Project User Programs Design

Group 06

Name	Autograder Login	Email
Seong Hyun Park	student366	spark6015@berkeley.edu
Ronald	student412	ronaldarifin@berkeley.edu
Noah	student89	noahku@berkeley.edu
Michael	student338	michael.wiradharma@berkeley.edu

Argument Passing

Data Structures and Functions

For argument passing, we will be using the stack to store our arguments given by the user. These are the steps we need to take before the user program starts executing:

1. Push the strings to the stack and maintain its address. We do this by first looping through `argv` and using `memcpy()`. While looping, we can store the address in a list and use this later in step 3.
2. We add in the null terminator sentinel, and if needed, we push stack-alignment beforehand. We need the sentinel because this follows C standard where `argv[argc]` must be a null pointer.
3. Going in reverse order looping through `argv`, we continue pushing the addresses of each of the argument strings to the stack.
4. We push the `argv` address to the stack, which is the address of the item that we last pushed.
5. We push the `argc` value to the stack, which corresponds to the number of arguments given.
6. Finally, we push the return value 0. This is a fake return address but is needed to follow the conventional stack format.

These steps are involved in `start_process` in the file process.c and will need to be implemented. If needed, we can separate these steps and put it in a separate function which would be called in start_process to make our code cleaner.

process.c

```
char *strtok_r(char *str, const char *delim, char **saveptr);
// Splits string by space

size_t strlcpy(char *dst, const char *source, size_t size);
// Prevents data race

void* palloc_get_page(enum palloc_flags flags);
// Allocates memory for parsed string -> palloc.c:93

void palloc_free_page(void *)
// Frees the memory allocated for the parsed string

void hex_dump(uintptr_t ofs, const void* buf_, size_t size, bo
ol ascii)
// DEBUG TOOL: stdio.c:560 --> hex_dump implementation.

static void start_process(void* file_name_)
// We need to modify this function

pid_t process_execute(const char* file_name)
// We need to modify this function
```

Algorithms

process_execute

To handle argument passing, we have to first look at the `process_execute` function in /userprog/process.c. This function takes in a string that is passed in by the user (file_name) which we can use to convert into individual arguments that can later be put onto the stack.

The first step is to split the string and store it in an array. We can use C's built-in function `strtok_r` and use the space character `" "` as a delimiter. It's important that we use `strtok_r` instead of `strtok` since this function is thread-safe. The way `strtok_r` works is that it takes a string, finds the delimiter, then saves the rest of the string at some given pointer. For example, if we call

```
char* token = strtok_r("arg1 arg2 arg3", " ", &some_save_locat
ion);
```

then "arg1" will be stored in token, and "arg2 arg3" will be stored in some_save_location. The separation of arguments is crucial for our processing.

Before directly using `strtok_r`, we handle errors in process_execute. If we notice that `file_name` has too many arguments or is too long, we can throw an error to tell the user to put less arguments. Additionally, if there are no arguments passed in, we also need to have a way so that errors won't be thrown and that our test case for no args will also pass.

After using `strtok_r`, we utilize the strlcpy function to create a copy of the token that we just parsed. By using this specific function, we can avoid data races between multiple threads that are concurrently accessing the arguments. Once we split the argument and copy it over, we pass this string into the thread_create function. The function `thread_create` takes in the arguments `name`, `priority`, `function`, and `aux`. In this case, we use the function by passing in `file_name`, `PRI_DEFAULT`, `start_process`, and `rest` respectively. Our `thread_create` function should look like this:

```
tid = thread_create(file_name, PRI_DEFAULT, start_process, fn_
copy);
```

start_process

In process_execute, we used the function `strtok_r` to split the first delimiter. For argument passing, we are splitting a given file name at the first space. In start_process, we will be handling when `strtok_r` is null. Because we know that the way `strtok_r` ends is indicated by the function returning null, we can check this with a while loop.

```
/* Example use case of strtok_r in start_process */
char* ptr = strtok_r(f, deli, rest);
while (ptr != NULL) {
  ptr = strtok_r(NULL, deli, rest);
}
```

