Pintos

27th July 2020

Course

BSc (HONS) Computer Science

Module

Operating Systems UFCFWK-15-2

Lecturer

Ramkumar Krishnamoorthy

Institute

Villa College

Authors

Mohamed Isaam Rameez - S1900343

Mariyam Yasmeen - S1800367

Contents

What is PintOS

Developed originally at Stanford University by Ben Pfaff, PintOS is a project aimed at allowing students to build core functions upon the provided base code so as to introduce them to the process of operating system development (Chao, 2017). This simple OS can handle kernel threads, writing to and reading from I/O devices and also loading and executing programs. Kernel-level projects provide the perfect opportunity for students to learn about how OS design and development take place in the real world (Laadan et. al, 2010). The code base, on which students build upon, is well commented so as to help students understand how it works and what needs to be done in order to reach their assignment goals.

Introduction

Students were provided with the base code for University of West England's version of PintOS along with instructions. This assignment is the final submission required for the Operating System module for year two. The first task assigned was to implement argument passing and the second task was to implement a number of system calls using C. This document outlines the code that was implemented in order to achieve these goals and includes the results that we got from testing the completed system.

Argument Passing

The base code starts off without being able to handle any arguments. Doing a simple change to ~/Pintos/userprog/process.c can help us run pintos -q run 'echo x' before we start implementing the actual code where we build a stack that is capable of handling arguments.

```
/* Create a minimal stack by mapping a zeroed page at the top of user virtual memory. */
static bool setup_stack (void **esp)
{
  uint8_t *kpage;
  bool success = false;

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success) {
        *esp = PHYS_BASE - 12;
      } else
        palloc_free_page (kpage);
    }
  return success;
}
```

In order to allow PintOS to handle arguments, we add command line arguments to the stack. As it is, when we run echo x, it is not able to identify that x is an argument. Therefore, we have made changes so that we can iterate over the line and store the values when we come across either \0 or spaces. The length is calculated by checking the difference from the end and start and we then add the argument to the stack using strlcpy

```
/* Adding CL args onto the stack */
bool stack_build (void **esp, const char *args){
  void *esp_tmp = *esp;
  size_t length, align;
  char *crnt = ((char *)args)+strlen(args);
  char *start, *end;

  while (crnt >= args){
      // Looks for the end of the argument
      while (crnt >= args && (*crnt == ' ' || *crnt == '\0'))
        crnt--;

      end = crnt + 1;

      // Looks up the start of the current argument
      while (crnt >= args && *crnt != ' ')
        crnt--;

      start = crnt + 1;
      // In order to make room for \0 we should add 1
      length = (end - start) + 1;

      if (!esp_move (&esp_tmp, length))
        return false;

      strlcpy (esp_tmp, start, length);
    }

  // Memory gets rounded down and stack pointer gets updated
  align = ((uint32_t) esp_tmp) % 4;

  if (!esp_move (&esp_tmp, align))
    return false;

  memset(esp_tmp, 0, align);
  *esp = esp_tmp;

  return true;
}
```

Additionally, we pad the stack with zeros, a NULL pointer is added so that we can set argv[argc] as NULL, pointers towards the beginning of an argument is added as well as the pointer to the initial argument, the signed integer value of the number of arguments and a return address that is NULL is also added to the stack.

The changes made to the code enable us to use the start of the command (in our case echo) as the thread name that we use in process_execute. We iterate over the whole argument and write them into an array up to null or spaces and then the new character array gets a null character added in. We also make sure that a stack overflow is detected by ensuring that the stack pointer is below PHYS_BASE. This is so that the stack pointer doesn't go below 0x00000000 and wrap.

Testing

We tested whether argument passing works using the following steps:

1. Head to ~/Pintos/userprog/build
2. Create a filesys.dsk file using `pintos-mkdisk filesys.dsk --filesys-size=2`
3. Format it using `pintos -f -q`
4. Copy the relevant file over from examples using `pintos -p ../../examples/echo -a echo -- -q`
5. Run the argument `pintos -q run 'echo x'`

```
Loading.....
Kernel command line: -q run 'echo x'
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  120,627,200 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 194 sectors (97 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 'echo x':
echo x
echo: exit(0)
Execution of 'echo x' complete.
Timer: 75 ticks
Thread: 0 idle ticks, 72 kernel ticks, 3 user ticks
hdb1 (filesys): 33 reads, 0 writes
Console: 635 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
jaze@jaze-VirtualBox:~/Pintos/userprog/build$
```

System Calls

In order to implement system calls we used parts of or made changes to the code in ~/Pintos/threads/thread.h, ~/Pintos/userprog/process.h and ~/Pintos/userprog/syscall.c.

