Pintos Design Document: User Programs

Matthew Dowdell (09011906) Jake Robertson (15016276)

Contents

1 Introduction

This document describes the implementation of user programs in Pintos. The first part the assignment is to implement argument passing, thus setting up the `argc` and `argv` arguments to a programs `main` function. The second part covers an implementation of a selection of system calls which allow child processes and file operations.

The repository storing the files and changes for this assignment can be found at https://gitlab.uwe.ac.uk/mattdowdell/pintos.

The files changed for the assignment are:

```
threads/synch.c      |  44 +++--
threads/thread.c     | 153 ++++++++++++----
threads/thread.h     |  22 ++-
userprog/exception.c |  23 ++-
userprog/process.c   | 456 ++++++++++++++++++++++++++++++++++++++++-----
userprog/process.h   |  27 ++-
userprog/syscall.c   | 522 ++++++++++++++++++++++++++++++++++++++++++++++++++-
userprog/syscall.h   |   3 +
8 files changed, 1130 insertions(+), 120 deletions(-)
```

See https://gitlab.uwe.ac.uk/mattdowdell/pintos/compare/5c7e71c4...master for more details.

2 Argument Passing

Argument passing requires setting up the stack ready for the main function of a C program to be called. In summary:

- The command line arguments are added to the stack. This is achieved by iterating over the arguments, character by character, starting at the end. When the end of the next argument is found, denoted by a space or `\0`, its position is stored. Once the start of the arguments string or a space is detected, the length of the argument is calculated based on the difference between the start and the end. To add the argument to the stack, `strlcpy` is called using the start position of the argument and it's length.
- The stack is padded with zeroes to the start of the next 4 byte section.
- A pointer to `NULL` is added to the stack, to ensure `argv[argc]` is `NULL` as required by the C specification.
- Pointers to the start of each argument are added to the stack from the last argument to the first.
- A pointer to the first argument pointer is added to the stack.
- The number of arguments, which is stored in a signed integer, is added to the stack.
- A return address is added to the stack. For the purposes of this assignment, the return address is `NULL`.

Taking command line arguments to be `"echo foo bar baz"`, the stack would end up as described in Table 1, assuming an original stack pointer value of `0xc0000000` (the default value of `PHYS_BASE`).

0xbffffffc	baz\0	char[4]	argv[3][...]
0xbffffff8	bar\0	char[4]	argv[2][...]
0xbffffff4	foo\0	char[4]	argv[1][...]
0xbfffffef	echo\0	char[5]	argv[0][...]
0xbfffffee	0	char	word-align
0xbfffffed	0	char	word-align
0xbfffffec	0	char	word-align
0xbfffffe8	NULL	char *	argv[4]
0xbfffffe4	0xbffffffc	char *	argv[3]
0xbfffffe0	0xbffffff8	char *	argv[2]
0xbfffffdc	0xbffffff4	char *	argv[1]
0xbfffffd8	0xbfffffef	char *	argv[0]
0xbfffffd4	0xbfffffd8	char **	**argv
0xbfffffd0	4	int	argc
0xbfffffcc	NULL	void *	return address

Table 1: An example user program stack.

The changes made under this part of the assignment are as follows:

- Use the first command line argument as the name of the thread in `process_execute`, rather than the entire command line argument string, for example `"echo"` rather than `"echo foo bar baz"`. This is achieved by iterating over the arguments and copying each character to a new char array until a space or null character is found, at which point a null character is appended to the new char array.
- Pass the first command line argument to `load` in `start_process` as opposed to the entire command line argument string using the same method detailed in the point above.
- Attempt to set up the stack for the given command line arguments in `load` at the last step before checking if `load` completed successfully. This operation may fail if a stack overflow is caused which is detected by ensuring that the stack pointer does not fall below `0x00000000` and causing it to wrap. This is detected by making sure that the stack pointer is always below `PHYS_BASE`. To ensure that the stack pointer cannot wrap to the extent that it end up between the previous location of the stack pointer and `0x00000000`, the size being added to the stack must be less than than `PHYS_BASE`.

3 System Calls

All requested system calls have been implemented, the details of which can be found in the sub-sections below. In implementing these system calls the following structures and enumerations were used/altered.

```
/* threads/thread.h */

struct thread
  {
    // ...
    #ifdef USERPROG
      // ...
      int exit_status;
      bool is_user;
      struct thread *parent;
      struct list children;
      struct list files;
      struct file *exec;
    #endif
    // ...
  };
```

The added members of the `thread` structure are used as follows:

- `exit_status`: Tracks the exit status of the thread which is passed to the `exit_status` member of the `process` structure when `process_exit` is called. By default, this is -1 to indicate the thread was killed by the kernel due to an exception. If the thread completes with no error, it will be updated to the status set by it's process.
- `is_user`: Denotes whether the thread is a thread for a kernel process or for a user process. This is used to differentiate between threads when checking whether the thread should communicate with its parent. Defaults to `NULL`.
- `parent`: A pointer to the threads parent. If the thread is a kernel thread, this will be `NULL`.
- `children`: A list containing the `elem` member of a `process` structure. Combined with `parent`, this is used to allow a child thread to update it's parent of changes when required, such as the its exit status.
- `files`: A list containing the `elem` member of a `file_map` structure. This list is used to track and access any open files associated with the thread.
- `exec`: To prevent the file being executed by a process being altered during execution, the file is locked to prevent writes after being loaded in `load`. To allow the file to be written to after the process has finished executing, a reference to it is stored and then unlocked in `process_exit` once the process has exited.

```
/* userprog/process.h */

enum load_status
  {
    NOT_LOADED,
    LOAD_SUCCESS;
    LOAD_FAILED;
  };

struct process
  {
    struct list_elem elem;
    pid_t pid;
    bool is_alive;
    bool is_waited;
    int exit_status;
    enum load_status load_status;
    struct semaphore wait;
    struct semaphore load;
  };
```

The `process` structure is used to store data about a thread for a user program that needs to persist beyond the end of the program. It is used as follows:

- `elem`: A list element to be added to a threads `children` list member. This allows a parent thread to access any changes to a child without being affected by the child thread exiting and its structure subsequently being de-allocated.

- pid: The id of the process. The same as the child threads tid member.
- is_alive: Whether the child thread is still alive. Used as an additional check before accessing the exit_status member to ensure if the child thread exited without errors it has updated the process structure in the children list of its parent. If the child process has not exited, then the wait member is used to block the parent thread until the exit status is known.
- is_waited: Whether the parent thread has already called wait for the child process. If this is true, wait will return -1 as required by the specification.
- exit_status: The exit status of the child thread, corresponding to the exit_status member of a thread structure. This is set during process_exit by copying the value from the current thread to the associated process structure in its parent.
- load_status: Denotes whether the file being executed by a process has loaded without issues or not and defaults to NOT_LOADED. If the file loaded without issues, it will be set to LOAD_SUCCESS. Otherwise, it will be set to LOAD_FAILURE.
- wait: This semaphore is used by process_wait when waiting for the child thread to exit if the child thread is still alive. When the exit status of the child thread is known, the child will increment wait to communicate that the parent can continue and return the now known exit status.
- load: This semaphore is used by process_execute to wait for a child process to load its file. When a child thread has loaded the file, it sets the load_status member of the process structure and allows the parent thread to continue.

A diagrammatic representation of how a process structure is linked to a thread structure is shown in Figure 1.

```
/* userprog/syscall.c */

struct file_map
  {
    struct list_elem elem;
    int fd;
    struct file *file;
  };
```

The file_map structure is used as follows:

- elem: A list element to be added to a threads files list member. This is used to associate an open file with a thread enabling the thread to close any leftover open files when exiting.
- fd: The file descriptor of the file. This effectively takes the role of an index and allows a user program to operate on a file without providing direct access to a files file structure.
- file: A pointer to the file structure representing an open file. This is the member that is operated upon by the various file-based system calls.

A file_map structure is linked to a thread structure in much the same way that a process structure is, see Figure 1, replacing the process structure for a file_map structure and using the files list in a thread structure rather than the children list.

3.1 System call handler

When a user process calls a system call function, control is passed to the kernel which calls syscall_handler. This function expects the stack to be set up as detailed in Table 2, where ARG_CODE is a code identifying the system call, and `ARG_0`, `ARG_1` and `ARG_2` are arguments to the system call, the number of which may be present depends on the system call. The system call code and arguments are validated using the techniques described in §Validation.

0x0000000c	0x12345990	void *	ARG_2
0x00000008	0x12345680	void *	ARG_1
0x00000004	0x1234567c	void *	ARG_0
0x00000000	0x12345678	void *	ARG_CODE

Table 2: An example system call argument stack.

Using a switch statement operating on the system call code, the respective system call function is called with its result being stored in the `eax` register if the function returns a value.

Using an invalid system call, or those outside the scope of this assignment, will cause the process to exit.

3.2 Validation

Validation of pointers into user memory is done using the functions found in Stanford's pintos project 2 documentation, although they have been modified to include a check that the address pointer argument is below `PHYS_BASE`. This strategy also requires modification of `page_fault` in `userprog/exception.c` as described in the project 2 documentation.

While the above strategy handles pointer validation, which is used for validating the system call code and arguments, they do not validate strings or buffers which require additional checks. For strings of unknown length, `get_user` is called for each byte in the string until `\0` is found. For strings of known length, `get_user` is called for every byte from 0 to the length of the string.