This while loop can also be used to calculate `argc`. Because `argc` represents the number of arguments given in the file_name, we can store the amount of times `strtok_r` is called by incrementing an integer in the loop that represents `argc`. Once we have the corresponding `argc` and `argv` ready, we can use the load() function to push the address of the strings. When pushing the strings, we have to also include the null pointer sentinel at the end (at `argv[argc]`) to indicate when we have reached the end of all the arguments. The null pointer sentinel is also used to follow the coding standard in C, similar to how a null terminator operates for strings.

Synchronization

To ensure synchronization in our code, we use the function `strtok_r` instead of `strtok` since it is thread safe. The way `strtok_r` works is slightly different in that it finds the first instance of the delimiter, splits the string, and keeps the rest of the string at a specified pointer (the third argument given in `strtok_r`). The function `sema_init` has already been implemented for us in process_execute. Semaphores are used as a tool to ensure concurrency in our code. The semaphores in our code can be used to also avoid data races when we are calling other functions like `process_wait`.

Rationale

The design we chose for argument passing has a straightforward approach. Utilizing built-in C functions like `strtok_r` and strlcpy ensures thread safety, which is critical when working on a project that involves multithreading. Although there are other approaches to splitting strings and determining what arguments are passed in, these functions are easier to integrate to our project and can lead to less problems such as data races. If we ever need to tweak our implementation or swap out the functions, it should still be a relatively easy change and will not force us to rewrite too many lines of code.

Process Control Syscalls

Data Structures and Functions

```
static void syscall_handler(struct intr_frame* f UNUSED);
  // We need to modify this function
int process_wait(pid_t child_pid UNUSED);
  // We are using this for wait and needs to be modified
void shutdown_power_off(void);
  // We are using this to implement halt part
pid_t process_execute(const char* file_name);
  // We are using this to exec
void shutdown_power_off(void);
  // This function is for halt

// This is used for both parents and children. The parent waits for the child's load()
typedef struct shared_data {
  sema_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  tid_t pid; // Parent
  tid_t cid; // Child
  struct list_elem elem;
}

struct thread {
  /* Owned by thread.c. */
  tid_t tid;                 /* Thread identifier. */
  enum thread_status status; /* Thread state. */
  char name[16];             /* Name (for debugging purposes).
*/
  uint8_t* stack;            /* Saved stack pointer. */
  int priority;              /* Priority. */
  struct list_elem allelem;  /* List element for all threads l
ist. */
  /* Shared between thread.c and synch.c. */
  struct list_elem elem; /* List element. */
#ifdef USERPROG
  /* Owned by process.c. */
  struct process* pcb; /* Process control block if this thread
is a userprog */
  struct list children; /* For the parent to keep track of */
#endif
  /* Owned by thread.c. */
  unsigned magic; /* Detects stack overflow. */
};
```

Algorithms

To manage the process control syscalls, we need to implement several functions that were listed. These include `practice`, `halt`, `exec`, `exit`, and `wait`. The exit function has already been implemented, so we will only focus on four functions in our design document. We handle the functions by editing syscall.c and using a switch case for `args[0]` (Examples: args[0] = SYS_PRACTICE, so we handle the command for practice by checking `args[0]` in the switch case).

practice

Practice is relatively simple. We are asked to return an integer that is passed in and increment it by one. What we can do here is to use the frame called `f` and use its `eax` to return a value to the user. We directly modify `f->eax` and set it to `args[1] + 1`. Once we modify the `eax`, we can also print it in this format.

```
printf("%s: practice(%d)\n", thread_current()->pcb->process_na
me, f→eax);
```

In our switch case, we are looking for `SYS_PRACTICE`. After we check for `SYS_PRACTICE`, we can tell our program to run what was stated above.

halt

Halt has been implemented for us. We can call the function `shutdown_power_off()`, which is located in src/devices/shutdown.c. All we have to do is to call this function when `args[0]` is `SYS_HALT`, which the switch case checks for.

exit

Exit has already implemented for us according to the project specification.