Breakdown of Utilisations & Changes

~/Pintos/threads/thread.h

```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    int exit_status;                    /* Exit status of the thread */
    bool is_user;                       /* If it's a user program thread */
    struct thread *parent;              /* Parent thread */
    struct list children;               /* Information for child process*/
    struct list files;                  /* Files for thread */
    struct file *exec;                  /* Executed file */
#endif
```

We had utilised these items in the following way:

int exit_status

When `process_exit` is called, we need the thread's exit status to be sent to `exit_status` in process. This variable is set to -1 by default because we need to identify when the kernel kills a thread due to an exception. If the thread is not killed, the process will update the status to this value.

bool is_user

This indicates whether the thread is for a user process or whether it is a kernel process instead. We need this in order to detect whether the thread and its parent need to communicate.

struct thread *parent

This is a pointer towards the parent of the thread. This value is NULL for kernel threads.

struct list children

This is a list of elem members of the process structure. When this is used along with the parent, we can let a child thread send updates to its parent. An example of this would be in the situation where we need the child thread to send its exit status to its parent.

struct list files

This is a list of elem members of a `file_map` structure and is used to access and track the open files that are a part of the thread.

struct file *exec

The file is locked to prevent any changes from being made to the file after it is loaded by the load function. The file can be changed when it gets unlocked during `process_exit` function.

~/Pintos/userprog/Process.h

```
// File states
enum load_status{
    NOT_LOADED,         /* Initial state */
    LOAD_SUCCESS,       /* Loaded successfuly */
    LOAD_FAILED         /* Did not load */
  };

/* Child process info struct */
struct process{
    struct list_elem elem;        /* Child process list */
    pid_t pid;                    /* Process thread identity */
    bool is_alive;                /* Process start status */
    bool is_waited;               /* Process wait status */
    int exit_status;              /* Process exit status */
    enum load_status load_status; /* Load status of file being executed */
    struct semaphore wait;        /* Wait for process to exit, then return state */
    struct semaphore load;        /* Wait for file to load or fail */
  };
```

struct process is used so that data regarding a thread has to be available even when the program ends. We have used these items in the following way:

struct list_elem elem

This is a list element that would be a children list member in a thread. This is needed so that even if a child thread ends and its structure is deallocated, the parent can still access changes.

pit_t pid

Process identity

bool is_alive

Denotes whether the child thread is alive and is used to make a check before the child thread is exited. If a child thread has not exited then we can wait until we get the exit status of the thread.

bool is_waited

This is used to check if the parent called wait on the child.

int exit_status

This is the exit status of the child which is set in `process_exit`.

enum load_status load_status

Indicates whether the file executed by the process loaded successfully. It will set to either `LOAD_SUCCESS` or `LOAD_FAILURE`.

struct semaphore wait

Semaphore used by `process_wait`. The child will increment wait when its exit status is known so that the parent can continue

struct semaphore load

Semaphore used by `process_execute` to wait on the file to load for the child process. Once successful, `load_status` is updated so that the parent can continue.

~/Pintos/userprog/syscall.c

```
/* Maps file structures and descriptors */
struct file_map{
    struct list_elem elem;      /* List element*/
    int fd;                     /* File descriptor. */
    struct file *file;          /* File structure. */
};
```

The struct `file_map` is utilised the following way:

struct list_elem elem

This is a list element which would be added to the files list member of a thread. It's needed to link any open files to a thread so that the thread would be able to close them when exiting.

int fd

File descriptor for a file. This lets user programs operate on a file without access to the file structure.

struct file *file

This is the pointer to the file structure representing open files. Any file-based system calls would be operating on this pointer.

