Project User Programs Design

Argument Passing

The argument to the process_execute() function is a constant string. Per instruction, we need to so that it is able to parse the argument such that the main function of the user process will receive the appropriate argc and argv.

Argc represents the number of words in the string and argv represents the words

e.g.

```
process_execute("ls -ahl")
argc = 2
argv = ["ls", "-ahl"]
```

To do this, we need to ......

Now that we have the arguments, we need to be able to put the arguments for the initial function on the stack before it allows the user program to begin executing, as well as follow x86 calling convention.

Right before asm volatile("movl %0, %%esp; jmp intr_exit" : : "g"(&if_) : "memory");

Data Structures and Functions

Argv is the array of the input, separated by spaces.
e.g.

```
process_execute("ls -ahl")
argv = ["ls", "-ahl"]
```

We will store argv in a string array:

```
char* argv
```

Arc is the length of argv, which represents the number of words/tokens in the command line argument.
e.g.

```
process_execute("ls -ahl")
```

```
argc = 2
```
We will store argc using an integer variable:
```
int argc
```

Algorithms

Now that we have the arguments, we need to be able to put the arguments for the initial function on the stack before it allows the user program to begin executing, as well as follow x86 calling convention.

do it before asm volatile so that the thread stack is setup.

Questions for Gaurav d

Synchronization

In order to ensure proper synchronization, we will use strtok_r() as it is considered to be thread-safe. The two options for parsing strings and converting them into tokens were either strtok() or strtok_r() in C. But strtok_r() is a reentrant version of strtok(), hence it is thread safe. This is to ensure that we do not come across data races and shared conditions when we do multiple calls/multi-threading.

Reference: https://www.geeksforgeeks.org/strtok-strtok_r-functions-c-examples/

Rationale

The algorithm for parsing the argument takes o(n) times (n = length of argv array). No matter what, we have to parse the arguments given and then follow x86 convention by placing argc and argv onto the stack so this is necessary no matter what.

Process Control Syscalls

Brainstorm Notes:
Understand relation to Argument Passing
How to distinguish/handle different syscalls: Syscall number?

Syscall number and arguments pushed on stack via x86 convention before interrupt
syscall_handler then accesses the stack to get the syscall number
syscall_handler gets the caller's stack pointer by getting the esp of struct intr_frame, and struct intr_frame is on the kernel stack
Avoid race condition/manage synchronization
In Pintos, syscalls are invoked by int $0x30

Design Questions:

1. How can we safely read and write memory that's in a user process's virtual address space? If pointer is null or invalid or buffer, how to avoid kernel crashing?
2. How to DRY when implementing each syscall?

Data Structures and Functions

We can add more variables to the Process struct in process.h:

Actual ones we need:

```
struct process {
    uint32_t* pagedir;          /* Page directory. */
    char process_name[16];      /* Name of the main thread */

    // new data structures
    struct data_node data_shared_with_parent; /* Shared data n
ode with the parent */
    struct thread* main_thread; /* Pointer to main thread */
    struct list data_nodes; /* List of data nodeswhichis a pro
cess struct*/

};
```

```
struct data_node{
        pid_t pid;
        tid_t tid;
        struct semaphore sema;  /* Semaphore for waiting on ch
ild processes */
        int exit_status;             /* This process's exit st
atus */
        int ref_count;
        bool child_has_exited;                /* True if the p
rocess has exited */
        bool child_has_waited;                /* True if the p
rocess has been waited */


}
```

Algorithms

Practice

```
int practice (int i)
```

We will implement this functionality in the syscall_handler function in file syscall.c

```
if (args[0] == SYS_PRACTICE){
  // Check if the argument exists:
  if (args[1] == NULL){
      //Do error handling
  }else{
      args[1]++;
      f->eax = args[1]
  }
}
```

halt

```
void halt (void)
```

```
if (args[0] == SYS_HALT){
  // Check if the argument exists
  printf("Shutting down Pintos!");
  shutdown_power_off();
 }
```

exit

```
void exit (int status)
```

Goal: Terminates the current user program, returning status to the kernel.
print out the exit status of each user program when it exits. The format should be %s: exit(%d)

Malloc the list before thread_create()

We do not need to verify the validity of a user-provided pointer, then dereference it.