Although exit has been implemented for us, we also need to modify it by closing all the open file descriptors that were passed in through the argument and ones that are in the process.

exec

Exec uses process_execute, which is implemented for us as well. We first create a `char*` variable called `cmd` and store `args[1]`. We call process_execute and use `cmd` as our input.

```
// We store cmd
char* cmd = args[1];
// Return the result of process_execute, which could be a -1 i
f it errors
f->eax = process_execute(cmd);
```

Given the frame `f`, we can pass in the result of process_execute(cmd) and put it into `f->eax`. In case the output is an error like `TID_ERROR`, we return -1 to indicate that something went wrong when the user used the command exec. This can be handled in `start_process` by checking the `success` boolean.

wait

Wait has a few more components compared to the other functions. Because this is related to threads, we will need to use semaphores and create a system where we can acquire and release certain functions.

Something we have to check for is whether or not the child process called `pid` is alive. If it is not alive, we can return -1 by setting `f->eax` to -1. Now, we examine `argv[1]`. The value from `argv[1]` is the value of `pid` that we will be using. Using this value, we can call `process_wait()` which is stored in process.c. In this function, we can use the current child we have and compare it to the argument we passed in from syscall.c.

We also need to make a list of children to keep track of the data that each child has. It will be easier for us to track each child's data if we iterate through the Pintos list that we make. When we're monitoring the children, we need to make sure they share the same exit code and also the same semaphore between parents and children. When we create data for the children and parent, we need to allocate each of them so that they stay on the heap. This allows us to have access once a child or parent finishes their process.

In order to make the parent wait for its children, we need to use acquire and release functions. We can use `sema_down` to decrement its nonnegative integer and also decrease the ref_count by one. At the end when all the related processes are complete, we check if the `ref_count` is 0. If it is, we free all the shared data that has been used by the parent or children.

Synchronization

Synchronization will be handled when we call process_wait since we are dealing with semaphores. In process_wait, we will use semaphore functions whenever we are changing the variables that the child holds. To use these semaphore functions, we can pass in the struct that we have created above (called `process_wait` that uses child_pid). We call these functions every time we decrement `ref_count`. Specifically, we will be calling `sema_down` when we are trying to access `ref_count` and `sema_up` after we finish decrementing. If we implement it in this way, we can avoid possible data races in our code.

Rationale

The structures, algorithms, and synchronization methods we chose are meant to ensure reliability and efficiency. A key component that we implemented is the `struct shared_data`, where data sharing is facilitated between the parent and the child. The use of dynamic allocation also allows us to ensure persistence of information like the exit codes and synchronizing variables. A lot of our functions make use of the pre-implemented functions as well like `shutdown_poweroff` and `process_execute` to make our code less complicated. Different approaches such as using other data structures instead of our `shared_data` struct and using other algorithms could produce inefficiencies and new errors. Replacing semaphores with locks could result in resource wastage or unexpected results, which is the main reason why we decided to go with the current implementation.

File Operation Syscalls

Data Structures and Functions

filesys.c

```
bool filesys_create(const char* name, off_t initial_size);
  // Use this to create a new file
bool filesys_remove(const char* name);
  // Use this to delete the file named "file"
struct file* filesys_open(const char* name);
  // Open file and return value is stored inside of thread
```

file.c

```
off_t file_length(struct file* file);
  // file size
void file_seek(struct file* file, off_t new_pos);
  // file seek
off_t file_tell(struct file* file);
  // file tell
void file_close(struct file* file);
  // file close
off_t file_read(struct file* file, void* buffer, off_t size);
  // file read
off_t file_write(struct file* file, const void* buffer, off_t
size);
  // file write
```