Handling system calls

Whenever a system call is called, the kernel would call `syscall_handler`. We have implemented a switch case so that these calls are handled correctly and if a value is returned, this would be saved in the EAX register. If a call beyond the implemented cases is called, the process will exit.

```
  switch (call){
    case SYS_HALT:
      sys_halt();
      break;

    case SYS_EXIT:
      sys_exit ((int) load_stack (f, ARG_0));
      break;

    case SYS_EXEC:
      f->eax = sys_exec ((const char *) load_stack (f, ARG_0));
      break;

    case SYS_WAIT:
      f->eax = sys_wait ((pid_t) load_stack (f, ARG_0));
      break;

    case SYS_CREATE:
      f->eax = sys_create ((const char *) load_stack (f, ARG_0),
        (unsigned int) load_stack (f, ARG_1));
      break;

    case SYS_REMOVE:
      f->eax = sys_remove ((const char *) load_stack (f, ARG_0));
      break;

    case SYS_OPEN:
      f->eax = sys_open ((const char *) load_stack (f, ARG_0));
      break;

    case SYS_FILESIZE:
      f->eax = sys_filesize ((int) load_stack (f, ARG_0));
      break;

    case SYS_READ:
      f->eax = sys_read ((int) load_stack (f, ARG_0),
          (void *) load_stack (f, ARG_1),
          (unsigned int) load_stack (f, ARG_2));
      break;

    case SYS_WRITE:
      f->eax = sys_write ((int) load_stack (f, ARG_0),
          (const void *) load_stack (f, ARG_1),
          (unsigned int) load_stack (f, ARG_2));
      break;

    case SYS_SEEK:
      sys_seek ((int) load_stack (f, ARG_0),
        (unsigned) load_stack (f, ARG_1));
      break;

    case SYS_TELL:
      f->eax = sys_tell ((int) load_stack (f, ARG_0));
      break;

    case SYS_CLOSE:
      sys_close ((int) load_stack (f, ARG_0));
      break;

  // For all unknown system calls
    default:
      sys_exit (EXIT_ERROR);
      break;
    }
}
```

Handling Files

We use semaphores in order to make file handling much safer in situations where multiple threads may be simultaneously calling functions inside the fileys directory. It is set to 1 and once a system call is operating on the file structure it is decremented. Once completed it is incremented once again. In this way, when we run calls such as read, write, create or remove we can allow threads on hold to complete their operations without messing up files.

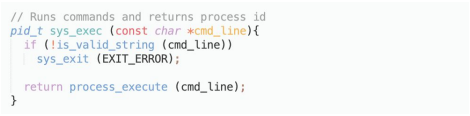
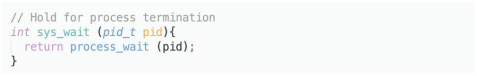
Validating Strings / Buffers & System Calls

If the length of a string is unknown, we use the function `get_user` over every byte till we reach \0 and if the length is known the same function is called to go over all bytes from 0 to string length. In the case where a buffer or the string is a system call argument, the validation is carried out by its handler. As for pointers, their validation into user memory is carried out using the functions provided by Stanford. The changes made to it include ensuring that the `PHYS_BASE` is above the address pointer argument. Additional changes were also made to exception.c found in the userprog directory.

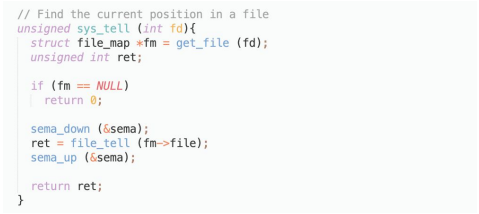
System Call	Function	Description

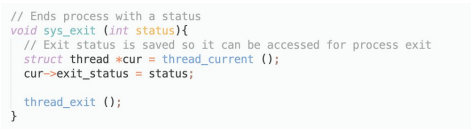
SYS_HALT	`// Exit pintos`
	`void sys_halt (void){`
	`  shutdown_power_off ();`
	`}`
		Exits by calling `shutdown_power_off`.

SYS_EXIT	`// Ends process with a status`
	`void sys_exit (int status){`
	`  // Exit status is saved so it can be accessed for process exit`
	`  struct thread *cur = thread_current ();`
	`  cur->exit_status = status;`
	`  thread_exit ();`
	`}`
		Exit status is stored and `thread_exit` is called which then in turn calls `process_exit` (which cleans up after thread exit by closing files and deallocating file map structures, removing and deallocating child processes and updating parents with the exit status).

SYS_EXEC		`sys_exec` performs a validity check and would return a -1 if the file is not executable. `process_execute` is used since exec needs to handle child processes and the user program.
SYS_WAIT		If a process was exited, `sys_wait` would return exit status. If not it checks on `is_waited` where if it's true -1 is returned. Else `is_waited` bool is set to true and state of `is_alive` is checked. If it's false then that means the child process has exited and the exit status is returned. If not, the process structure's wait semaphore is used to wait on the child process. `process_execute` is used since the main kernel threads can wait on user programs when needed.
SYS_CREATE		First we determine that the file name is valid and then implement `filesys_create` to create files.
SYS_REMOVE		First the name of the file passed is validated and then filesys_remove is used to remove the specific file.