a. **Print Exit Status**: Print the exit status using the process's name, which should be stored in the data node structure associated with the current thread.
b. **Update Parent's Child Process Structure**:
   - If the exiting process is a child (which can be determined if it has a non-null parent), store the exit status in its `data_node` structure. This is the shared data structure between the child and parent.
c. **Signal Parent Process**:
   - If the process is a child, up the semaphore in the `data_node` structure to unblock the parent process waiting on the child's exit status.
d. **Decrement Reference Counts**:
   - Decrement the reference count in the `data_node` structure of the parent.
   - If the reference count is now zero, free the `data_node` structure.
   - Similarly, decrement the reference counts for any child data structures that represent children of the current process. If any counts reach zero,

remove the **`child_process`** from the process's data_node list and free the structure.

e. **Terminate the Thread**:
   - Terminate the current thread by calling **`thread_exit()`**. This should trigger the release of the thread's resources, including the **`struct thread`**.

exec

```
pid_t exec (const char *cmd_line)
```

Goal: execute the program whose name is passed in the cmd_line.
We also need to use the global file lock during the exec syscall above so that it doesn't modify a file during the syscall.
Algorithm:
a. need to first verify the validity of cmd_line. This pointer can point to an invalid address. steps for validating the pointer:
- check pointer is below the PHYS_BASE using function `is_user_vaddr` in `vaddr.h`
- try reading a byte from the pointer using `get_user`, and write the same byte back using `put_user`
- if the operations didn't fail, the pointer is valid.
- otherwise, the pointer points to an invalid address, and exec returns -1

b. call `process_execute`. when creating the new data_node for this process, initialize the semaphore to have value 0, and have the parent process do down on it.
c. after `load` call is done in `start_process`, up the `load_sema`, this will finally unblock the exec
d. for error checking: check if the returned tid is valid or a `TID_ERROR`
- If it doesnt run correctly, then we need to free the shared data.

preventing write: input program will be loaded in as a file, so `file_deny_write()` to the fd

wait

```
int wait (pid_t pid)
```

Goal: We want to ensure that a parent process can wait for a specified child process to finish executing and obtain its exist status.

Assumption: The exit syscall is implemented correctly (properly updating the struct fields).

Algorithm:
Use pid to get a pointer to the child_process, we can try to accomplish this using the list_entry function inside a for loop. If, during the loop, the pid of the child_process matches our desired pid, we save the child_process pointer and break from the loop. If the pid is not a valid one, we return -1 for error.

Check has_waited. If true, return -1.

Then, we check that the `child_process` we just retrieved has not already exited. If it has not exited, we need to make sure that the parent waits for the child to finish. In this case, one way we might make the parent be aware of the unfinished `child_process` is to try decreasing the `exit_sema` using the `sema_down` built in function. When we create a `child_process`, we intialize the `exit_sema` to be zero. This is helpful in this case since `sema_down` is essentially blocking the parent process until the `exit_sema` increments to 1 for `sema_down` to finish its operation, which only happens when a child exits. At this point, the `child_process` should have exited, so we want to remove the now useless `child_process` from the parents list and free the memory occupied by the `child_process ` object. Lastly, return `exit_status` as required.

If the `child_process` has already exited, we remove the now useless `child_process` from the parents list and free the memory occupied by the `child_process` object. Lastly, return exit_status.

Synchronization

Reference counts in the shared data structures are protected by a lock to prevent race conditions.
No additional synchronization is required for the exit code or success information because a shared semaphore already provides the necessary synchronization.

Rationale

We decided to make modifications to the `process` struct so that it includes a list of data_nodes. A data_node serves as the connection between a parent process and the child processes. It serves as a shared data structure, so that semaphores and ref

count values can be access between the parent and the child processes, while preserving states of the child processes even after they exit. Our previous design included a `child_process` struct, but that introduces unnecessary complexity and differentiates between the a child process and a parent process, despite both being processes. If the parent needs information about the child process, it can find that process in the list `data_nodes` in O(n) time.
During `exec`, since the thread creation completes before loading of the program, we modify the `thread` struct by adding a load semaphore, so the parent process can start to wait as soon as the thread is created, and only after the loading finishes the parent process is unblocked, thus ensuring that it knows whether the load is successful or not.