thread.h

```
typedef struct thread {
 //...
  struct list files;
  // Linked List of type file
  int fd;
  // To avoid same fd, we increment this one every time this i
s used. Used as counter
}
// We have to initialize values like files and fd. fd should b
e 2
```

process.h

```
typedef struct file_elem {
  int file_descriptor;
  //file descriptor
  struct file* f;
  //file pointer that points to the file
  struct list_elem elem;
  //for thread's list files
}

int put_files_and_get_descriptor(struct *file);
  /* This function is the function will make one which in case
of args[0] == SYS_OPEN
  this will store file information into current thread and ret
urn to file
  descriptor */

struct *file get_file(int fd);
  // This will return file object based on the file descriptor
```

File operation syscalls involve a restructuring of our data structures and functions. Whenever we first open a file and use any functions given from file.c, we have to call functions that we implement ourselves. When we use `filesys_open`, we put the file we are working with into the current thread. For this portion of the project, we will be implementing a function called:

```
int put_files_and_get_descriptor(struct *file);
```

The way this function works is that it will help us store the corresponding file information into the current thread we are working with and will return the file descriptor. `put_files_and_get_descriptor` will first allocate space for the struct called `file_elem`. In this file elem, we initialize the `file_descriptor` with `thread_current()→fd`, increase the `thread_current()→fd` by one to avoid duplicate fd numbers, and initialize file `f` from the parameter values.

Now, we will focus on how we will implement `get_file`. First, we will get the list from `current_thread()`. Next, we iterate through that list (which is from `struct list files`) until we find the file based on the given descriptor. We return the matching file if we find one, and if we cannot find it, we return `NULL`.

Algorithms

create

The create operation will be using the given arguments to create the file. We can utilize a function that has already been implemented for us in filesys.c, which is

```
bool filesys_create(const char* name, off_t initial_size);
```

We can use this function by passing in `argv[1]` for the parameter `name` and pass in `argv[2]` for the parameter `initial_size`. Once we call `filesys_create`, we assign the given value to `f→eax` for our return value.

remove

The remove operation is simple because of how it is already implemented in filesys.c and how it is similar to create. The function is

```
bool filesys_remove(const char* name)
```

We pass in `argv[1]` into the name parameter and store the result in `f→eax`. Something to note is that files that are either open or closed can still removed, and removing an open file does not mean it will be closed after running the command.

open

The open function will be relatively simple. To open a file, we will be using the function called

```
struct file* filesys_open(const char* name);
```

which will give us the file. After we retrieve the file by calling `filesys_open`, we can store it in our current process by calling the function:

```
int put_files(struct *file);
```

The way we will implement `put_files` has already been mentioned above. After putting the files, we can return the integer value we get from calling `put_files` and store it into `f→eax` as a way to return the value to the user. If a file cannot be opened, the functions will return -1 back to the user.

filesize

The filesize function is rather simple. We search through our list of files using the function we implemented ourselves called `get_file` which returns a `struct *file` after searching through the list for the corresponding file. Once we call `get_file`, we can pass our result into the function:

```
off_t file_length(struct file* file)
```

and return the given result by setting it in `f→eax`.

read

We first call the `get_file` function that we have implemented earlier to get a file object and use:

```
off_t file_read(struct file* file, void* buffer, off_t size);
```

to read the file given to us. For the parameters `void* buffer` and `off_t size`, we pass in `args[2]` and `args[3]` respectively. One thing to note is that when we pass in `args[2]`, we must cast it as a (void *) type. If there are any errors with reading or if we cannot read the file, we return -1 back to the user by assigning it to `f→eax`.

write

We first call the `get_file` function that we have implemented earlier to get a file object and use:

```
off_t file_write(struct file* file, void* buffer, off_t size);
```

to write the file given to us. For the parameters `void* buffer` and `off_t size`, we pass in `args[2]` and `args[3]` respectively. One thing to note is that when we pass in `args[2]`, we must cast it as a (void *) type. If there are any errors with writing or if we cannot write the file, we return -1 back to the user by assigning it to `f→eax`.