SYS_OPEN		File name is validated, file descriptor is generated, `file_map` structure is set and then `filesys_open` is used to open the file. These values get added to the files list of the thread after being added to the `file_map` structure.
SYS_FILESIZE		The file descriptor is utilised to look for the file structure in the list of files for the current thread.
SYS_READ		File reading is implemented using `file_read`. When using stdin, we utilise `input_getc` to read the characters one by one till the buffer's max length is reached. Buffer validation is done using `put_user` and we look for the file descriptor from the files list of the current thread which is how we retrieve the file structure. The process will exit if there is no match for the file descriptor or if the file descriptor is for stdout(1).

SYS_WRITE		File writing is implemented using `file_write`. The buffer is validated and since stdout is handled separately, `putbuf` is used when writing to stdout. We also make sure that large buffers get broken up. If we are writing to a file, the files list of the current thread is checked for the file descriptor so as to retrieve the file structure. The process will exit if there is no match for the file descriptor or if the file descriptor is for stdin(0).
SYS_SEEK		File seek is implemented with the use of `file_seek`. The files list of the current thread is checked for the file descriptor so as to retrieve the file structure.
SYS_TELL		File tell is implemented with the use of `file_tell`. The files list of the current thread is checked for the file descriptor so as to retrieve the file structure.

SYS_CLOSE		The files list of the current thread is checked for the file descriptor so as to retrieve the `file_map` structure. When found, the file member of this `file_map` structure is sent to the function `filesys_close`. The file_map structure then gets removed from the files list of the thread and is deallocated so as to prevent memory leaks.
Default		For all invalid or unknown system calls, we call `sys_exit` which ends the process.

Testing

We tested to see if all system calls are implemented correctly by heading to the following files and making sure that all of them are built using the command `make`

~/Pintos/examples
~/Pintos/filesys
~/Pintos/threads
~/Pintos/userprog
~/Pintos/utils
~/Pintos/vm

We then created a lorem.txt file inside ~/Pintos/examples, where the first line had our full name, second line had our student ID and the third had our email.

Now we are ready to run our tests by navigating to ~/Pintos/userprog/build and running `make check`. If you implemented PintOS correctly you should get a result of 76 tests passing. The inbuilt testing functions for PintOS are meant to produce deterministic output and the grading script lets students easily identify the points at which the actual output differs from the expected output. Some of these tests include testing the implemented code for functionality, robustness, memory mapping, stack growth and even stress tests for correct behavior on nested directories that are deep and disks that are almost full (Pfaff et. al, 2009).

```
TOTAL TESTING SCORE: 100.0%
ALL TESTED PASSED -- PERFECT SCORE

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                    Pts Max  % Ttl  % Max
------------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality         108/108  35.0%/ 35.0%
tests/userprog/Rubric.robustness             88/ 88  25.0%/ 25.0%
tests/userprog/no-vm/Rubric                   1/  1  10.0%/ 10.0%
tests/filesys/base/Rubric                    30/ 30  30.0%/ 30.0%
------------------------------------------- --- --- ------ ------
Total                                               100.0%/100.0%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Test results

Mohamed Isaam Rameez

```
pass tests/filesys/base/syn-write
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 76 tests passed.
ark@ubuntu:~/Pintos/userprog/build$
```

Mariyam Yasmeen

```
jaze@jaze-VirtualBox:~/Pintos/userprog/build$ make check
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 76 tests passed.
jaze@jaze-VirtualBox:~/Pintos/userprog/build$
```

Bibliography

1. Chao, L. (2017) Symmetric MultiProcessing for the Pintos Instructional Operating System [online]. MSc, Virginia Polytechnic Institute and State University. Available from: https://vtechworks.lib.vt.edu/handle/10919/78293 [Accessed 27 July 2020].

2. Laadan, O., Nieh, J. and Viennot, N. (2010) Teaching Operating Systems Using Virtual Appliances and Distributed Version Control. SIGCSE '10: Proceedings of the 41st Acm Technical Symposium on Computer Science Education [online]., pp. 480-484. [Accessed 28 July 2020].

3. Pfaff, B, Romano, A. and Back, G. (2009) The pintos instructional operating system kernel. SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education [online]., pp. 453-457. [Accessed 28 July 2020].