If a string or buffer is an argument for a system call, its respective handler function is responsible for validating it.

3.3 File synchronisation

As calling the functions in the filesys directory from multiple threads simultaneously is unsafe, a semaphore is used to prevent it. The semaphore is initialised to `1`, and decremented by any system call operating on a `file` structure before doing so, specifically `create`, `remove`, `open`, `close`, `read`, `write`, `seek`, `tell` and `filesize`. Once the function has completed, the semaphore is incremented, allowing any waiting threads to complete their file operations.

3.4 halt

`halt` simply calls `shutdown_power_off`.

3.5 exit

`exit` Stores the exit status of the process in the `exit_status` element of the thread structure and calls `thread_exit` which calls `process_exit`. `process_exit` outputs the exit status to stdout in the format `"$name: exit($status)"`.

`process_exit` is also responsible for cleaning up after an exiting thread, including updating its parent of its exit status, removing and de-allocating remaining child process structures and closing any open files and de-allocating any file map structures.

3.6 exec

`exec` is implemented in terms of `process_execute` as it needed to handle both child processes of a user program and the user program itself which is considered to be a child of the main kernel thread.

Before calling `process_execute`, `exec` first ensures that string representing the file to be executed is valid and does not cause any errors. If it does, it will immediately exit.

As `exec` is required to return `-1` in the case it could not execute the requested file, the `load` member of the `process` structure is used to allow the child to pass the success or failure of the file to the parent which is stored in the `load_status` member. If the file is loaded successfully, the threads id is returned as normal.

3.7 wait

`wait` is implemented in terms of `process_wait`. As with `exec`, this allows user programs to be waited upon by the main kernel thread if required.

`wait` returns the exit status of the thread if it has already exited. Otherwise, it will check if the thread is already being waited upon by using the `is_waited` member of the `process` structure and returning `-1` if it is true. If it is false, it sets `is_waited` to true, and checks the `is_alive` element of the process structure. If the child process has exited, denoted by `is_alive` being false, the exit status is returned immediately. Alternatively, it uses the `wait` semaphore of the `process` structure to wait for the child process to finish.

Once the process has exited, the exit status is retrieved. The child `process` structure is then removed from the `children` list of the `thread` structure and de-allocate, before the exit status is returned. Due to an already waited upon process having `wait` called upon its process id and an invalid process id returning the same result, this was deemed safe to do as well as saving memory for long running processes that use child processes.

3.8 create

`create` is implemented using `filesys_create`.

Before `filesys_create` is called, the name of the file passed as an argument is validated to ensure it does not cause errors using `is_valid_string`. If it does, then it will immediately exit.

In order to prevent multiple threads operating on files simultaneously, the synchronisation method described in §File synchronisation is used.

3.9 remove

`remove` is implemented using `filesys_remove`.

Before `filesys_remove` is called, the name of the file passed as an argument is validated to ensure it does not cause errors using `is_valid_string`. If it does, then it will immediately exit.

In order to prevent multiple threads operating on files simultaneously, the synchronisation method described in §File synchronisation is used.

3.10 open

In open, the name of the file to be opened is validated first to ensure it does not cause errors using is_valid_string. Next a file descriptor is generated by starting at the minimum valid value and incrementing it until it does not match an open file associated with the thread.

Once the file descriptor has been generated, a file_map structure is allocated and filesys_open is called to open the file. Both of these values are then added to the file_map structure before adding it to the files list of the current thread. The new file descriptor is then returned.

In order to prevent multiple threads operating on files simultaneously, the synchronisation method described in §File synchronisation is used.

3.11 close

In close, the file descriptor is used to look up a file_map structure in the current threads files list. When the file_map structure is found, its file member is passed to filesys_close.

To avoid memory leaks, the file_map structure is removed from the threads files list and de-allocated.

In order to prevent multiple threads operating on files simultaneously, the synchronisation method described in §File synchronisation is used.

3.12 read

In read, there is different behaviour depending on whether the file descriptor is for a file or stdin (0). If the file descriptor is for stdout (1), the process will exit immediately.

When reading from stdin, input_getc is used to read one character at a time until the maximum length of the buffer is reached. To validate the buffer, put_user is used, which is the opposite operation to get_user as described in §Validation. As with get_user, it is from Stanford's pintos project 2 documentation.

When reading from a file, the buffer is first validated using the method described in §Validation. Next, the file descriptor is looked up in the current threads files list from which the associated file structure is retrieved. If no match for the file descriptor can be found, the process will exit. The file read operation itself is implemented with file_read, applying the synchronisation method described in §File synchronisation.

3.13 write

In write, there is different behaviour depending on whether the file descriptor is for a file or stdout (1). If the file descriptor is for stdin (0), the process will exit immediately. Before any write occurs, the buffer containing the data to be written from is first validated using the method described in §Validation.