seek

The seek function is rather simple. We search through our list of files using the function we implemented ourselves called `get_file` while checking for `argv[1]` which returns a `struct *file` after searching through the list for the corresponding file. Once we call `get_file`, we can pass our result along with `argv[2]` into the function:

```
void file_seek(struct file* file, off_t new_pos)
```

to seek that specific file. Assuming this fails in the `get_file` section of the code, we will return -1 to the user.

tell

The tell function is rather simple. We search through our list of files using the function we implemented ourselves called `get_file` which returns a `struct *file` after searching through the list for the corresponding file. Once we call `get_file`, we can pass our result into the function:

```
off_t file_tell(struct file* file)
```

and return the given result by setting it in `f→eax`. Assuming this fails in the `get_file` section of the code, we will return -1 to the user.

close

We get the file by using `get_file` from the `file_descriptor`. If we are unable to find the file that was given to us, then we return -1. We then we call the given function:

```
void file_close(struct file* file)
```

to help us close the file given to us.

Synchronization

According to the specification, the file system used by Pintos is not file safe. This means that we need to add a global lock so that our syscalls do not call different file functions at the same time. The specification states that although a more detailed implementation will be made in a later project, we can stick to using global locks for these file operation syscalls.

While we are dealing with files, it is necessary to have a lock since we do not want to have any data races. We will have to use locks whenever we call our file functions or modify the file structure. We implement this by using the locks before every syscall. We treat each file syscall as a critical section.

Rationale

Many of the syscalls related to file operations make use of the pre-implemented functions given to us in file.c and filesys.c. To utilize these functions and pass in the right parameters, we decided to use a Pintos list to keep track of all the file descriptors and elements. By doing so, we can easily search through the list and return any corresponding files in our `get_file()` implementation. Our other function `put_files_and_get_descriptor()` also makes use of this data structure and makes it more convenient for us to insert files.

By storing our data in such a way, we can easily call the functions given to us. Another way to go about this is to use a hash map to store all the files for the file descriptor table, but for this current project we felt that it would be easier to implement a list. Although the hash map is a possible alternative that does not have too many drawbacks compared to the list data structure, the list is easier to visualize and has slightly less caveats, such as not needing to store values with unique keys.

Floating Point Operations

Data Structures and Functions

```
- orl $CR0_PE | CR0_PG | CR0_WP | CR0_EM, %eax
+ orl $CR0_PE | CR0_PG | CR0_WP, %eax

// C
struct intr_frame {
  ...
  uint8_t fpu_state[108]; /* Save FPU onto intr_frame through assembly. */
  ...
};

// Assembly
```

```
.func intr_entry
.func intr_exit

struct switch_threads_frame {
  ...
  uint32_t ebx;          /* 12: Saved %ebx. */
  void (*eip)(void);     /* 16: Return address. */
  uint8_t fpu_state[108] /* Save FPU onto stack frame to switc
h and save value. */
  struct thread* cur;    /* 20: switch_threads()'s CUR argumen
t. */
  struct thread* next;   /* 24: switch_threads()'s NEXT argume
nt. */
};

struct thread* switch_threads(struct thread* cur, struct threa
d* next);

Functions in Assembly: // i think this should go inside proces
s.c (start_process
fninit - initializes FPU to be ready for use by a new process.
fnsave - saves all 108 bytes of FPU into a specified memory ad
dress.
frstor - restores the FPU registers from a memory address.
```

Algorithms

In general the algorithms for the starting a new thread or process will always start with an fninit instruction call. During any context, thread switches, or syscalls, we call fnsave before the switch, and frstor to restore the FPU state before the swap.