File Operation Syscalls

Data Structures and Functions

```
struct process {
    ...
    struct list file_descriptor_table; /* List of file descriptors for this process*/
    int fd_next; /* the next unique fd number, this should start with 3. */
    ...
};

struct file_descriptor {
    int fd; /* the file descriptor id number. 0, 1, 2 reserved */
    char *file_name; /* name of the file pointed to by this descriptor. this may be slow for search though*/
    struct file* file_ptr; /* file pointer */
    struct list_elem elem; /* list member */
};
```

// Struct file has position and size
// global file lock variable
// during syscall_init, implement global file lock.
// open i nodes

Algorithms

All operations will start with aquiring the global lock and end with releasing the lock before the return.

```
file_descriptor* find(list fd_table, int fd):
```

traverse fd_table until we find fd, if we do return the element
if we reach the end of the list and we do not find it, return null

```
bool create (const char *file, unsigned initial_size)
```

- Validate the input parameters to ensure none are `NULL` and that the specified file is within user memory.
- Add a new entry to the File Description Table.
  - Use the `filesys_create` function from `filesys.c` to create a new file entity.
  - Retrieve the file's identifier using the `file_count` variable.

// to generate file descriptor table id. global variable that adds

```
bool remove (const char *file)
```

- Perform checks on the input parameters to confirm none are `NULL` and that the file resides within user memory.
- Utilize the `find` function on the linked list to ascertain the file's existence.
- If the file is found, proceed with the `filesys_remove` function from `filesys.c` to delete the file.

```
int open (const char *file)
```

Opens a file and get the fd number of that file

- Utilize the `find` function on the linked list to ascertain the file's existence.
- If the file is found, proceed with the `filesys_open` function from `filesys.c` to open the file
- Update the `file_ptr` to be the return value of `filesys_open`

- return the `fd` of the found file

```
int filesize (int fd)
```

Get the file size of a file

- Utilize the **`find`** function on the linked list to ascertain the file's existence.
- If the file is found, return the **`size`** of the file descriptor's `file`'s `inode`

```
int read (int fd, void *buffer, unsigned size)
```

Goal: Read `size` bytes from the file open as `fd` into `buffer`.

Algorithm:

- If input buffer pointer is valid, return -1 if invalid.
- Check fd to see if we are dealing with standard input.
- If fd belongs to standard input, we could use a for loop with the built in input_getc() function to read the standard input character by character (character is a byte) up to a total of size into the buffer. Then, we return size as required.
- If fd does not belong to standard input, we use fd to find the file descriptor. Ensure error handling here.
- Secure a lock here.
- Read the specified number of bytes from the file into the buffer. Track the number of bytes read.
- Release the lock.
- Return the number of bytes read.

```
int write (int fd, const void *buffer, unsigned size)
```

Goal: Write `size` bytes from `buffer` to the open file with file descriptor `fd`.

Algorithm:

- Check that buffer pointer is valid, if not then error
- If we use standard output (check with fd), use the built-in putbuf function to write size number of characters directly into standard output. Then, we return size.
- If not stdout, we get the file_descriptor object pointer and write size bytes from buffer into the file_descriptor object using file_write built-in function. Before and after writing, we need to lock and unlock to prevent race condition. We might also

need to further check the validity of the file we are writing to for possible edge cases. Ensure proper error handling. Lastly, return the number of bytes written.

```
void seek (int fd, unsigned position)
```

Goal: Changes the next byte to be read or written in open file `fd` to `position`, expressed in bytes from the beginning of the file.

Algorithm:

- Use fd to get the file_descriptor object pointer. Handle error when needed.
- Use file_tell built in function to get the position.
- Return the position.

```
int tell(int fd)
```

Goal: get the position of the next byte to be read.
Algorithm:

get the `file_descriptor` struct, and return the `pos` variable of the `file` struct.

```
void close(int fd)
```

Goal: close the opened file
Algorithm:
free the file pointer with `file_close` and file name
free the `file_descriptor` struct from the heap

for `precess_exit`: before it exists, get it's file descriptor table, free all file descriptor structs.

Synchronization

In order to ensure that only one thread can do a file operation syscall, we implement a global lock in `syscall.c`. Before we make a file operation syscall, the `pthread_mutex_t` lock will be locked. it remains locked until the syscall finishes, and unlocked before we return from syscall_handler.