Writing to stdout is implemented using putbuf. If the buffer is greater than 512 bytes, then the first 512 bytes is output, and the end of the section is recorded. If the remaining data in the buffer is still greater than 512 bytes, this is repeated, otherwise the remaining data is output. This separation of long buffers was implemented due to a recommendation in Stanford's pintos project 2 documentation.

When writing to a file, the file descriptor is looked up in the current threads files list from which the associated file structure is retrieved. If no match for the file descriptor can be found, the process will exit. The file write operation itself is implemented with file_write, applying the synchronisation method described in §File synchronisation.

3.14 seek

`seek` uses the file descriptor to find the associated `file` structure within the current threads list of files. If no matching file descriptor can be found, the function will silently fail.

The seek operation is itself implemented using `file_seek`, applying the synchronisation method described in §File synchronisation.

3.15 tell

`tell` uses the file descriptor to find the associated `file` structure within the current threads list of files. If no matching file descriptor can be found, the function will silently fail.

The tell operation itself is implemented using `file_tell`, applying the synchronisation method described in §File synchronisation.

3.16 filesize

`filesize` uses the file descriptor to find the associated `file` structure within the current threads list of files. If no matching file descriptor can be found, the function will silently fail.

The filesize operation itself is implemented using `file_length`, applying the synchronisation method described in §File synchronisation.

4 Appendix

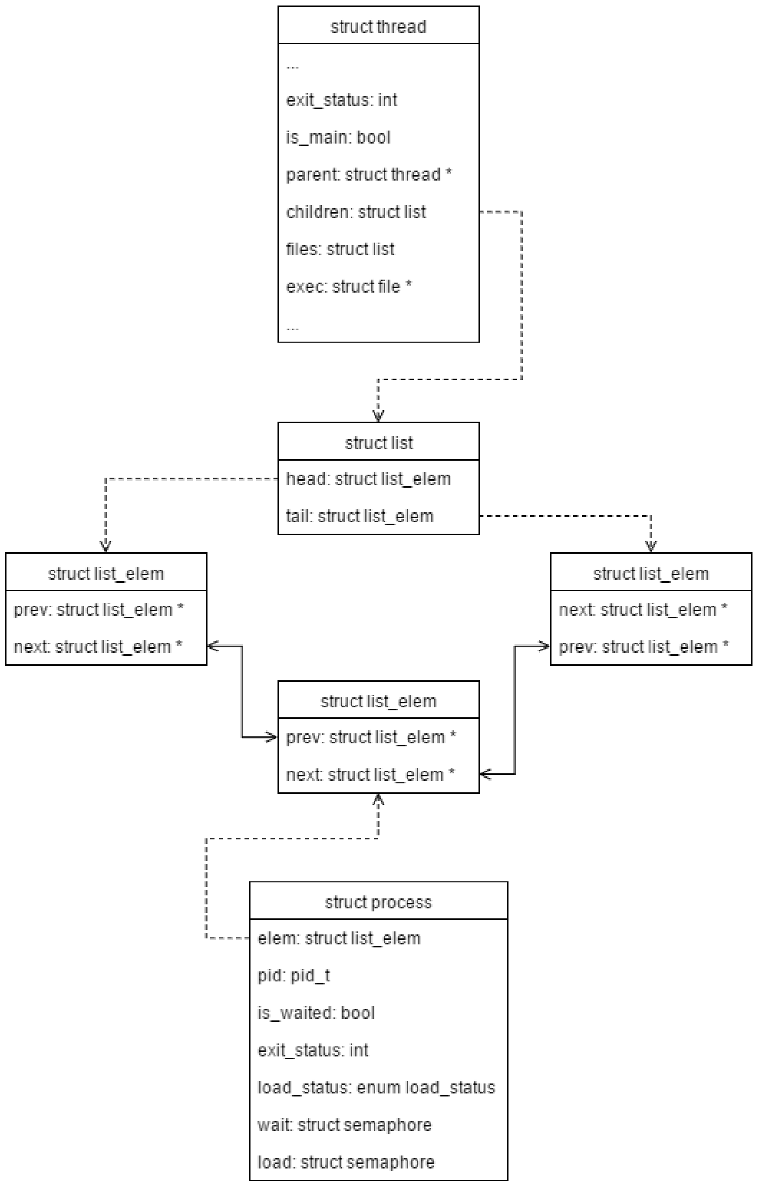

Figure 1: How a `process` structure links to a `thread` structure.

5 References

- Pfaff, B. et al. 2009. *Pintos Projects: Project 2 – User Programs* [Online]. Available at: https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html [Accessed 3 February 2017].
- Tanenbaum, A., Bos, H. 2015. *Modern Operating systems*. Harlow, England: Pearson.