Setting FPU for interrupts

We will need code both in assembly and C in order to initialize the FPU for interrupts and context switching. We added a `uint8_t fpu_state[108]` array in the intr_frame in order to store the state of the FPU registers, which will be loaded and pushed onto the `intr_frame` struct stack. We use `fsave` to the appropriate memory address in the stack. Then we initialize the FPU for the new context switch in assembly using `fninit`. Lastly, we can run `frstor` to restore the FPU state values from the array stored in `intr_frame` struct after the context switch has been completed, located in `intr_exit`.

Setting up/initializing FPU for thread switch

This is very similar to the set up for interrupts. The function switch_threads will save the FPU registers onto `switch_threads_frame` using `fsave` in assembly. This can be achieved by modifying the `switch_threads` function in `switch.S`. Then using `fninit` in assembly, we can initialize the FPU registers for the next thread to use. This is done after the thread stack movement to the new addresses. Finally, we can restore the FPU registers (if any) that are currently on the stack and bring them back into the registers using `frstor`.

Syscall compute_e

As with other syscalls, this will done in syscall.c. We start with the usual check

```
if args[0] == SYS_COMPUTE_E
```

Then we validate if args[1] is a positive integer. As this syscall requires the FPU, we'll prepare it by calling `fnsave` at the beginning of the function and store it into an `uint8_t fpu[108]` array stored on the heap via `malloc`. This is followed by an `fninit` call to initialize the FPU. The `lib/user/syscall.c` contains the function `compute_e`, which we call and store the return value into `f→eax`. Finally, call `frstor` from `fpu[108]` to restore the FPU state before the syscall.

Synchronization

By storing FPU register values on the stack, we are dealing with synchronization of the threads. We allow switching between stacks and all various processes and threads are able to use the FPU register. However, there is no other synchronization beyond this as in general, we won't allow threads and processes to use the same FPU register values. This is similar to regular GPR registers. Such values should just be stored as regular primitives on the stack.

Rationale

A few design choices were made in this to enable floating point computation in PintOS. The first includes storing the FPR values in a uint8_t array of 108 bytes. We decided that this is the best way to store all the memory bytes from the ST states from the FPU. We wanted a way to 'dump' all of the bytes into the stack and let FPU instructions handle the rest. We considered other types such as char* arrays or just a regular floating point primitive, however, these felt less appropriate.

Next, we decided to store the FPU register values in the intr_frame and switch_threads_frame as opposed to the thread or process structs themselves. We felt that this is more fitting as this puts them all in the same place as other additional GPR registers that are also stored on the stack in a similar way between context and thread switches. This way, the values can be restored immediately on a switch and aren't attached to the thread struct.

We also made a design choice in how we store the FPU registers in compute_e syscall. We considered implementing a general `fnsave` call for every syscalls but this felt unnecessary as not all syscalls require the FPU register, thus we decided to localize it within our implementation of compute_e.

Concept Check

1. We should take a look at `src/tests/userprog/sc-bad-sp.c`. This test uses a virtual memory that is 64 MB below the user space (as seen in the comment block from `sc-bad-sp.c`). This test should automatically fail purely from trying to access an address that is negative and way too far from user space. This will try to access memory that is not initialized, so it will return with -1.
2. We should take a look at `src/tests/userprog/sc-bad-arg.c` according to the virtual memory layout in the Pintos Documentation, it tells us that the virtual

address goes up to `0xc0000000`. Since the esp is being set to `0xbffffffc`, which is 4 bytes from the boundary, we will reach it after passing in the argument SYS_EXIT as `argv[0]`. Thus the argument to SYS_EXIT call will be able the PHYS_BASE boundary.

3. One of the file operation syscalls that has not been covered is the `remove` function. The two possible results of calling remove is to get a true or false. First, `remove-true.c` which checks if the file is removed successfully. We would first create a file and try to remove it so that there is something to remove. Secondly, we'll create `remove-false.c` which checks if the file is not removed successfully. We will use the remove function without creating an actual file so that it should return false since the function was unable to locate that specific file.