zero. Exceptions are also the means by which a user program can request services ("system calls") from the operating system.

In Pintos, user programs invoke 'int $0x30' to make a system call. The system call number and any additional arguments are expected to be pushed on the stack in the normal fashion before invoking the interrupt. This instruction is handled in the same way as other software exceptions. (see Section 3.5 [80x86 Calling Convention], page 35).

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. The caller's stack pointer is accessible to **syscall_handler()** as the '**esp**' member of the **struct intr_frame** passed to it. (**struct intr_frame** is on the kernel stack.) The 80x86 convention for function return values is to place them in the EAX register. System calls that return a value can do so by modifying the '**eax**' member of **struct intr_frame**.

3. Virtual Memory Layout

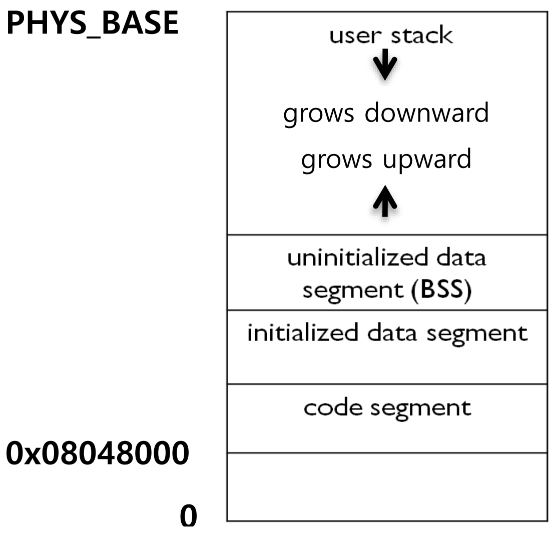
Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address 0 up to **PHYS_BASE**, which is defined in '**threads/vaddr.h**' and defaults to 0xc0000000 (3 GB). Kernel virtual memory occupies the rest of the virtual address space, from PHYS_BASE up to 4 GB.

User virtual memory is per-process. When the kernel switches from one process to another, it also switches user virtual address spaces by changing the processor's page directory base register (see *pagedir_activate()* in '**userprog/pagedir.c**'). *struct thread* contains a pointer to a process's page table.

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.

Conceptually, each process is free to lay out its own user virtual memory however it chooses. In practice, user virtual memory is laid out like this:



The code segment in Pintos starts at user virtual address 0x08084000, approximately 128 MB from the bottom of the address space.

4. Program Startup Details

The kernel must put the arguments for the initial function on the stack before it allows the user program to begin executing. The arguments are passed in the same way as the normal calling convention (see Section 3.5 [80x86 Calling Convention], page 35 in Pintos documentation). Consider how to handle arguments for the following example command: '**/bin/ls –l foo bar**'. The table below shows the state of the stack, assuming PHYS_BASE is 0xc0000000. The stack pointer would be initialized to **0xbfffffcc**

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]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()