We also need to use the global file lock during the exec syscall above so that it doesn't modify a file during the syscall.

Rationale

Using two different ways to keep things in sync—semaphores and locks—makes the system work better because it lets us handle timing and access separately. Also, adding a counter that tracks how many processes are using shared data helps us clean up unused data right away. This method greatly improves how efficiently memory is used.

Floating Point Operations

Data Structures and Functions

In `threads/switch.h`

```
/* switch_thread()'s stack frame. */
struct switch_threads_frame {
  uint8_t fpu_state[108]; /* Saved the FPU state. */

  uint32_t edi;        /*  0: Saved %edi. */
  uint32_t esi;        /*  4: Saved %esi. */
  ...
};
```

In `threads/interrupt.h`

```
struct intr_frame {
  /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
  uint8_t fpu_state[108]; /* Saved the FPU state. */

  uint32_t edi;       /* Saved EDI. */
  uint32_t esi;       /* Saved ESI. */
```

```
  ...
};
```

In `user/syscall.c` and `user/syscall.h`:

```
double compute_e (int n)
```

Algorithms

Initializing the FPU: create a clean slate of FPU registers

In `threads/start.S`

- Remove the `CR0_EM` bit and indicate that the FPU is present
- Initialize the FPU with `finit`

In `threads/thread.c/thread_create`

- Initialize the FPU with `finit` for a new thread/process.
- Requires `asm`

In `threads/switch.S/switch_threads`

- After saving the FPU state of the parent thread, initialize the FPU for the child thread with `finit`

In `threads/intr-stubs.S`

- After saving the FPU state of the process/thread, initialize the FPU for the interrupt handler with `finit`

Saving the FPU: save FPU state on the stack during thread/context switches to preserve current thread's floating point data

In `threads/switch.S/switch_threads`

- Allocate 108 bytes on the stack by decrementing `esp`
- Save the FPU state onto the stack using `fsave`

In `threads/intr-stubs.S/intr_entry`

- Allocate 108 bytes on the stack by decrementing `esp`
- Save the FPU state onto the stack using `fsave`

Restoring the FPU: restore FPU state before resuming thread execution

In `threads/switch.S/switch_threads`

- Restore the FPU state from the stack using `frstor`

In `threads/intr-stubs.S/intr_exit`

- Restore the FPU state from the stack using `frstor`

```
double compute_e (int n)
```

- Validate input `n` , cannot be null or nonpositive
- Call `lib/float.c/sys_sum_to_e`, store the result in `eax` for return

Synchronization

We do not need to implement synchronization for floating point operations. During thread/context switches, the current FPU state is stored on the stack, the FPU is initialized again to create a clean slate of FPU registers for the new thread, and previous FPU state is restored from the stack. With this implementation, the FPU state belongs to the thread/process that is currently running, and the FPU state is saved during thread/context switches. As such, FPU registers are not shared between threads.

Rationale

By saving and restoring a thread's FPU state on the stack during thread/context switches, we're able to ensure that each thread retains its FPU state when resuming execution. With this implementation, we've created the abstraction that each thread has its own exclusive set of FPU registers. This implementation is also rather simple, as it does not require any synchronization methods between threads.

Concept check

1. In child-bad.c, we directly manipulates the address stored in the esp to a random address 0x20101234 (line 10). This address is pointing to unmapped memory as the code can't just directly choose which memory to use, dynamic memory allocation should handle this instead. When trying to access this invalid memory at the start of the syscall, the syscall should recognize that this address is unmapped, and subsequently terminate the process. Otherwise, we fail the test (line 11).
2. Test name: exec-bound-2.c. The test first gets a pointer `p` that is 5 bits away from the boundry of good & bad pointers (line 11). this way, only the first byte (4 bits) from this pointer maps to valid memory, while every subsequent byte will not. Then, the `SYS_EXEC` interger is saved to the address pointed to by `p` (line 12). In the assembly line (line 16), the address stored in `p` is passed into the `esp`, so when

we start the syscall, we should check the address in `esp` and exit, since it's invalid. Otherwise, we fail the test (line 17).

3. for `seek` and `tell` syscalls, the test cases provided doesn't test for when an invalid fd is passed in. It only uses `seek` in `rox-child.inc`, but more tests for the behaviors when an input error happens should be implemented.